Секреты программирования во Flash MX

2-е издание

Предисловие Гэри Гроссмана, создателя ActionScript

ActionScript *для* Flash MX

Подробное руководство

СИМВОЛ®

O'REILLY®

Колин Мук

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-068-5, название «ActionScript для Flash MX. Подробное руководство» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

ActionScript for Flash MX

The Definitive Guide

Second Edition

Colin Moock

O'REILLY®

ActionScript для Flash MX

Подробное руководство

Второе издание

Колин Мук

Санкт-Петербург — Москва
2004

Колин Мук

ActionScript для Flash MX
Подробное руководство, 2-е издание

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*М. Антипин*
Редактор	*В. Овчинников*
Корректор	*С. Беляева*
Верстка	*Н. Гриценко*

Мук К.
ActionScript для Flash MX. Подробное руководство. – Пер. с англ. – СПб: Символ-Плюс, 2004. – 1120 с., ил.
ISBN 5-93286-068-5

Второе издание бестселлера «ActionScript для Flash MX. Подробное руководство» – исчерпывающий и незаменимый ресурс по ActionScript, настольная книга дизайнеров и Flash-программистов, пишущих серьезные приложения. Из-под пера Мука вышел полностью переработанный справочник, не только подробно описывающий бесчисленные нововведения Flash MX, но и раскрывающий в ясной и доступной манере глубинные изменения, произошедшие на методологическом уровне (особенно касающиеся компонентов и событий).

Издание уникально – это самая точная и хорошо организованная книга по программированию во Flash MX. Объем справочника увеличился почти вдвое и включает описание более 250 новых классов, объектов, методов и свойств. Мук добавил в книгу сотни новых примеров кода, иллюстрирующих практическое применение новых технологий Flash MX: отрисовку фигур на этапе исполнения, преобразование массивов в экранные таблицы, создание компонентов многократного использования, работу с XML и звуком. Помимо стандартных методик описываются лучшие приемы программирования и скрытые возможности, особое внимание уделяется объектно-ориентированному программированию и новой модели событий Flash MX. Книгу оценят как новички, так и специалисты.

ISBN 5-93286-068-5
ISBN 0-596-00396-X (англ)

© Издательство Символ-Плюс, 2004
Authorized translation of the English edition © 2002 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 25.06.2004. Формат 70×100 1/16 . Печать офсетная.
Объем 70 печ. л. Тираж 3000 экз. Заказ № 782
Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29.

Оглавление

Предисловие . . . 12
Введение . . . 15

Часть I. Основы ActionScript . . . 35

1. Легкое введение для непрограммистов . . . 37
Некоторые основные фразы . . . 39
Другие понятия ActionScript . . . 48
Создание вопросника с вариантами ответов . . . 58

2. Переменные . . . 74
Создание переменных (объявление) . . . 74
Присваивание значений переменным . . . 78
Изменение и извлечение значений переменных . . . 80
Типы значений . . . 81
Область видимости переменной . . . 83
Загрузка внешних переменных . . . 97
Некоторые практические примеры . . . 98

3. Данные и типы данных . . . 100
Данные и информация . . . 100
Сохранение смысла данных с помощью типов . . . 100
Создание данных и задание их категорий . . . 102
Преобразование типов данных . . . 104
Элементарные и сложные типы данных . . . 111
Копирование, сравнение и передача данных . . . 112

4. Элементарные типы данных . . . 116
Числовой тип . . . 116
Целые числа и числа с плавающей точкой . . . 116
Числовые литералы . . . 116
Действия с числами . . . 121
Строковый тип . . . 122
Работа со строками . . . 127
Булев тип . . . 145
Тип undefined . . . 147
Тип null . . . 148

5. Операторы 151

Общие характеристики операторов 151
Оператор присваивания 155
Арифметические операторы 157
Операторы равенства и неравенства 161
Операторы строгого равенства и неравенства 166
Операторы сравнения 167
Строковые операторы Flash 4 171
Логические операторы 171
Оператор группирования 177
Оператор «запятая» 178
Оператор void 178
Прочие операторы 179

6. Инструкции 185

Типы инструкций 186
Синтаксис инструкций 186
Инструкции ActionScript 188
Сравнение инструкций с действиями 196

7. Условные инструкции 197

Инструкция if 198
Инструкция else 199
Инструкция else if 201
Инструкция switch 202
Компактный синтаксис условных инструкций 205

8. Инструкции цикла 206

Цикл while 206
Терминология циклов 209
Цикл do-while 211
Цикл for 212
Цикл for-in 213
Досрочное завершение цикла 215
Циклы временной диаграммы и событий клипа 217
Альтернатива циклам временной диаграммы: setInterval() 223

9. Функции 224

Создание функций 225
Запуск функций 225
Передача информации в функции 226
Выход из функций и возвращение значений 231
Литералы функций 234
Доступность и срок жизни функций 235

Область видимости функции . . . 237
Еще раз о параметрах функции . . . 242
Рекурсивные функции . . . 246
Вложенные функции . . . 248
Встроенные функции . . . 251
Функции как объекты . . . 252
Централизация кода . . . 256
Еще раз вопросник с вариантами ответов . . . 256

10. События и обработка событий . . . 263

Синхронное выполнение кода . . . 263
Асинхронное выполнение кода, управляемое событиями . . . 263
Типы событий . . . 264
Обработка событий . . . 265
Свойства-обработчики событий . . . 266
События приемников . . . 268
Обработчики событий on() и onClipEvent() во Flash 5 . . . 272
Срок жизни обработчиков событий . . . 275
Область видимости обработчиков событий . . . 275
Значения ключевого слова this . . . 281
Порядок выполнения onClipEvent() в стиле Flash 5 . . . 281
Копирование обработчиков событий клипа . . . 284
Обновление экрана с помощью updateAfterEvent() . . . 285
Повторное использование кода . . . 286
Динамические обработчики событий клипов . . . 286
Применение обработчиков событий . . . 287

11. Массивы . . . 290

Что такое массив? . . . 290
Анатомия массива . . . 291
Создание массивов . . . 292
Обращение к элементам массива . . . 295
Определение размера массива . . . 297
Именованные элементы массива . . . 298
Добавление элементов в массив . . . 300
Удаление элементов из массива . . . 305
Общие средства обработки массивов . . . 309
Массивы как объекты . . . 314
Многомерные массивы . . . 315
Вопросник с вариантами выбора: подход № 3 . . . 316

12. Объекты и классы . . . 319
Анатомия объекта . . . 322
Создание экземпляров объектов . . . 323
Свойства объектов . . . 324
Методы объекта . . . 326
Классы и объектно-ориентированное программирование . . . 327
Отдельные экземпляры Object как ассоциативные массивы . . . 346
Всемогущая цепочка прототипов . . . 348
Встроенные классы и объекты ActionScript . . . 359
Краткий справочник по ООП . . . 362
Дополнительные темы . . . 367
Моделирование пространств имен . . . 368
Вопросник с вариантами ответов в стиле ООП . . . 369

13. Клипы . . . 374
«Объектность» клипов . . . 375
Типы клипов . . . 376
Создание клипов . . . 379
Порядок расположения фильмов и экземпляров в стеке . . . 389
Ссылки на экземпляры и главные фильмы . . . 395
Удаление экземпляров клипов и главных фильмов . . . 408
Проблемы, связанные с перекрытием методов и глобальных функций . . . 411
Рисование в клипе на этапе выполнения . . . 413
Использование клипов в качестве кнопок . . . 414
Фокус ввода и клипы . . . 417
Создание часов с помощью клипов . . . 418

14. Подклассы класса MovieClip и компоненты . . . 423
Создание символа в библиотеке . . . 426
Создание и вызов конструктора подкласса . . . 426
Назначение MovieClip надклассом . . . 429
Объединение кода подкласса с библиотечным символом . . . 430
Создание компонентов . . . 431
Подклассы подклассов MovieClip . . . 437
Резюме . . . 439

15. Лексическая структура . . . 441
Пробельные символы . . . 441
Символы, завершающие инструкции (точка с запятой) . . . 442
Комментарии . . . 444
Зарезервированные слова . . . 446
Идентификаторы . . . 447
Чувствительность к регистру . . . 447

16. Среда разработки ActionScript 450
Панель Actions 450
Помещение сценариев в кадры 453
Добавление кода к кнопкам 454
Добавление сценариев в клипы 455
Куда подевался код? 456
Производительность 457
Сохранение кода ActionScript во внешних файлах 458
Создание компонентов 460

17. Создание формы Flash 469
Цикл данных форм Flash 469
Создание заполняемой формы Flash 472

Часть II. Справочник по языку 481

Справочник по языку ActionScript 483
Глобальные функции 484
Глобальные свойства 484
Встроенные классы и объекты 485
Заголовки статей 486
Алфавитный справочник по языку 488

Часть III. Приложения 1027

A. Ресурсы 1029
B. Набор символов Latin 1 и коды клавиш 1035
C. Обратная совместимость и обновление версий проигрывателя 1042
D. Отличия от ECMA-262 и JavaScript 1055
E. Поддержка HTML в текстовых полях 1059
F. Поддержка GET и POST 1069
G. Обзор компонентов Flash UI 1071
H. Встраивание фильмов Flash в веб-страницы 1081

Алфавитный указатель 1087

Отзывы на 2-е издание книги «ActionScript для Flash MX. Подробное руководство»

Подобно своему предшественнику, 2-е издание книги «ActionScript for Flash MX: The Definitive Guide» (ASDG2) – настольная книга тех, кто пишет на ActionScript. Из-под пера Мука вышел полностью переработанный справочник, не только подробно описывающий бесчисленные нововведения Flash MX, но и раскрывающий в ясной и доступной манере глубинные изменения, произошедшие на методологическом уровне (особенно касающиеся компонентов и событий).

Роберт Пеннер, автор «Robert Penner's Programming Macromedia Flash MX»

Мне нравится эта книга. Это доступное и в то же время весьма глубокое исследование ActionScript. Если вы работаете с Flash MX как дизайнер или программист, вы должны иметь эту книгу.

Хилман Кертис, основатель hillmancurtis, Inc. (http://www.hillmancurtis.com), автор «Flash Web Design» и «MTIV: Process, Inspiration and Practice for the New Media Designer»

Об этой книге невозможно говорить, не прибегая к превосходным степеням. Второе издание «ActionScript for Flash MX: The Definitive Guide» Мука – одна из самых необходимых книг по программированию в среде Flash, какую можно купить. В этом исчерпывающем и основополагающем справочнике одинаковое внимание уделяется стандартным методикам, лучшим приемам и скрытым возможностям. Прочитав его, любой, кто пишет на ActioScript, узнает об этом языке что-то новое.

Найджел Пег, разработчик компонентов Flash UI, Macromedia

Как разработчик, преподаватель и коллега по перу я не могу нахвалиться на ASDG2. Эта книга просто незаменима для всех, кто работает с Macromedia Flash. Колин прекрасно владеет ActionScript и умеет изложить свои знания понятным, кратким и зачастую остроумным способом.

Брэнден Дж. Холл, основатель и администратор Flashcoders List, автор «Object-oriented Programming with ActionScript»

Колин еще раз написал незаменимый справочник Flash-разработчика. Всесторонний охват ActionScript делает этот справочник самым полным информационным ресурсом из всех доступных. Я опять выбрал эту книгу.

Дейв Янг, главный исполнительный директор (CEO) и разработчик, Quantumwave Interactive Inc.

Первое издание этой книги содержало информацию, которую просто нельзя было найти где-либо в другом месте. То же самое и в еще большей мере справедливо в отношении второго издания, посвященного более сложной системе Flash MX. В очередной раз Колин Мук и O'Reilly создали отлично написанный и организованный, тщательно выверенный источник информации для программистов Flash MX всех уровней. Кем бы вы ни были – проектировщиком, программистом или художником, эта книга – ваш бесценный помощник в разработке приложений и в художественном самовыражении средствами Flash MX.

Амит Питару, художник и технолог (http://www.pitaru.com, http://www.insert-silence.com), преподаватель Pratt Institute & NYU (http://itp.nyu.edu), соавтор «New Masters of Flash: The 2002 Annual»

Эта книга – исчерпывающий ресурс по ActionScript, созданный человеком, который, собственно, и стоял у его истоков.

Тодд Пургасон, креативный директор и соучредитель Juxt Interactive (http://www.juxtinteractive.com), автор «Flash Deconstruction»

ASDG2 – бесспорно, основополагающая книга по программированию на ActionScript во Flash MX. Она оказалась неоценимой даже для инженеров Macromedia Flash, которые рассматривают ее как дополнение к нашей собственной промышленной документации. Дотошное внимание, которое Мук уделяет деталям, всюду просматривается в этой прекрасной книге и в сочетании с легко воспринимаемым стилем обучения гарантирует, что книгу оценят как новички, так и специалисты.

Гэри Гроссман, создатель ActionScript, Macromedia

ASDG2 – великолепный по лаконичности и непосредственности источник информации. Он отличается полнотой и доскональностью, не обходя, по сути, молчанием ни одного аспекта Flash. Это единственная книга по Flash, с которой я работаю, и не будь ее у меня, я не раз оказался бы в затруднительном положении. В умении Колина подать информацию или идею есть что-то такое, благодаря чему даже сугубо технические области Flash для меня совершенно прозрачны.

Джеймс Паттерсон, художник (http://www.presstube.com), соавтор «New Masters of Flash»

ASDG2 – это единственный источник, к которому следует обращаться, чтобы узнать обо всех возможностях, предоставляемых Flash MX. Даже Macromedia не вникает в подробности так глубоко, как это делает Колин Мук. Если вам надо освоить ActionScript за неделю или найти решение какой-то задачи за час, то ASDG2 – единственный справочник, который позволит вам это сделать. Самое замечательное, что Мук – изумительный и увлеченный писатель. Я не встречал другой книги по программированию, которую было бы столь же приятно читать.

Джош Улм, основатель ioResearch Studios (http://www.ioresearch.com) и The Remedi Project (http://www.theremediproject.com)

Книга Мука – из тех, которые постоянно находятся на рабочем столе. В любых случаях – когда надо найти решение в условиях поджимающих сроков выполнения проекта или углубить свои навыки программирования, эта книга оказывается бесценным руководством, показывающим все входы и выходы ActionScript.

Гленн Томас, директор и основатель Smashing Ideas, Inc. (http://www.smashingideas.com), автор «Flash Studio Secrets», соавтор «Flash Enabled»

ASDG2 – это действительно самая точная и хорошо организованная книга по программированию во Flash из всех напечатанных. Очевидно, что Колин Мук увлечен предметом, и этим обусловлена удача книги.

Джаред Тарбелл, специалист по вычислительным технологиям (http://www.levitated.net), соавтор «Flash Math Creativity» и «Fresh Flash»

На рынке немало книг по Flash (в том числе и написанных мной самим!), но я не устаю рекомендовать только одну из них – ASDG2 Колина Мука. Используемая самостоятельно или в сочетании с любой другой книгой по Flash, она представляет собой поистине Святой Грааль новых идей для тех, кто профессионально работает с Flash, и помогает нам не совершать синтаксических ошибок.

Джошуа Дэйвис, художник и технолог (http://www.praystation.com), автор «Flash to the Core»

Предисловие

Прошло всего восемнадцать месяцев с того момента, когда я написал предисловие к первому изданию «ActionScript: подробное руководство». За это время первое издание зарекомендовало себя как необходимое руководство по программированию на ActionScript. Оно стало необходимо столь многим разработчикам, что кажется, будто оно существует гораздо дольше.

Flash MX, выпущенный в марте 2002 года, стал самым амбициозным к настоящему моменту выпуском Flash. Команда талантливых личностей, участвовавших в его создании, была более многочисленна, чем когда-либо прежде, и мы добавили в продукт свыше 100 существенных новых функций. ActionScript стал средоточием главных усилий, что потребовало изменений в способе разработки. До Flash MX разработка ActionScript осуществлялась небольшой группой лиц, в которую входил и я. В MX наши грандиозные планы в отношении ActionScript потребовали большого количества разработчиков. Располагая дополнительными ресурсами, мы смогли значительно усовершенствовать редактор сценариев и отладчик, оптимизировать производительность и добавить массу новых API, предоставивших новые возможности программирующим на ActionScript.

Сегодня Macromedia проявляет к Flash повышенный интерес. Принято считать Flash просто средством создания анимации, но в сообществе Flash-разработчиков растет убеждение, что Flash представляет собой нечто большее. Благодаря Flash MX у веб-разработчиков появилась возможность предоставлять через Сеть насыщенную интерактивную среду для пользователя – не только традиционные для Flash мультфильмы и динамическую графику, но и сложные веб-приложения.

Flash всегда был и, видимо, останется лучшим средством для придания веб-сайту некой веселости, а разработчики серьезных веб-приложений стремятся преодолеть ограничения HTML. Они ищут новую платформу, которая предлагала бы пользователям более заманчивые, привлекательные и практичные ощущения – *обогащенного клиента*, – и обнаруживают, что Flash идеально подходит для доставки такого содержимого. Единообразная работа Flash на разных платформах и повсеместное распространение этого продукта предлагают технологию этапа исполнения, на которой разработчики могут создавать веб-приложения нового рода, более интересные и живые по сравнению с существовавшими до сих пор. Бьюсь об заклад, что мы увидим широкий спектр новых применений Flash, от многопользовательских игр до электронной торговли и визуализации данных. И в Macromedia стремятся обеспечить соответствие Flash новым требованиям, которые предъявляют к нему разработчики приложений. ActionScript играет важную роль в этом новом представлении о Flash MX. Поскольку полезность платформы Flash зависит

от мощи его языка сценариев, мы постарались сделать ActionScript достаточно мощным, чтобы удовлетворить самых требовательных веб-разработчиков.

Стремление сделать Flash подлинной платформой приложений заставило разработчиков Flash MX решать особые задачи. Flash можно считать продуктом, развитие которого стимулируется сразу в нескольких направлениях, поскольку он должен удовлетворять требованиям различных клиентов, от создателей анимированных персонажей до тех, кто занимается динамической графикой (motion graphics) и разработкой мощных приложений. Усовершенствование механизма сценариев рассматривалось как решающая задача, но мы сознавали, что столь же важно усилить средства Flash для творческого выражения, поскольку изобразительное искусство – душа и сердце Flash.

С целью гарантировать удовлетворение разнообразных потребностей наших клиентов, мы разделили всю команду разработчиков Flash на три группы, каждой из которых была назначена своя цель:

Доступность

Предоставить отличные начальные условия новичкам

Созидательность

Усилить средства Flash для творческого выражения

Мощь

Расширить мощь ActionScript как инструмента для разработки сложных приложений

К моему удовольствию, я возглавил последнюю группу, которая занялась усовершенствованием ActionScript для поддержки представления о Flash как о платформе. Мы пересмотрели и улучшили модели объектов и событий Flash; мы развили «умные» клипы Flash 5 в более надежную архитектуру компонентов и переписали наиболее употребительные объекты ActionScript с целью оптимизации эффективности их работы. Кроме того, мы добавили средства облегчения труда разработчиков, введя подсказчик кода (Code Hints) и переработав отладчик.

Однако над ActionScript трудились не мы одни. Объединение Macromedia с Allaire в 2001 году принесло в компанию солидный опыт работы с серверами. Сотрудники из нового офиса Macromedia в Ньютоне, штат Массачусетс, создали Macromedia Flash Remoting MX (Flash Remoting), новую серверную технологию, позволяющую осуществлять прямую и легкую связь с сервером. Бригада блестящих разработчиков Macromedia Flash Communication Server MX (Comm Server) показала, что можно сделать с ActionScript, введя новые API ActionScript (в том числе ServerSide ActionScript), которые обеспечивают великолепные возможности: живую двустороннюю связь и сотрудничество через Интернет!

Еще одна группа была выделена для создания компонентов. Эта группа, два человека из состава которой были научными редакторами данной книги, разработала компоненты UI, позволяющие быстро создавать HTML-подобные формы, и дополнительные элементы управления, которые выводят за

рамки того, что можно сделать в HTML, например полнофункциональный элемент дерева, календарь, сетку данных. В сочетании с Flash Remoting эти компоненты образуют мощные средства создания приложений, управляемых данными.

Компоненты Flash MX дают очень хорошее представление о том, что ожидает нас в будущем: абстракции высокого уровня, из которых можно быстро собирать интерактивный контент и приложения. Мы в Macromedia будем стремиться к тому, чтобы сделать создание и применение компонентов в будущих версиях Flash еще более простым и эффективным. Компоненты, поставляемые с Comm Server, дают отличный пример такой мощи. Даже без компонентов, используя Comm Server, довольно просто создать приложение для видеоконференций с помощью всего нескольких строк ActionScript. Компоненты Comm Server делают решение этой задачи еще более простым: с помощью обычного перетаскивания нескольких компонентов новичок может фактически создать сценарий, не пользуясь ActionScript. Мы заинтересованы в развитии этого направления, поскольку оно позволяет новичкам сразу же создавать полезные продукты. Будьте уверены, что по мере облегчения доступности ActionScript и Flash большие возможности станут открываться и перед опытными разработчиками. Взяв на себя черновую работу и разработку употребительных компонентов интерфейса пользователя, мы даем возможность опытным пользователям и программистам еще более повысить свою продуктивность. Усовершенствованная объектная модель и архитектура компонентов Flash MX позволяют квалифицированным разработчикам расширять существующие компоненты или разрабатывать собственные пользовательские библиотеки. Поэтому, хотя данная книга не описывает подробно имеющиеся компоненты, она предлагает продвинутым и честолюбивым разработчикам средства для создания собственных компонентов. Всегда радует, когда разработчики открывают новые направления применения ActionScript, получив инструменты и разобравшись с тем, как их использовать.

Таким образом, второе издание бесспорно является необходимой книгой для программирования на ActionScript во Flash MX. Она оказалась незаменимой даже для разработчиков команды Macromedia Flash, которые считают ее дополняющей нашу собственную документацию. Эта книга стала результатом безграничного таланта и энергии Колина Мука, которые заставили его глубоко вникнуть в ActionScript, раскрывая для читателя внутренние тайны этого языка. Дотошное внимание Колина к деталям, явно просматриваемое на протяжении всей этой замечательной книги, в сочетании с неторопливым педагогическим стилем, гарантируют, что книга будет оценена как новичками, так и специалистами. Получите удовольствие от книги и от ActionScript во Flash MX!

Гэри Гроссман,
создатель ActionScript,
Senior Engineering Manger,
Macromedia Flash Team,
октябрь 2002

Введение

Добро пожаловать во второе издание «ActionScript for Flash MX: The Definitive Guide»! По сравнению с первым оно сильно изменилось: появились сотни новых страниц, а старый материал значительно переработан, что привело его в соответствие с лучшей современной практикой работы во Flash MX. Надеюсь, вы с таким же удовольствием прочтете эту книгу, с каким я ее писал!

Как и первое издание, эта книга обучает читателя языку ActionScript с самых основ, рассказывая о фундаментальных понятиях и более сложных случаях применения языка, но особое внимание в ней уделяется технологии Macromedia Flash MX. В части I мы изучим основы ActionScript, начав с переменных и управления клипами, и дойдем до более сложных тем, таких как объекты, классы и связь с сервером. В части II «Справочник по языку» мы расскажем обо всех объектах, классах, свойствах, методах и обработчиках событий базового языка ActionScript. Сюда вы будете заглядывать регулярно, узнавая новое и вспоминая то, что часто забывается, поэтому книга должна лежать на вашем рабочем столе, а не на полке!

Во Flash MX язык ActionScript стал сложнее, но для чтения книги не обязательно быть программистом. В этом издании я по-прежнему старался проявлять заботу о новичках. Изложение достаточно насыщенно, но для освоения материала не нужна предварительная подготовка в программировании. Достаточно иметь опыт работы с Flash без использования ActionScript и желание учиться. Конечно, лучше иметь опыт программирования – тогда вы сразу сможете применить свои навыки к ActionScript. Чтобы облегчить переход на Flash опытным программистам, я намеренно приводил полезные аналогии с такими языками, как JavaScript, Java и C.

Кроме того, эта книга – действительно «подробное руководство» по ActionScript для Flash MX. Это результат почти четырех лет исследований, тысяч электронных писем сотрудникам Macromedia и откликов, полученных от пользователей всех уровней подготовки. Надеюсь, что книга явственно демонстрирует мой горячий интерес к предмету и приобретенный нелегким трудом практический опыт, которым вы можете непосредственно воспользоваться. Она представляет собой источник исчерпывающей и – благодаря рецензированию, выполненному Гэри Гроссманом (Gary Grossman), создателем ActionScript, – исключительно точной информации по ActionScript.

Краткий обзор отличий второго издания

Тем, кто знаком с первым изданием этой книги и кому не терпится узнать, что же содержится во втором, я могу указать на несколько отличительных

особенностей последнего. Однако не ограничивайтесь приводимым списком. Читайте книгу, и вы найдете много других важных изменений.

Следующие главы части I «Основы ActionScript» были существенно переработаны. В них рассказывается о некоторых наиболее примечательных нововведениях, таких как компоненты и важные изменения в способах обработки событий и действий с объектами.

- Глава 9 «Функции»
- Глава 10 «События и обработка событий»
- Глава 12 «Объекты и классы»
- Глава 14 «Подклассы класса MovieClip и компоненты»

Обратите также внимание на переработанные и новые приложения, в особенности следующие:

- Приложение C «Обратная совместимость и обновление версий проигрывателя»
- Приложение E «Поддержка HTML в текстовых полях»
- Приложение F «Поддержка GET и POST»
- Приложение G «Обзор компонентов Flash UI»
- Приложение H «Встраивание фильмов Flash в веб-страницы»

Нижеперечисленные статьи в части II «Справочник по языку» или переписаны заново, или весьма существенно переработаны по сравнению с первым изданием. Например, вам надо будет внимательно прочесть статью о *SharedObject* и изучить методы Drawing API, добавленные в класс *MovieClip*.

- *Accessibility*, объект
- *Button*, класс
- *Capabilities*, объект
- *Function*, класс
- `_global`, объект
- `#initclip` и `#endinitclip`, прагмы
- *LoadVars*, класс
- *LocalConnection*, класс
- *MovieClip*, класс (новые события и Drawing API)
- *Object*, класс
- *setInterval( )* и *clearInterval( )*, глобальные функции
- *SharedObject*, объект
- *Sound*, класс
- *Stage*, объект
- *System*, объект
- *TextField*, класс
- *TextFormat*, класс

- События приемника для *Key*, *Mouse*, *TextField* и *Stage* (см. табл. B.1)

Что нового в ActionScript для Flash MX

ActionScript чрезвычайно эволюционировал с переходом от Flash 5 к Flash MX (так называется новая среда разработки) и соответствующему Flash Player 6, и эта книга претерпела изменения вместе с ним. Подробные сведения о названиях версий Flash можно найти в табл. B.2.

Действие многих новых функций можно увидеть на сайте:
http://www.moock.org/webdesign/lectures/newInMX

В табл. B.1 представлен общий обзор основных нововведений в ActionScript. В ней указывается, где можно найти дополнительную информацию по каждой новой теме данной книги. Если не оговорено иное, перекрестные ссылки указывают на часть II «Справочник по языку».

Таблица B.1. Новые функции во Flash MX ActionScript

Функция	Подробное описание см. в
Drawing API: рисование фигур и заливок на этапе исполнения при помощи новых методов *MovieClip*	*MovieClip.beginFill( ), MovieClip.beginGradientFill( ), MovieClip.clear( ), MovieClip.curveTo( ), MovieClip.endFill( ), MovieClip.lineStyle( ), MovieClip.lineTo( ), MovieClip.moveTo( )*; «Рисование в клипе на этапе выполнения» в главе 13
Загрузка изображений в формате JPEG на этапе исполнения	*MovieClip.loadMovie( ), loadMovie( )*
Загрузка звука в формате MP3 на этапе исполнения	*Sound.loadSound( )*
Получение длительности звукового фрагмента и текущей продолжительности его исполнения	*Sound*.`position`, *Sound*.`duration`
Событие завершения исполнения звукового фрагмента	*Sound.onSoundComplete( )*
Создание, обработка и форматирование текстовых полей на этапе исполнения	Класс *TextField*, класс *TextFormat*, *MovieClip.createTextField( )*
Установка и снятие маски клипа на этапе исполнения	*MovieClip.setMask( )*
Создание клипов с нуля на этапе исполнения	*MovieClip.createEmptyMovieClip( )*
Определение глубины клипа на этапе исполнения	*MovieClip.getDepth( )*

Таблица В.1 (продолжение)

Функция	Подробное описание см. в
Периодическое выполнение функции или метода	*setInterval( )*, *clearInterval( )*
Ускорение обработки XML, строк и массивов данных благодаря повышению производительности	Класс *XML*, класс *String*, класс *Array*
Локальное хранение данных (аналогично cookies в JavaScript)	Объект *SharedObject*
Создание упакованных модулей кода с помощью подклассов MovieClip и компонентов	`#initclip`, `#endinitclip`, *Object.registerClass( )*, *attachMovie( )*; глава 14
Обмен данными между двумя Flash-проигрывателями на одном и том же компьютере	Класс *LocalConnection*
Объявление глобальных переменных	`_global`; «Переменные клипов и глобальные переменные» в главе 2
Использование международных символов в кодировке Unicode	«Строковый тип» в главе 4, приложение С
Определение обработчиков событий клипов с помощью функций обратного вызова	Глава 10
Приемники событий, генерируемых любыми объектами	Глава 10 и *Key.addListener( )*, *Mouse.AddListener( )*, *Stage.addListener( )*, *Selection.AddListener( )*, *TextField.addListener( )*
Задание режима кнопок для клипов	«Использование клипов в качестве кнопок» в главе 13
Объекты кнопок управления на этапе исполнения	Класс *Button*
Средства обеспечения доступности содержимого людям с нарушениями зрения	Объект *Accessibility*
Получение ширины и высоты фильма на этапе исполнения и перемещение элементов при изменении размеров фильма	*Stage.height*, *Stage.width*, *Stage.onResize( )*
Использование лексической или вложенной области видимости функции либо выполнение функции как метода произвольного объекта	*Function.call( )*, *Function.apply( )*; «Цепочка областей видимости» в главе 2; «Область видимости функций» в главе 9

Функция	Подробное описание см. в
Получение информации о проигрывателе и системе (о разрешении экрана, типе операционной системы и об установленном языке)	Объект *Capabilities*
Перехват событий от клавиатуры и мыши через централизованный API ввода	Объект *Key*, объект *Mouse*
Загрузка переменных с помощью класса LoadVars вместо функции *loadVariables( )*	Класс *LoadVars*
Контроль хода загрузки XML или переменных	*XML.getBytesLoaded( ), LoadVars.getBytesLoaded( )*
Управление порядком обхода кнопок, текстовых полей и клипов	*TextField*.tabIndex, *Button*.tabIndex, *MovieClip*.tabIndex
Отключение изменения курсора для кнопок	*Button*.useHandCursor, *MovieClip*.useHandCursor
Добавление свойств доступа к объекту и получение уведомления при изменении свойства	*Object.addProperty( ), Object.watch( )*

Что нового во втором издании

Второе издание «ActionScript for Flash MX: The Definitive Guide» – не просто дополнение к первому (называвшемуся «ActionScript: The Definitive Guide»). Текст полностью переработан, а изменение структуры руководства делает более заметными новые функции, появившиеся во Flash MX ActionScript. Практически каждый абзац подвергся редактированию, добавлено 400 страниц, посвященных новым возможностям ActionScript. Описания синтаксиса старого ActionScript для Flash 4 перемещены из основной части книги в приложение C и статьи в Интернете. Мы приняли такое решение, чтобы сделать книгу тоньше, но она все равно оказалась заметно более увесистой, чем первое издание. Когда эта книга дойдет до читателя, Flash Player 6 будет распространен почти повсеместно, поэтому больше нет смысла подробно рассматривать Flash 4, о котором здесь рассказано достаточно для того, чтобы помочь понять и модифицировать старый код, с которым вы можете столкнуться. Кроме того, мы тщательно рассмотрели различия между версиями Flash 5 и 6, что поможет вам разобраться в новых принципах и системах понятий и модифицировать имеющийся код. Примеры старого кода из первого издания по-прежнему доступны на *http://www.moock.org/asdg/codedepot*.

Обновленные примеры кода

Все примеры кода из первого издания были переработаны с учетом изменений синтаксиса ActionScript во Flash MX и лучших приемов работы. Например:

- В примерах с вопросником для обработчиков событий кнопок теперь используются функции обратного вызова вместо обработчиков *on( )* в стиле Flash 5.
- Текстовые поля, ранее создававшиеся в среде разработчика, теперь генерируются программно с помощью *createTextField( )*.
- Классы определяются в `_global` (новое свойство, в котором содержатся глобальные переменные).
- Вместо прежней глобальной функции *loadVariables( )* используется класс *LoadVars*.

Добавлены десятки новых примеров, специфических для Flash MX. Вот наиболее интересные из них:

- Полностью основанный на коде объектно-ориентированный вопросник; его можно загрузить из сетевого хранилища кода (Code Depot), о котором рассказывается ниже.
- Настраиваемая бегущая строка текста (text ticker) (см. *TextField*.`hscroll`).
- Преобразователь массивов в таблицы (см. *TextFormat*.`tabStops`).
- Загрузчик звука (см. *Sound.getBytesLoaded( )*).

Сотни усовершенствований

Содержимое первого издания улучшено массой небольших дополнений. Вот лишь некоторые из сотен улучшений:

- В описании свойства *MovieClip*.`_x` рассказано о *twips* (минимальном расстоянии, на которое можно переместить клип).
- В описание *MovieClip*.`_visible` добавлено предупреждение о том, что события кнопок не генерируются, когда `_visible` имеет значение `false`.
- В статье «Справочника про языку», посвященной методу *XML.parseXML( )*, подробно рассказано о секции CDATA и о предопределенных сущностях XML (&, <, >, " и ').
- В статье «Справочника по языку», посвященной методу *MovieClip.getBytesLoaded( )*, приведен список возможных значений, возвращаемых при асинхронном выполнении *loadMovie( )*.
- В главе 2 обсуждаются квалифицированные и неквалифицированные ссылки на переменные и венгерскую нотацию (*Hungarian notation*).
- В главе 4 подробно сравнивается `null` с `undefined` при описании операции *delete*.

Конечно, есть и более существенные изменения. Расскажем о них.

Главные изменения по сравнению с первым изданием

Ниже перечислены основные изменения в содержании и структуре второго издания. Обратите внимание, что некоторые из этих глав в первом издании располагались в части II «Применение ActionScript». Остальной материал его второй части теперь распределен по различным разделам второго издания, а часть сведений перемещена в файлы онлайн-справки. Несмотря на такие структурные изменения, не сомневайтесь, что во втором издании присутствуют десятки прикладных примеров, щедро разбросанных по всей книге. «Справочник по языку», бывший в первом издании частью III, стал теперь частью II.

Глава 1 «Легкое введение для непрограммистов»

- Добавлено введение в объектно-ориентированное программирование.
- Учебный пример вопросника переработан для Flash MX.
- Раздел по обработке событий переработан для Flash MX.

Глава 2 «Переменные»

- Добавлены рекомендации по суффиксам имен переменных.
- Добавлено описание глобальных переменных.
- Добавлен раздел по загрузке внешних переменных.
- Добавлено подробное обсуждение цепочки областей видимости.

Глава 3 «Данные и типы данных»

- Добавлен раздел «Копирование, сравнение и передача данных» (ранее находившийся в главе 15).

Глава 4 «Элементарные типы данных»

- Добавлено описание Unicode.

Глава 5 «Операции»

- Добавлено описание операторов строгого равенства и *instanceof*.

Глава 6 «Инструкции»

- Добавлено описание предложения *switch*.
- Пересмотрено описание *with*, в которое включены цепочки областей видимости.
- Удален старый оператор *call* (теперь о нем говорится только в «Справочнике по языку»).

Глава 8 «Инструкции цикла»

- Добавлен раздел о *setInterval( )*, позволяющей периодически выполнять код.
- Переработан раздел «Циклы временной диаграммы и событий клипа» с целью описания возможностей Flash MX (*MovieClip.createEmptyMovieClip( )* и обработчика *MovieClip.onEnterFrame( )*).

Глава 9 «Функции»

- Добавлен раздел о различиях между литералами функций и предложением *function*.
- Добавлено описание вложенных функций.
- Пересмотрен раздел «Область видимости функций», в который включено более подробное описание лексической области видимости.
- Учебный пример вопросника переработан для Flash MX.

Глава 10 «События и обработка событий»

- Добавлено полное описание свойств обработчиков событий.
- Добавлено описание приемников событий (event listeners), появившихся во Flash MX.
- Добавлено углубленное обсуждение областей видимости, включая табл. 10.1, сравнивающую старые и новые правила видимости.
- Добавлено описание ключевого слова `this` в различных обработчиках, включающее общую сводку в табл. 10.2.
- Все конкретные описания событий кнопок и клипов перемещены в «Справочник по языку» (см. также табл. 10.3).

Глава 11 «Массивы»

- Добавлено описание метода *Array.sortOn( )*.
- Учебный пример вопросника переработан для Flash MX.

Глава 12 «Объекты и классы»

- Глава полностью переработана и уделяет больше внимания процедуре создания класса с методами и свойствами.
- Добавлено описание ключевого слова Flash MX *super*, с помощью которого вызывается конструктор надкласса (superclass) и его методы.
- Добавлено формальное описание *цепочки прототипов*.
- Добавлено формальное описание вопросов, связанных с *организацией наследования классов*.
- Добавлен раздел по статическим методам и свойствам.
- Добавлено описание показа объекта на экране.
- Добавлен шаблон приложения объектно-ориентированного программирования (ООП).
- Добавлен раздел «Краткий справочник по ООП».
- Добавлено краткое описание UML и паттернов проектирования.

Глава 13 «Клипы»

- Добавлены сведения о создании чистого клипа с нуля с помощью *MovieClip.createEmptyMovieClip( )*.
- Добавлен раздел о рисовании в клипе на этапе исполнения с помощью нового Drawing API.

- Добавлен раздел о реализации в клипе поведения кнопки.
- Добавлен раздел о работе с фокусом ввода в клипах.
- Переработано (исправлено) содержащее ошибки описание *MovieClip.duplicateMovieClip( )*.
- Список методов и свойств *MovieClip* перемещен в «Справочник по языку».
- Описание устаревшего *tellTarget* перенесено в приложение С.
- Усовершенствован пример часов, в котором теперь использованы лучшие приемы работы с Flash MX.
- Пример вопросника заменен новой загружаемой ООП-версией (старую по-прежнему можно найти в Интернете).

Глава 14 «Подклассы класса MovieClip и компоненты» (полностью новая)

- Рассмотрено создание подклассов клипов (специальных типов клипов, связанных с классом).
- Рассказано, как создать элементарный компонент; сложным примером компонентов являются компоненты пользовательского интерфейса Flash.

Глава 15 «Лексическая структура» (глава 14 первого издания)

- Переработан список зарезервированных слов.
- Удалены или переработаны следующие темы главы 15:
  - «Копирование, сравнение и передача данных» перенесено в главу 3.
  - «Поразрядное программирование» перенесено в статьи на *http://www.moock.org/asdg/technotes*.
  - Удален раздел «Более сложные проблемы областей видимости функций» (проблема была решена во Flash MX).
  - Раздел «Тип данных MovieClip» перенесен в статьи на *http://www.moock. org/asdg/technotes*.

Глава 16 «Среда разработки ActionScript»

- Переработан раздел о старых «умных клипах» (smart clip), в котором теперь рассказывается о новой компонентной архитектуре Flash MX.

Глава 17 «Создание формы Flash»

- Переработан код примеров с целью использования класса *LoadVars* вместо *loadVariables( )*.

Расформирована глава 18 «Экранные текстовые поля» из первого издания:

- Содержимое всей главы перемещено в «Справочник по языку» (в описание класса *TextField*) и в приложение Е (с существенными дополнениями к классу *TextField*).

Удалена прежняя глава 19 «Отладка»

- Вся глава полностью перемещена в статьи на *http://www.moock.org/asdg/technotes*.

Часть II «Справочник по языку» (бывшая часть III)

- Ранее в этом предисловии мы отметили основные изменения и дополнения, касающиеся «Справочника по языку». Полный список новых методов, свойств, классов, объектов, глобальных функций и директив, добавленных в «Справочник по языку», можно найти на *http://www.moock.org/webdesign/lectures/newInMX*. (Обратите внимание, что статьи *CustomActions* и *LivePreview* не включены в «Справочник по языку», о чем говорится далее.)

Чего в этой книге нет

Книга велика, но ActionScript больше. Сейчас невозможно рассказать в одной книге обо всем, что есть в ActionScript. Из этого издания было специально изъято формальное описание следующих тем (о которых все же попутно рассказывается, когда это уместно):

- Функции, применяемые исключительно для расширения среды разработки Flash MX (например, *CustomActions* и *LivePreview*). О них рассказывается в статье Macromedia «Creating Components in Flash MX», расположенной по адресу *http://www.macromedia.com/support/flash/applications/creating_comps*.
- Библиотека Macromedia Flash UI Components, расширяющая возможности среды разработки за рамки базового языка. В приложении G «Сводка по компонентам интерфейса пользователя Flash» рассмотрены свойства и методы компонентов. Об источниках информации, подробно освещающих Flash UI Components, рассказано в «Резюме» главы 14.
- Macromedia Flash Communication Server MX (Comm Server) API (например, *Remote SharedObject*, *Camera*, *Microphone*, *NetConnection* и *NetStream*). Comm Server применяется для создания многопользовательских веб-приложений со звуком и видео. См. подробности в *http://www.macromedia.com/software/flashcom/*.
- Основы использования среды разработчика Flash MX. Однако если вы – программист и не знакомы с Flash, мы дадим ряд советов относительно ввода и выполнения примеров кода. Чтобы изучить анимацию и разработку графики в среде Flash MX, начните с электронной подсказки и учебника; затем посетите сайты, перечисленные по адресу *http://www.moock.org/moockmarks*.

К книге не прилагается CD, но все примеры могут быть загружены из хранилища кода в Интернете, о котором говорится ниже.

Недокументированные функции ActionScript

Сообщество разработчиков Flash наловчилось раскапывать так называемые *недокументированные функции* ActionScript – внутренние возможности языка, о которых Macromedia официально не сообщает и не санкционирует

их применение. Как правило, применять недокументированные функции не рекомендуется по следующим причинам:

- Они не тестировались для внешнего использования и потому могут содержать ошибки или работать неустойчиво.
- Они могут быть удалены из последующих версий языка без предупреждения.

В этой книге мы ставили задачу обеспечить возможно лучшую документацию по тем функциям, которые поддерживаются, но документированы недостаточно подробно или неправильно. Поэтому о функциях, документация или поддержка для которых отсутствует, не сообщается, за исключением следующих случаев:

- Источники в Macromedia передали или подтвердили имеющуюся информацию.
- Функция применяется так широко, что ее необходимо описать.

В любом случае описания недокументированных возможностей снабжены соответствующими предупреждениям на видном месте. Книга освещает следующие недокументированные темы:

- `__proto__` (применяется для установки наследования)
- *ASBroadcaster* (частично освещена в главе 12)
- *ASSetPropFlags( )* (частично освещена в главе 8)
- *LoadVars.decode( )*
- *LoadVars.onData( )*
- *Object.hasOwnProperty( )*
- *System.showSettings( )*
- *TextField*.`condenseWhite`
- Задание шрифта при помощи свойства *TextFormat*.`font`
- Класс *XMLNode*

Узнать (сохраняя при этом благоразумие), что обнаружили сыщики в ActionScript , можно на сайте:

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?Undocumented%20Features

Соглашения по именованию версий Flash

С появлением семейства продуктов MX, включающего Flash MX, Macromedia отказалась от обычной системы номеров версий для среды разработки Flash. Однако числовая нумерация версий Flash Player сохранена. В табл. B.2 приводятся соглашения по именам версий Flash, принятые в этой книге.

Таблица В.2. Соглашения по именам версий Flash, принятые в книге

Название	Смысл
Flash MX	Среда разработки Flash MX (в отличие от проигрывателя Flash Player).
Flash Player 6	Flash Player версии 6. Flash Player – модуль (plugin), подключаемый к большинству веб-броузеров в Windows и на Макинтошах. Существуют версии в виде ActiveX и в стиле Netscape, к которым применяется общее название «Flash Player 6», за исключением особо отмеченных случаев, как в статье, посвященной объекту *Accessibility*, в «Справочнике по языку».
Flash Player *x*.0.*y*.0	Flash Player конкретной версии, заданной *x* и *y*, как во Flash Player 6.0.47.0. См. подробности в «Справочнике по языку», в статье о capabilities.version.
Flash 6	Сокращение для «Flash Player 6», употребляемое преимущественно в «Справочнике по языку» или там, где различие между Flash MX (средой разработки) и Flash Player 6 (подключаемым модулем броузера) несущественно.
Flash 5, среда разработки	Среда разработки Flash 5 (в противоположность Flash Player), предшествовавшая Flash MX.
Flash Player 5	Flash Player версии 5.
Flash 5	Сокращение от «Flash Player 5», употребляемое в основном в «Справочнике по языку» или там, где различие между Flash 5 (средой разработки) и Flash Player 5 (подключаемым модулем броузера) несущественно.
Flash 2, Flash 3 и Flash 4	Версии Flash Player, предшествовавшие 5-й; употребляются преимущественно в «Справочнике по языку» и при этом указывается, какие версии Flash поддерживают данную функцию.
автономный проигрыватель	Версия Flash Player, работающая на машине непосредственно, а не как подключаемый модуль веб-броузера или элемент ActiveX.
Проектор (Projector)	Самодостаточный выполняемый модуль, содержащий в себе как файл *.swf*, так и автономный проигрыватель. Проекторы можно создавать для Mac OS или Windows с помощью функции Flash File → Publish.

Что умеет ActionScript?

ActionScript используется для создания любых типов интерактивных приложений, обычно для использования в Сети. Вот лишь некоторые возможности: проигрыватель MP3, многопользовательский графический редактор, трехмерное перемещение в здании, электронный магазин, доска объявлений, редактор HTML и игра Pacman. Каждое из этих приложений задействует некоторую группу возможностей ActionScript, выборочно рассмотренных ниже. Прикиньте, с помощью каких из этих приемов вы сможете создавать свои приложения.

Управление временной диаграммой

Фильмы Flash состоят из кадров, располагающихся в линейной последовательности, называемой *временной диаграммой* (*timeline*). ActionScript позволяет управлять воспроизведением временной диаграммы фильма, проигрывать отрезки фильма, выводить конкретный кадр, останавливать воспроизведение фильма, зацикливать анимации и синхронизировать анимационное содержимое. *Клипы* (*movie clips*) внутри главного фильма располагают собственными временными диаграммами.

Диалоговый режим

Фильмы Flash могут воспринимать вводимые пользователем данные и реагировать на них. С помощью ActionScript можно создавать следующие интерактивные элементы:

- Кнопки, реагирующие на щелчок мыши (например, классические кнопки навигации).
- Элементы GUI, такие как списки, выпадающие меню, флажки.
- Содержимое, анимируемое движениями мыши (например, шлейф мыши).
- Объекты, перемещаемые с помощью мыши или клавиатуры (например, автомобиль в симуляторе вождения).
- Текстовые поля, показывающие данные пользователю или позволяющие ему передавать данные в фильм (например, заполняемый бланк).

Управление визуальным и звуковым содержимым

ActionScript можно использовать для исследования или модификации свойств визуального и звукового содержимого фильма. Можно, например, изменять цвет и местоположение объекта, уменьшать громкость звука или устанавливать тип шрифта для текстового блока. Можно также многократно изменять эти свойства с течением времени, чтобы создавать такие эффекты, как движение, соответствующее законам физики, или обнаружение столкновений.

Программное создание содержимого

ActionScript позволяет генерировать визуальное и звуковое содержимое непосредственно из библиотеки фильма или посредством копирования содержимого, уже имеющегося в рабочем пространстве, или на сцене (stage). Во Flash MX можно посредством методов *createEmptyMovieClip( )* и *createTextField( )* из Drawing API класса *MovieClip* создавать графику и текст с нуля на этапе исполнения. Содержимое, генерируемое программно, может быть строго статическим элементом, например случайным узором или интерактивным элементом, таким как кнопка в диалоговом окне, вражеский космический корабль в видеоигре или пункт выпадающего меню.

Связь с сервером

Очень часто функциональность Flash расширяют путем организации связи с каким-либо приложением на стороне сервера, например Macromedia ColdFusion MX или сценарием Perl. Хотя связь с ColdFusion находится в сфере действия Macromedia Flash Remoting MX (Flash Remoting), базовый язык ActionScript предоставляет широкий набор инструментов для обмена данными с любыми приложениями или сценариями, выполняемыми на сервере (например, Java, PHP, ASP и т. д.). Обмен данными с сервером присутствует в следующих приложениях:

Ссылка на веб-страницу

См. *getURL( )*.

Гостевая книга

См. классы *LoadVars* и *XML*, главу 17 и хранилище кода, описываемое в следующем разделе.

Приложение чата

См. класс *XMLSocket* и пример на *http://www.moock.org/chat*.

Сетевая игра с нескольким участниками

См. класс *XMLSocket* и пример на *http://www.moock.org/unity*.

Операции торговли через Интернет

См. классы *LoadVars* и *XML*.

Персонализированный сайт с регистрацией и авторизацией пользователей

См. классы *LoadVars* и *XML*.

Подробное рассмотрение реализации даже такого ограниченного набора возможных приложений ActionScript выходит за рамки этой книги. Здесь мы постараемся обучить вас базовым навыкам, которые позволят вам самостоятельно исследовать бесчисленные возможности. Это не книга готовых рецептов, а уроки – как «приготовить» программный код с самого начала. Что будет в меню, решайте сами.

Хранилище кода

В последующих главах встретятся десятки примеров кода. Чтобы получить соответствующие файлы с исходным кодом, а также многие другие обучающие файлы, не включенные в книгу, посетите сетевое хранилище кода (Code Depot), находящееся здесь:

http://www.moock.org/asdg/codedepot

Хранилище кода – это развивающийся ресурс, содержащий практические приложения ActionScript и базовые коды. Вот список некоторых из них:

- Вопросник с вариантами ответов
- Средство просмотра графики с изменением масштаба

- Инструменты для текстовых полей, такие как конвертор массивов в таблицы и настраиваемая бегущая строка
- Приложение для чата, использующее XML
- Приложение гостевой книги
- Настраиваемые курсор мыши и кнопка
- Базовый код игры с астероидами
- Программные эффекты движения
- Демонстрация текстовых полей HTML
- Предзагрузчики
- Обработка строк
- Элементы интерфейса, такие как ползунки и средства прокрутки текста
- След мыши и другие зрительные эффекты
- Управление громкостью и звучанием

Кроме того, на указанный URL будут помещаться новые сведения о книге, уточнения, технические замечания и информация о выявленных ошибках.

Витрина

Практически в каждом существующем сайте, использующем Flash, есть хоть капелька ActionScript, а на некоторых сайтах его достаточно много. В табл. В.3 представлен ряд адресов, на которых вы можете получить вдохновение для собственной работы. Посмотрите также сайты, перечисленные в приложении А, и закладки, собранные автором на *http://www.moock.org/moockmarks*.

Таблица В.3. ActionScript Showcase

Тема	URL
Эксперименты в дизайне, интерактивности и сценариях	*http://www.yugop.com* *http://www.praystation.com** *http://www.presstube.com* *http://www.pitaru.com* *http://www.flight404.com* *http://www.bzort-12.com* *http://www.benchun.net/mx3d/** *http://www.protocol7.com** *http://www.uncontrol.com** *http://flash.onego.ru** *http://www.figleaf.com/development/flash5** *http://nuthing.com* *http://www.deconcept.com* *http://www.natzke.com*

* Можно загрузить файлы *.fla*. В других случаях доступны только файлы *.swf*.

Таблица В.3 (продолжение)

Тема	URL
Игры	*http://www.orisinal.com* *http://www.gigablast.com* *http://www.sadisticboxing.com* *http://www.huihui.de* *http://www.sarbakan.com* *http://www.electrotank.com/games/multiuser* *http://www.titoonic.dk/products/games/spider* *http://content.uselab.com/acno* *http://www.neave.com/webgames*
Интерфейс, приложения и динамическое содержимое	*http://www.mnh.si.edu/africanvoices* *http://www.curiousmedia.com* *http://www.smallblueprinter.com* *http://davinci.figleaf.com/davinci* *http://host.oddcast.com* *http://www.enteryourinformation.com/broadmoor/onescreen.cfm*

Типографские обозначения

В этой книге приняты следующие обозначения, призванные выделить различные синтаксические составляющие ActionScript:

Пункты меню

Выбор пункта меню обозначается символом → , например File → Open.

`Моноширинный шрифт`

Используется в примерах кода, именах экземпляров клипов, метках кадров, именах свойств и именах переменных. Имена переменных часто оканчиваются суффиксами, перечисленными в табл. 2.1 (например, `_mc` для переменных, ссылающихся на экземпляры клипов). Добавление суффиксов считается хорошим правилом, но иногда мы избегали их, если оказывалось, что они существенно затрудняют чтение окружающего текста. Поэтому для краткости суффиксы могут опускаться.

Курсив

Применяется в названиях функций, методов, классов, слоев, в URL, именах файлов и суффиксах файлов, например *.swf*. Кроме того, выделению функций и методов служат круглые скобки, например *duplicateMovieClip( )*.

`Моноширинный полужирный шрифт`

Указывает на текст, который надо ввести дословно при пошаговом выполнении процедуры. Его также используют внутри примеров кода, например для того, чтобы выделить важную строку в большом примере.

Моноширинный курсив

Этим шрифтом форматируется программный код, который надо заменить соответствующим значением (например, *здесь должно быть ваше имя*). Кроме того, он используется для выделения имен переменных, свойств, методов и функций, упоминаемых в комментариях в примерах кода.

Поэкспериментировав с разными шрифтами, мы пришли к выводу, что для «Справочника по языку» лучше всего подойдут следующие соглашения, не противоречащие нашему общему подходу:

- Свойства классов выделяются моноширинным шрифтом для имени класса и свойства, поскольку и то и другое должны быть введены буквально (например, `Stage.width`, `Math.NaN`).
- Свойства экземпляров выделяются моноширинным курсивом для класса или экземпляра объекта, поскольку их должно заменить имя конкретного экземпляра. Само свойство выделяется моноширинным шрифтом и должно быть введено точно так, как показано (например, *`Button`*`.tabEnabled`, где *`Button`* следует заменить экземпляром кнопки).
- Имена методов и функций, а также класс или объект, к которому они относятся, всегда выделяются курсивом с добавлением круглых скобок, как в *MovieClip.duplicateMovieClip( )*. Обратитесь к «Справочнику по языку», сопровождающему материалу или близлежащим примерам кода, чтобы выяснить, следует ли указать имя класса буквально, как в *TextField.getFontList( )*, или заменить его именем экземпляра, как в *ball_mc.duplicateMovieClip( )*.
- В «Справочнике по языку» при обсуждении свойства или метода класса для краткости имя класса часто опускается. Например, если при обсуждении свойства `htmlText` класса *TextField* мы говорим «установить свойство `htmlText`», по контексту должно быть понятно, что имеется в виду «установить свойство *`someField_txt`*`.htmlText`, где *`someField_txt`* – идентификатор конкретного текстового поля».
- В некоторых случаях свойство объекта содержит ссылку на метод или обработчик обратного вызова. При этом не всегда ясно, следовало ли с помощью моноширинного шрифта указать, что это свойство (хотя и хранящее имя метода), или применить курсив и скобки, указывая, что это метод (хотя и хранящийся в свойстве). Если граница между свойством, указывающим на метод, и самим методом оказывается расплывчатой, приносим свои извинения. Постоянно подчеркивая формальную разницу, мы сделали бы текст менее удобным для чтения.
- В сводке свойств класса их имена для экономии места могут выделяться *курсивом*, а не `моноширинным` шрифтом. Это происходит только при перечислении их под заголовком «Свойства» без круглых скобок, поэтому ясно, что речь идет о свойствах, а не о методах.

Что-то из сказанного может показаться сейчас неясным, но к тому моменту, когда вы доберетесь до «Справочника по языку», прочтя предварительно об объектах, классах и клипах в главах 12, 13 и 14, неясностей у вас не останется.

Обратите особое внимание на замечания и предупреждения, отделенные от текста следующими значками:

Это совет. В нем содержится полезная информация по рассматриваемой теме, часто освещающая важные понятия или лучшие приемы.

Это предостережение, которое поможет вам решить или избежать неприятных проблем или предупредит о неизбежном событии. За последствия игнорирования предупреждения вы будете отвечать сами.

Сообщите нам свое мнение

Информация, приведенная в данной книге, была проверена со всей возможной тщательностью, но некоторые функции могли измениться (не исключены даже ошибки!). Ждем ваших писем с сообщениями о найденных ошибках и предложениями для будущих изданий по адресу:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (в США и Канаде)
(707) 829-0515 (международные/местные)
(707) 829-0104 (факс)

У этой книги есть веб-страница, содержащая сведения об уже найденных ошибках, примеры и различную дополнительную информацию:

http://www.oreilly.com/catalog/actscript2

С комментариями и техническими вопросами по данной книге обращайтесь по адресу:

bookquestions@oreilly.com

Дополнительные сведения о наших книгах, конференциях, программном обеспечении, центрах ресурсов и сети O'Reilly Network можно найти на нашем веб-сайте:

http://www.oreilly.com

Благодарности

Как и первое издание, эта книга имела бы бледный вид без огромного вклада сотрудников Macromedia – групп разработки, контроля качества, сопровождения и управления продуктом Flash MX. В частности, огромная благодарность Гэри Гроссману (Gary Grossman) за критику, руководство и терпение, а также за написанное им предисловие. Вот имена сотрудников Macromedia, участвовавших в формировании этого текста: Джонатан Гэй (Jonathan Gay), Дже-

реми Кларк (Jeremy Clark), Эрик Виттман (Eric Wittman), Майкл Уильямс (Michael Williams), Пит Сантанджели (Pete Santangeli), Мэтт Вобенсмит (Matt Wobensmith), Бен Чан (Ben Chun), Трой Эванс (Troy Evans), Ли Томасон (Lee Thomason), Бентли Вулф (Bentley Wolfe), Джон Дауделл (John Dowdell), Ребекка Сан (Rebecca Sun), Дженис Пирс (Janice Pearce), Брайан Дистер (Brian Dister), Генриэтта Коэн (Henriette Cohn), Джефф Мотт (Jeff Mott), Майкл Моррис (Michael Morris), Денеб Мекета (Deneb Meketa), Тайник Уро (Tinic Uro), Роберт Тацуми (Robert Tatsumi), Колм Мак-Кеон (Colm McKeon) и Майк Чамберс (Mike Chambers).

Эту книгу редактировал Брюс Эпштейн (Bruce Epstein), у которого, как мне кажется, сверхчеловеческие способности. Он обладает исключительными познаниями в писательском деле и программировании, а его способность применять эти познания к тексту поразительна. Мне необычайно повезло, что у меня был такой выдающийся редактор (и самостоятельный автор).

Далее имею честь представить технических редакторов данного издания, которые входят в команду разработчиков Macromedia Flash MX: Гэри Гроссман, Крис Тилген (Chris Thilgen), Жиль Дрю (Gilles Drieu), Нигель Пегг (Nigel Pegg), Славик Лозбен (Slavik Lozben) и Майкл Ричардс (Michael Richards). Эрика Нортон (Erica Norton) редактировала первое издание. Спасибо, друзья, за потраченное время и самоотверженность.

Читателями предварительного текста этого издания были известные разработчики Flash, пользующиеся моим огромным уважением: Роберт Пеннер (Robert Penner) (*http://www.robertpenner.com*), Дейв Янг (Dave Yang) (*http://www.quantumwave.com*), Бренден Холл (Branden Hall) (*http://www.waxpraxis.org*), Амит Питару (Amit Pitaru) (*http:/ /www.pitaru.com*), Майкл Кей (Michael Kay) (*http://www.peep.org/wizard/*) и Вероника Броссье (Veronique Brossier) (*http://www.v-ro.com*). Точность этой книги обеспечена остротой их взгляда.

Спасибо Тиму О'Рейли (Tim O'Reilly), который установил высокий стандарт основательности, качества и точности во всем, что им публикуется. Моя благодарность Брайену Сойеру (Brian Sawyer), Клэр Клотье (Claire Cloutier), Глену Бисигнани (Glenn Bisignani), Майку Сьерре (Mike Sierra), Робу Романо (Rob Romano), Эди Фридман (Edie Freedman), Сенди Торр (Sandy Torre) и многим другим редакторам, составителям указателей, корректорам, сотрудникам служб продажи и маркетинга в O'Reilly, усилия которых способствовали появлению этой книги на полках.

Выражаю признательность своему доброму другу Дереку Клейтону (Derek Clayton), который систематически делился со мной своим опытом программирования. Дерек написал код Perl для главы 17, сервер Java *XMLSocket* для «Справочника по языку» и общую систему базы данных на плоских (flat) файлах, которые можно получить из сетевого хранилища данных. Он также является ведущим разработчиком Unity Socket Server, коммерческого приложения moock.org для создания во Flash многопользовательских приложений (*http://www.moock.org/unity*).

Сообществу Flash: спасибо за вдохновение и красоту, которые вы создаете. В частности, благодарность Джеймсу Паттерсону (James Patterson), Юго Накамуре (Yugo Nakamura), Наоки Мицусе (Naoki Mitsuse), Джошуа Девису (Joshua Davis), Джеймсу Бейкеру (James Baker), Марселю Марсу (Marcell Mars), Филиппу Торроне (Phillip Torrone), Роберту Рейнгардту (Robert Reinhardt), Марку Феннелу (Mark Fennell), Джошу Улму (Josh Ulm), Даррелу Планту (Darrel Plant), Тодду Пургасону (Todd Purgason), Джону Накку (John Nack), Джейсону Крогу (Jason Krogh), Хилману Куртису (Hillman Curtis), Глену Томасу (Glenn Thomas), Хосс Гиффорд (Hoss Gifford), Мануэлю Клементу (Manuel Clement), Андреасу Хейму (Andreas Heim), Роберту Ходжину (Robert Hodgin), Маргарет Карлсон (Margaret Carlson), Эрику Нацке (Erik Natzke), Андреасу Одендаалу (Andries Odendaal), Джеймсу Тиндаллу (James Tindall), Джону Вильямсу (Jon Williams), Ферри Халим (Ferry Halim), Джобу Макару (Jobe Makar), Джаред Тарбелл (Jared Tarbell), Джеффу Стирнсу (Geoff Stearns), Полю Сципула (Paul Szypula), Линде Вейнман (Lynda Weinman), перечисленным выше читателям бета-версии и всем тем, кого я по каким-то причинам пропустил.

Большая благодарность и признание в любви моей жене Венди Шаффер (Wendy Schaffer), родителям, семье и друзьям. Надеюсь, что это издание не было для них таким обременительным, как первое.

И наконец, хочу поблагодарить вас, читатель, за то, что вы нашли время для чтения этой книги. Надеюсь, она сможет передать вам мою страсть.

Колин Мук
Торонто, Канада
декабрь 2002

I

Основы ActionScript

В части I освещаются основы синтаксиса и грамматики языка ActionScript: переменные, данные, приложения, функции, обработчики событий, массивы, объекты, клипы и компоненты. Изучив ее, читатель будет знать все, что требуется для написания программ ActionScript.

- Глава 1 «Легкое введение для непрограммистов»
- Глава 2 «Переменные»
- Глава 3 «Данные и типы данных»
- Глава 4 «Элементарные типы данных»
- Глава 5 «Операторы»
- Глава 6 «Инструкции»
- Глава 7 «Условные инструкции»
- Глава 8 «Инструкции цикла»
- Глава 9 «Функции»
- Глава 10 «События и обработка событий»
- Глава 11 «Массивы»
- Глава 12 «Объекты и классы»
- Глава 13 «Клипы»
- Глава 14 «Подклассы класса MovieClip и компоненты»
- Глава 15 «Лексическая структура»
- Глава 16 «Среда разработки ActionScript»
- Глава 17 «Создание формы Flash»

1

Легкое введение для непрограммистов

Я буду учить вас разговаривать с Flash.

Не просто программировать на Flash, но говорить ему что-то и понимать то, что он скажет в ответ. Это не метафора или простой риторический прием. Это философский подход к программированию.

Языки программирования предназначены для отправки информации компьютерам и получения информации от них. Язык программирования, как и естественный язык, представляет собой совокупность словаря и грамматики, используемую для общения. При помощи языка программирования мы указываем компьютеру, какие действия он должен совершить, или запрашиваем у него информацию. Компьютер слушает, пытается выполнить запрашиваемые действия и выдает ответы. Поэтому, хотя вы можете думать, что читаете эту книгу для того, чтобы «научиться программировать», на самом деле вы учитесь общаться с Flash. Конечно, Flash не разговаривает ни по-английски, ни по-французски, ни по-немецки, ни по-китайски. Родным языком для Flash является ActionScript, и вы будете учиться разговаривать на нем.

Обучение компьютерному языку иногда рассматривается как синоним обучения программированию. Но программирование – это больше, чем простое изучение синтаксиса языка. Что если бы Flash мог разговаривать по-английски, то есть для общения с ним не требовалось бы изучать ActionScript?

Что произошло бы, если бы мы сказали: «Flash, сделай так, чтобы шарик прыгал по экрану»?

Flash не смог бы выполнить наш запрос, поскольку он не понимает слово «шарик». Ладно, это просто семантическая проблема. Flash ждет, чтобы ему описывали объекты того мира, который ему известен: клипы, кнопки, кадры и т. д. Поэтому перефразируем наш запрос в терминах, понятных Flash, и посмотрим, что произойдет: «Flash, сделай так, чтобы клип с именем `ball_one` прыгал по экрану».

Flash по-прежнему не сможет выполнить нашу просьбу без дополнительной информации. Какого размера должен быть шарик? Куда его надо поместить? В каком направлении он должен начать перемещаться? С какой скоростью

он должен двигаться? В какой части экрана он должен прыгать? Как долго? В двух измерениях или в трех? Да... такого количества вопросов мы не ожидали. В действительности Flash не задает нам эти вопросы. Если Flash не может нас понять, то просто не делает то, чего мы от него хотим, или выдает сообщение об ошибке. Но сейчас представим себе, что Flash запросил у нас более явные инструкции, и переформулируем свой запрос как последовательность шагов:

1. Шарик – это клип, содержащий круг, с именем `ball`.
2. Квадрат – это клип, содержащий четырехугольник, с именем `square`.
3. Создать новый зеленый шарик диаметром 50 пикселов.
4. Дать новому шарику имя `ball_one`.
5. Создать новый черный квадрат со стороной 300 пикселов и поместить его в центр рабочей области.
6. Поместить `ball_one` где-то поверх квадрата.
7. Перемещать `ball_one` в случайном направлении со скоростью 75 пикселов в секунду.
8. Если `ball_one` ударится об одну из сторон квадрата, заставить его отскочить (в обратном направлении).
9. Продолжать, пока не поступит команда остановиться.

Хотя мы давали команды на английском языке, нам все же пришлось пройти всю логику управления прыгающим мячиком, чтобы Flash смог нас понять. Очевидно, программирование не ограничивается лишь синтаксисом языков программирования. Аналогично знание большого количества слов английского языка не обязательно означает, что вы хороший собеседник.

Наш гипотетический пример Flash, разговаривающего на английском языке, выявляет четыре важных аспекта программирования:

- Каким бы ни был язык, искусство программирования заключается в формулировании логических шагов.
- Прежде чем сказать что-либо на языке компьютера, обычно полезно сказать это на английском языке.
- Разговор, ведущийся на одном языке, будучи переведенным на другой, по-прежнему состоит из тех же основных предложений.
- Компьютеры не очень догадливы. Кроме того, у них довольно ограниченный словарь.

По большей части программирование никак не связано с написанием кода. Прежде чем написать хоть одну строчку ActionScript, тщательно продумайте, что вы хотите сделать, и спланируйте функции своей системы в виде блок-схемы или проекта. Достаточно хорошо описав свою программу на концептуальном уровне, можете перевести ее на ActionScript.

В программировании, так же, как в любви, политике и бизнесе, ключом к успеху является эффективное общение. Чтобы Flash мог понять ваш ActionScript, синтаксис должен быть абсолютно правильным, вплоть до последней

кавычки, знака равенства и точки с запятой. И чтобы гарантировать, что Flash поймет, о чем вы говорите, надо ссылаться только на тот мир, который ему известен, и употреблять понятные ему термины. То, что может быть очевидным для вас, не всегда очевидно компьютеру. Программирование компьютера можно представить себе, как разговор с ребенком: ничто не следует считать само собой разумеющимся, необходимо быть точным в каждой детали и перечислять все шаги, необходимые для выполнения задачи. Но помните, что, в отличие от детей, Flash сделает точно то, что вы ему скажете, и ничего сверх этого.

Некоторые основные фразы

В первый же день занятий любым иностранным языком вы, скорее всего, выучите несколько основных фраз («Добрый день», «Как поживаете» и т. д.). Даже если вы просто выучили фразу и не знаете значения каждого слова, можно понять действие фразы и повторить ее, чтобы воспроизвести это действие. Когда вы изучите правила грамматики, расширите свой словарь и станете употреблять слова из заученных фраз в различном контексте, то сможете лучше понять свои первые фразы. Оставшаяся часть этой главы будет во многом напоминать первый день на курсах иностранного языка: вы увидите различные фрагменты кода и познакомитесь с некоторыми основами грамматики программирования. Остальная часть книги будет построена на этом фундаменте. Возможно, закончив чтение книги, вы захотите вернуться к этой главе, чтобы посмотреть, как далеко вы продвинулись.

Создание кода

В качестве первого упражнения мы попробуем добавить к фильму Flash четыре простые строки кода. Почти все программирование на ActionScript происходит в панели Actions (процедуры, действия). Все инструкции, которые мы добавляем к панели Actions выполняются Flash при воспроизведении фильма. Откройте теперь панель Actions, выполнив шаги:

1. Запустите Flash с новым пустым документом.
2. На главной временной диаграмме выберите кадр 1 слоя 1.
3. Выберите Window → Actions (F9). (Обратите внимание на использование символа «→» для отделения названия меню от пункта меню: вы должны выбрать пункт «Actions» в меню «Windows».)

Панель Actions разделена на две части: окно Script (справа) и окно Toolbox (слева). В окне Script (сценарии) размещается весь наш код. В режиме Normal Mode в верхней части окна Script размещается окно Parameters, которое упрощает ввод кода новичкам. Окно Toolbox (инструменты) предоставляет быстрый доступ к элементам языка ActionScript. Вероятно, вы узнаете показанные на рис. 1.1 действия по управлению фильмом исходя из прежних версий Flash.

Но в окне Toolbox можно обнаружить и многое другое. В этой книге подробно рассказывается об операторах, функциях, константах, свойствах и объектах

Рис. 1.1. *Действия по управлению фильмом Flash MX*

(Operators, Functions, Constants, Properties, Objects). Некоторые компоненты для коммуникаций относятся непосредственно к Macromedia Flash Communication Server MX (Comm Server) и выходят за рамки этой книги. Аналогично компоненты интерфейса пользователя Flash не входят в базовый язык ActionScript. Их свойства и методы приведены в приложении G. Описание элементов интерфейса пользователя Flash можно найти в документации Flash в разделе Help → Tutorials → Introduction to Components.

С помощью иерархического меню окна Toolbox, имеющего вид книги, можно создавать код ActionScript. Однако чтобы изучить синтаксис, принципы и структурный состав ActionScript, мы будем вводить весь код с клавиатуры.

Так называемые *Actions* являются не просто действиями, они содержат различные базовые средства языка программирования – переменные, условные инструкции, циклы, комментарии, вызовы функций и т. д. Они все находятся в одном меню, а общее название *Action* скрывает индивидуальное назначение программных структур. Мы будем раскрывать смысл Actions, что позволит взглянуть на них с точки зрения программиста. Везде в этой книге я употребляю для описания рассматриваемого действия подходящий термин программирования. Например, вместо «добавить действие *while*» я буду писать «создать цикл *while*». Вместо слов «добавить действие *if*» я буду писать «создать новую условную инструкцию». Вместо «добавить действие *play*» я буду писать «вызвать функцию (или метод) *play( )*». Эти различия составляют важную часть обучения языку ActionScript.

Готовы поработать руками? Тогда поздороваемся с Flash.

Поздороваемся с Flash

Прежде чем вводить код в панели Actions, надо отключить автопилот ActionScript. Для этого во всплывающем меню в правом верхнем углу окна Actions следует выбрать режим Expert Mode, как показано на рис. 1.2. Как вам это нравится? Вы уже эксперт.

Рис. 1.2. *Выбор режима Expert Mode*

Для краткости в оставшейся части книги мы станем называть любое всплывающее меню в одном из окон Flash – меню «Options». Поэтому можно также написать «выберите в окне Actions пункт Options → Expert Mode». Также заметим, что можно включить нумерацию строк с помощью Options → View Line Numbers. Подробности об окне Actions представлены в главе 16.

При входе в режим Expert Mode окно Parameters вверху панели Actions исчезает. (Во Flash 5 окно Parameters появлялось внизу окна Actions в режиме Normal Mode.) Пусть вас это не тревожит: мы не программируем с помощью меню, и поэтому оно нам не понадобится.

Теперь выберите кадр 1 в слое 1 на главной временной диаграмме.

Ваш ActionScript (называемый также *кодом*) всегда должен быть прикреплен к кадру, клипу или кнопке; выбор кадра 1 влечет за собой прикрепление к нему создаваемого вслед за этим кода. Когда временная диаграмма доходит до этого кадра, Flash исполняет прикрепленный к нему код.

В режиме Expert Mode можно вводить код с клавиатуры прямо в окно Script в правой части панели Actions, где мы и будем осуществлять все программирование.

А теперь волнующий момент – ваша первая строка кода. Настало время познакомиться с Flash! Введите в окно Script следующее:

```
var message = "Hi there, Flash!";
```

Эта строка кода образует законченную команду, называемую *инструкцией* (*statement*).[1] За ней введите вторую и третью строки кода, показанные после данного абзаца. Замените *здесь ваше имя* на свое имя (везде в книге, где вы увидите *код, набранный моноширинным курсивом*, замените эту часть кода собственным содержимым):

```
var firstName = "здесь ваше имя";
trace (message);
```

Хм. Пока ничего не произошло. Дело в том, что наш код ничего не выполнит, пока мы не экспортируем файл *.swf* и не воспроизведем наш фильм. Прежде чем сделать это, попросим Flash ответить на наше приветствие. Введите под уже имеющимися строками четвертую строку кода (ну, дело пошло...):

```
trace ("Hi there, " + firstName + ", nice to meet you.");
```

Теперь Flash готов вас встретить. Выберите Control → Test Movie и посмотрите, что произойдет. Некий текст должен появиться в окне Output, как показано на рис. 1.3.

Здорово, правда? Давайте разберемся, как все это произошло. Чтобы вернуться к редактированию файла исходного кода *.fla*, закройте окно файла *.swf*, созданное нашим тестом.

Рис. 1.3. *Flash становится дружелюбным*

[1] При описании действий программы часто используются слова «команда» и «инструкция». В таком контексте они имеют примерно одинаковый смысл, хотя, строго говоря, обозначают разные понятия. Команда может включать в себя несколько инструкций. – *Примеч. науч. ред.*

Отслеживание событий (переменные)

Помните, что я назвал программирование просто общением с компьютером? Так оно и есть, но, пожалуй, оно менее личное, чем я это до сих пор представлял. В первой строке кода

```
var message = "Hi there, Flash!";
```

вы в действительности не поздоровались с Flash. Скорее, вы сказали что-то вроде следующего:

> Flash, запомни, пожалуйста, кое-какую информацию для меня, а именно фразу «Hi there, Flash!». Впоследствии мне может потребоваться эта информация, поэтому дай ей метку `message`. Если когда-нибудь я попрошу у тебя `message`, возврати мне текст «Hi there, Flash!»

Возможно, это не так дружелюбно, как сказать «Привет!», но при этом иллюстрируется одна из подлинных основ программирования: Flash может запомнить для вас нечто при условии, что оно будет помечено так, чтобы потом его можно было найти. Например, во второй строке кода мы хотели, чтобы Flash запомнил ваше имя, и пометили ссылку на него как `firstName`. Flash запомнил ваше имя и вывел его в окне Output с помощью команды *trace( )*, когда вы тестировали свой фильм.

Способность Flash что-то запоминать для нас является решающим обстоятельством в программировании. Flash может запоминать данные любого типа, в том числе текст (например, ваше имя), числа (например, 3,14159) и более сложные типы данных, о которых мы расскажем позднее.

Официальная номенклатура переменных

Пора дать несколько формальных терминов для описания того, как Flash осуществляет запоминание. Вы уже знаете, что Flash запоминает данные. Отдельный фрагмент данных называется *элементом данных* (*datum*). Элемент данных (например, «Hi there, Flash!») и идентифицирующая его метка (например, `message`) вместе называются *переменной* (*variable*). Метка переменной называется ее *именем* (*name*), а элемент данных переменной называется ее *значением* (*value*). Мы говорим, что переменная *хранит* (*stores*) или *содержит* (*contains*) свое значение. Обратите внимание, что «Hi there, Flash!» заключено в двойные кавычки для обозначения того, что это *строка* текста, а не число или другой вид информации (*тип данных*).

В первой строке кода вы задали значение переменной `message`. Действие, связанное с приданием переменной значения, называется *присваиванием переменной значения*, или просто *присваиванием*. Но прежде чем присвоить переменной значение, вы должны создать ее. (JavaScript, близкий родственник ActionScript, требует, чтобы переменные создавались до того, как к ним происходит обращение.) Формально существование переменной начинается с ее *объявления* с помощью особого ключевого слова *var*, которое вы использовали ранее.

Поэтому проинструктировать вас о том, как создать первую строку кода, можно более формальным языком: объявить новую переменную с именем `message` и присвоить ей начальное значение «Hi there, Flash!». Это означает, что в окне Actions вы должны ввести:

```
var message = "Hi there, Flash!";
```

Волшебник за кулисой (интерпретатор)

Вспомните первые две строки кода:

```
var message = "Hi there, Flash!";
var firstName = "здесь ваше имя";
```

В каждой из этих инструкций вы создавали переменную и присваивали ей значение. Однако третья и четвертая строки несколько отличаются от них:

```
trace (message);
trace ("Hi there, " + firstName + ", nice to meet you.");
```

В этих инструкциях используется команда *trace( )*. Вы уже видели действие этой команды – она заставила Flash вывести текст в окне Output. В третьей строке Flash вывел значение переменной `message`. В последней строке Flash также преобразовал переменную `firstName` в ее значение (то, которое вы ввели) и вставил его в предложение после слов «Hi there». После этого команда *trace( )* заставляет указанные данные появиться в окне Output (что удобно для слежения за происходящим во время выполнения программы).

Спрашивается, что заставило команду *trace( )* поместить ваш текст в окно Output? Когда вы создаете переменную или подаете команду, то фактически обращаетесь к *интерпретатору* ActionScript, который запускает ваши программы, управляет кодом, ждет указаний, выполняет все команды ActionScript и инструкции, запоминает ваши данные, посылает вам информацию, вычисляет значения и даже запускает основное программное окружение, когда фильм загружается во Flash Player.

Интерпретатор транслирует ваш ActionScript в тот язык, который понятен компьютеру и используется им для выполнения вашего кода. Во время воспроизведения фильма интерпретатор всегда активен и готов постараться понять команды, которые вы ему подаете. Если интерпретатору понятны ваши команды, он посылает их на выполнение процессору компьютера. Если команда генерирует результат, интерпретатор передает вам соответствующий отклик. Если интерпретатору не понятна команда, он посылает сообщение об ошибке или молча пропускает ее. Интерпретатор, следовательно, действует как волшебный джинн – выполняет приказы, которые вы отдаете в своем коде, и возвращает ответ Flash, когда приказы выполнены.

Разберем подробнее работу интерпретатора на примере обработки простого действия *trace( )*.

Рассмотрим следующую команду, как это сделал бы интерпретатор:

```
trace ("Nice night to learn ActionScript.");
```

Интерпретатор сразу распознает ключевое слово *trace* по специальному списку допустимых имен команд. Интерпретатору известно также, что *trace( )* применяется для отображения текста в окне Output, поэтому он ждет указаний о том, какой текст выводить. Он находит «Nice night to learn ActionScript.» между круглыми скобками, следующими за словом *trace*, и думает: «Ага! Это как раз то, что мне нужно. Я сразу пошлю это в окно Output!»

Заметьте, что команда оканчивается точкой с запятой (;). Точка с запятой действует как точка в конце предложения. За немногими исключениями, все инструкции ActionScript должны оканчиваться точкой с запятой. Если инструкция понятна и есть вся необходимая информация, интерпретатор транслирует команду, чтобы ее мог выполнить процессор, в результате чего наш текст появляется в окне Output.

Это сильно упрощенное описание механизма работы процессора компьютера и интерпретатора, но оно иллюстрирует следующие обстоятельства:

- Интерпретатор всегда ждет ваших указаний.
- Интерпретатор должен прочесть код буква за буквой и постараться понять его. Примерно так человек старается прочесть и понять предложение в книге.
- Интерпретатор читает ваш ActionScript, применяя строгие правила: если бы, к примеру, отсутствовали круглые скобки в инструкции *trace( )*, то интерпретатор не смог бы разобраться в том, что происходит, и команда не была бы выполнена.

Вы еще только познакомились с интерпретатором, но вскоре вы станете с ним близки, как с родным: ссоры и крики «Почему ты меня не слушаешь?!», и чудные мгновения полного взаимопонимания. Довольно странно, но мой отец всегда говорил мне, что лучший способ выучить новый язык состоит в том, чтобы влюбиться в кого-то, кто на нем разговаривает. Позвольте мне стать первым, кто пожелает вам счастья в новых отношениях между вами и интерпретатором ActionScript. С этого момента, описывая выполнение инструкций ActionScript, я буду говорить «интерпретатор», а не «Flash».

Требуются дополнительные данные (аргументы)

Вы уже сталкивались с передачей интерпретатору текста, который нужно отобразить, при подаче команды *trace( )*. Это обычный подход: мы будем часто подавать команду и снабжать интерпретатор дополнительными данными, используемыми при выполнении этой команды. Для данных, передаваемых команде, есть специальное название – *аргумент*, или, что то же самое, *параметр*. Чтобы послать команде аргумент, его надо заключить в круглые скобки:

```
команда(аргумент);
```

Чтобы снабдить команду несколькими аргументами, их надо разделить запятыми:

```
команда(аргумент1, аргумент2, аргумент3);
```

Посылка команде аргументов называется *передачей* аргументов. Например, в коде *gotoAndPlay(5)* именем команды является *gotoAndPlay*, а *5* представляет собой передаваемый аргумент (в данном случае номер кадра). Некоторые команды, например *stop( )*, требуют после себя скобок, но не принимают аргументы. Почему так происходит, мы узнаем в главе 9, где функции будут рассмотрены подробно.

Связующие элементы ActionScript (операторы)

Взглянем еще на четвертую строку кода, содержащую инструкцию *trace( )*:

```
trace ("Hi there, " + firstName + ", nice to meet you.");
```

Видите знаки «плюс»? Они применяются для соединения вместе (*конкатенации*) нашего текста и представляют собой один из многих имеющихся *операторов* (*operators*). Операторы в языке программирования сродни союзам («и», «или», «но» и т. д.) в естественных языках. Это средства для соединения фраз кода и манипуляций над ними. В примере *trace( )* оператор «плюс» присоединяет заключенный в кавычки текст «Hi there,» к тексту, содержащемуся в переменной `firstName`.

Все операторы связывают фразы кода воедино в процессе их выполнения. Оператор почти всегда осуществляет какой-то вид трансформации фраз, состоящих из текста, цифр или других типов данных. Очень часто операторы соединяют вместе два элемента, как это делает оператор «плюс». А другие операторы сравнивают значения, присваивают значения, участвуют в принятии логических решений, определяют типы данных, создают новые объекты и предоставляют другие полезные услуги.

При использовании с числовыми операндами знаки «плюс» (+) и «минус» (–) осуществляют элементарные арифметические действия. Следующий код выводит в окне Output число 3:

```
trace(5 - 2);
```

Оператор меньше проверяет, какое из двух чисел меньше или какая из двух букв раньше встречается в алфавите:

```
if (3 < 300) {
  // Сделать что-то...
}
if ("a" < "z") {
  // Сделать что-то еще...
}
```

Сочетания, сравнения, присваивание и прочие манипуляции, совершаемые операторами, называются *операциями*. Арифметические операции понять легче всего, потому что они выполняют элементарные математические действия: сложение (+), вычитание (–), умножение (*) и деление (/). Но некоторые операторы покажутся вам не столь понятными, так как они выполняют специальные задачи программирования. Возьмем, к примеру, оператор *typeof*. Он сообщает о том, какого вида данные хранятся в переменной. Поэто-

му если создать переменную x и присвоить ей значение 4, то можно спросить у интерпретатора, какой тип данных содержится в x:

```
var x = 4;
trace (typeof x);
```

Когда эта строка кода будет выполнена во Flash, в окне Output будет получено слово «number». Обратите внимание, что оператору *typeof* передается значение, над которым он должен выполнить действие, то есть x, но при этом не используются круглые скобки: *typeof* x. Поэтому может показаться непонятным, является ли x *аргументом typeof*. На самом деле x играет ту же роль, что и аргумент (это вспомогательные данные, необходимые при выполнении фразы кода), но в контексте оператора его принято называть *операндом*. Операнд – это то, над чем оператор выполняет действие. Например, в выражении *4 + 9* числа 4 и 9 представляют собой операнды оператора +.

В главе 5 подробно рассказано обо всех операциях над данными в ActionScript. Сейчас просто запомните, что операции связывают фразы кода в выражения.

Соединяем все вместе

Повторим теперь все изученное. Вот снова первая строка:

```
var message = "Hi there, Flash!";
```

Ключевое слово *var* сообщает интерпретатору, что мы объявляем (создаем) новую переменную. Слово message представляет собой имя нашей переменной, выбранное произвольно. Знак равенства является оператором, присваивающим текстовую строку («Hi there, Flash!») переменной message. Текст «Hi there, Flash!» становится в результате значением message. Наконец, точка с запятой (;) сообщает интерпретатору, что мы завершили свою первую инструкцию.

Вторая строка очень похожа на первую:

```
var firstName = "здесь ваше имя";
```

Здесь мы присваиваем текстовую строку, набранную вместо *здесь ваше имя*, переменной firstName. Точка с запятой завершает наше второе предложение.

Затем мы снова используем переменные message и firstName в третьей и четвертой строках:

```
trace (message);
trace ("Hi there, " + firstName + ", nice to meet you.");
```

Ключевое слово *trace* сообщает интерпретатору, что надо вывести в окно Output некоторый текст. Текст, который надо вывести, передается как аргумент. Открывающая круглая скобка обозначает начало аргумента. Команде *trace( )* требуется один аргумент, но им может быть достаточно сложное выражение. Например, в четвертой строке аргумент содержит две *операции*, каждая из которых использует *оператор* «плюс», соединяя три *операнда*.

Первая операция присоединяет свой первый *операнд*, «Hi there,», к значению второго операнда, `firstName`. Вторая операция присоединяет текст «nice to meet you» к результату первой. Закрывающая круглая скобка обозначает конец аргумента, а точка с запятой снова указывает на конец предложения. Формально, круглые скобки ограничивают список из одного или нескольких аргументов. Если команде требуется несколько аргументов, они отделяются друг от друга запятыми (запятые в предшествующем примере являются частью текста, потому что заключены в двойные кавычки, и их не следует путать с запятыми, разделяющими аргументы (см. ниже).

Вот и готова ваша первая программа ActionScript! Здорово звучит, и это важное событие.

Другие понятия ActionScript

Вы уже познакомились с многими базовыми элементами ActionScript: данными, переменными, инструкциями, операторами, функциями и аргументами. Прежде чем глубже погрузиться в эти темы, обрисуем остальные главные характеристики ActionScript.

Программы Flash

Для большинства пользователей компьютеров *программа* (*program*) является синонимом *приложения* (*application*), такого как Adobe Photoshop или Macromedia Dreamweaver MX. Очевидно, это не то, что мы создаем, программируя на Flash. С другой стороны, программисты определяют программу как совокупность кода («последовательность инструкций»), но это будет лишь частью того, что мы строим.

Фильм Flash – это больше, чем последовательность строк кода. Во Flash код перемежается элементами фильма, такими как кадры, клипы и кнопки. В конечном счете не существует такой вещи, как «программа» Flash в классическом смысле этого слова. На ActionScript мы пишем не законченные программы, а сценарии (*scripts*): участки кода, которые задают программируемое поведение нашего фильма, так же, как сценарии JavaScript задают программируемое поведение документов HTML. То, что мы создаем, представляет собой не программу, а законченный фильм Flash (экспортируемый файл *.swf*, содержащий код, временные диаграммы, зрительные и звуковые элементы и прочее ценное содержимое).

Наши сценарии содержат большую часть того, что можно увидеть в обычных программах, но без всех тех вещей уровня операционной системы, что пишут на языках типа C++ или Java для вывода на экран графики или воспроизведения звука. Мы избавились от необходимости вникать в технические подробности программирования графики и звука, что позволяет сосредоточиться на разработке поведения наших фильмов.

Выражения

Сценарий, как мы выяснили, содержит программные действия – инструкции. Но инструкции обычно бесполезны при отсутствии данных. Когда мы, например, устанавливаем переменную, то присваиваем некоторые данные в качестве ее значения. Когда мы используем команду *trace( )*, то передаем данные в качестве аргумента для вывода в окне Output. Данные представляют собой содержимое, которое мы обрабатываем в своем коде ActionScript. В своих сценариях вы будете извлекать, передавать, запоминать и вообще повсюду рассылать массу данных.

В программе любая фраза кода, выдающая во время выполнения один элемент данных, называется *выражением*. Число 7 и строка «Добро пожаловать на мой веб-сайт» являются простейшими выражениями. Они представляют простые данные, которые в таком виде и будут использованы при выполнении программы. Поэтому такие выражения называются символьными *выражениями*, или просто *литералами*.

Литералы – лишь один тип выражений. Переменная тоже может быть выражением (переменные содержат данные и могут находиться везде, где нужны данные, поэтому считаются выражениями). Выражения становятся еще более интересными, когда объединяются между собой с помощью операторов. Например, составное выражение *4 + 5* представляет собой выражение с двумя операндами, 4 и 5, но оператор сложения заставляет все выражение выдать одиночное значение 9. Сложные выражения могут содержать другие, более короткие выражения, если получившаяся конструкция в целом может быть преобразована в одиночное значение.

Вот мы видим переменную `message`:

```
var message = "Hi there, Flash!";
```

При желании можно объединить переменное выражение `message` с символьным выражением «How are you?» следующим образом:

```
message + " How are you?"
```

Все это во время выполнения программы превратится в «Hi there, Flash! How are you?». При работе с арифметикой длинные выражения часто включают в себя более короткие, например:

```
(2 + 3) * (4 / 2.5) - 1
```

Начиная программировать, важно разобраться с выражениями, потому что термин «выражение» часто используется при описании понятий программирования. Например, можно было написать так: «Чтобы присвоить значение переменной, введите ее имя, затем знак равенства и произвольное выражение».

Два важных типа инструкций: условные и циклические

Почти во всех программах используются *условные инструкции* (*conditionals*), чтобы добавить в программу логику, и *циклы* (*loops*) для выполнения повторяющихся задач.

Выбор с помощью условных инструкций

Одной из действительно стоящих черт Flash-программирования является возможность сделать свои фильмы умными. Что я имею в виду? Представьте себе, что есть девушка по имени Уэнди, которая не любит, чтобы ее одежда промокала под дождем. Каждое утро, перед тем как выйти из дома, Уэнди смотрит в окно и проверяет, какая на улице погода. Если идет дождь, она берет с собой зонтик. Уэнди умная. Она использует элементарную логику – способность видеть ряд вариантов и принимать решение в зависимости от обстоятельств. Такую же элементарную логику мы применяем при создании интерактивных фильмов Flash.

Вот несколько примеров логики в фильмах Flash:

- Предположим, что в нашем фильме три части. Когда пользователь переходит в одну из частей, мы используем логику, чтобы решить, показывать ли ему введение для этой части. Если он уже был в этой части, мы пропускаем введение, в противном случае показываем его.
- Предположим, что доступ к части фильма ограничен. Для входа в зону с ограниченным доступом пользователь должен ввести пароль. Если пароль правильный, мы показываем содержимое с ограниченным доступом, а если неправильный – нет.
- Допустим, что мы перемещаем шарик по экрану и хотим, чтобы он отскакивал от стены. Если шарик проходит некоторую точку, мы меняем направление его движения. В противном случае позволяем шарику продолжить движение в том же направлении.

Такие примеры логики в фильме требуют применения инструкций особого типа, называемых *условными*. Условные инструкции позволяют задать условия, при которых некоторый раздел кода должен (или не должен) выполняться. Вот пример условной инструкции:

```
if (userName == "James Bond") {
  trace ("Добро пожаловать, 007.");
}
```

Общая структура условной инструкции такова:

```
if (это условие выполняется) {
  тогда выполнить эти строки кода
}
```

Подробнее о синтаксисе условных инструкций вы узнаете из главы 7. А пока запомните, что условные предложения позволяют Flash принимать логические решения.

Выполнение повторяющихся задач с помощью циклов

Мы хотим, чтобы наши фильмы не только принимали решения, но и решали за нас утомительные повторяющиеся задачи. То есть, пока они не захватили весь мир и не поработили нас, начав выращивать в маленьких энергетических коконах, как... стоп... забудьте то, что я сейчас сказал... Допустим, что вы хотите вывести в окно Output серию из пяти чисел, начинающуюся с определенного числа. Если первое число последовательности равно 1, то ее можно вывести так:

```
trace (1);
trace (2);
trace (3);
trace (4);
trace (5);
```

Для того чтобы начать последовательность с 501, придется заново ввести все числа:

```
trace (501);
trace (502);
trace (503);
trace (504);
trace (505);
```

Можно избежать перепечатывания, сделав команды *trace( )* зависимыми от переменной, например:

```
var x = 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
x = x + 1;
trace (x);
```

В строке 1 мы присваиваем переменной `x` значение 1. Затем, в строке 2, мы посылаем это значение в окно Output. В строке 3 мы говорим: «Возьми текущее значение `x`, добавь к нему 1 и помести результат обратно в переменную `x`», поэтому `x` становится равным 2. Затем мы снова посылаем значение `x` в окно Output. Этот процесс мы повторяем еще три раза. В результате в окно Output выводится последовательность из пяти чисел. Прелесть в том, что если теперь потребуется сделать начальным числом последовательности 501, достаточно будет изменить начальное значение `x` с 1 на 501. Так как весь последующий код использует `x`, при выполнении программы меняется вся последовательность.

Это усовершенствование нашего первоначального подхода работает довольно успешно, пока надо вывести всего пять чисел, но оказывается непригод-

ным, если количество чисел увеличивается до 1000. Для выполнения многократно повторяющихся задач применяется *цикл* – управляющая конструкция, заставляющая блок кода (то есть одну или несколько команд) повторяться произвольное количество раз. Есть несколько типов циклов, каждый из которых имеет собственный синтаксис. Очень часто применяется цикл типа *while*. Вот как может выглядеть наш пример с применением цикла *while* вместо серии повторяющихся инструкций:

```
var x = 1;
while (x <= 5) {
  trace (x);
  x = x + 1;
}
```

Ключевое слово *while* указывает на то, что мы хотим начать цикл. Выражение (*x <= 5*) определяет, сколько раз должен исполниться цикл (пока x остается меньше или равным 5), а команды *trace(x)*; и *x = x + 1;* выполняются при каждом повторении (или *итерации*) цикла. В данном случае цикл экономит нам только пять строк кода, а мог бы сэкономить сотни строк, если бы мы считали до чисел более высокого порядка. И кроме того, наш цикл гибок. Чтобы сосчитать до 1000, достаточно изменить выражение (*x <= 5*) на (*x <= 1000*):

```
var x = 1;
while (x <= 1000) {
  trace (x);
  x = x + 1;
}
```

Как и условные инструкции, циклы относятся к наиболее часто используемым и важным типам инструкций в программировании.

Модульный код (функции)

До сих пор ваш самый длинный сценарий состоял из пяти строчек кода. Но пройдет немного времени, и эти 5 строк превратятся в 500, а то и 5000 строк. Рано или поздно вы станете думать, как эффективнее управлять кодом, сократить объем работы и облегчить использование вашего кода в различных сценариях. Тогда вы впервые действительно полюбите *функции*. Функция – это объединенная в один пакет последовательность инструкций, выполняющих какое-либо полезное действие или расчет. На практике функции обычно служат в качестве многократно используемых блоков кода.

Предположим, что вы хотите написать сценарий, который быстро подсчитывает площадь прямоугольника. Без использования функций ваш сценарий мог бы выглядеть так:

```
var height = 10;
var width = 15;
var area = height * width;
```

А теперь допустим, что надо вычислить площади пяти прямоугольников. Ваш код увеличивается в размерах впятеро:

```
var height1 = 10;
var width1 = 15;
var area1 = height1 * width1;
var height2 = 11;
var width2 = 16;
var area2 = height2 * width2;
var height3 = 12;
var width3 = 17;
var area3 = height3 * width3;
var height4 = 13;
var width4 = 18;
var area4 = height4 * width4;
var height5 = 20;
var width5 = 5;
var area5 = height5 * width5;
```

Мы многократно повторяем расчет площади, поэтому лучше один раз поместить его в функцию и затем выполнять ее многократно:

```
function area(height, width){
  return height * width;
}
area1 = area(10, 15);
area2 = area(11, 16);
area3 = area(12, 17);
area4 = area(13, 18);
area5 = area(20, 5);
```

Сначала мы создали и определили функцию расчета площади с помощью ключевого слова *function*, объявляющего функцию подобно тому, как *var* объявляет переменную. Затем мы дали нашей функции имя, `area`, подобно тому, как мы даем имена переменным. Между круглыми скобками мы перечислили аргументы, принимаемые нашей функцей при каждом ее вызове, – `height` и `width`. А в фигурные скобки { } заключены команды, которые должна выполнять наша функция:

```
return height * width;
```

Создав функцию, мы можем выполнять код, содержащийся в ней, из любого места нашего фильма, указывая ее имя. В нашем примере мы вызвали функцию *area( )* пять раз, каждый раз передавая ей значения `height` и `width`, которые она должна получить: *area(10, 15)*, *area(11, 16)* и т. д. Результаты всех вычислений возвращаются, и мы записываем их в переменные от `area1` до `area5`. Мило и точно, и работы значительно меньше, чем в версии кода без функций. Помещение многократно выполняемого кода в функции экономит время и уменьшает количество строк в коде. Кроме того, код становится модульным, и его значительно легче читать. Тот, кто читает этот код, сможет угадать, для чего предназначена функция *area( )*, просто по ее имени.

Не волнуйтесь, если у вас возникли вопросы в связи с этим примером функции: в главе 9 мы узнаем о функциях больше. Пока просто запомните, что функции дают нам чрезвычайно мощный способ создания сложных систем. Функции помогают повторно использовать код и упаковывать его функциональность, позволяя расширить границы того, что можно построить.

Встроенные функции

Обратите внимание, что функции принимают аргументы, подобно тому как это делает команда *trace( )*. Вызов функции *area(4, 5)*; очень похож на подачу команды *trace( )*, например *trace(x)*;. Разница в том, что *area( )* требует два аргумента, разделенных запятой, а команда *trace( )* – один аргумент. Сходство не является простым совпадением. Как отмечалось выше, многие действия (Actions), в том числе *trace( )*, фактически представляют собой функции. Но это особый тип функций, которые встроены в ActionScript (в противоположность функциям, определяемым пользователем, таким как наша *area( )*). Поэтому допустимо (и технически более точно) сказать «вызвать функцию *gotoAndStop( )*», чем сказать «выполнить действие *gotoAndStop*». Встроенная функция – это просто многократно используемый блок кода, который для удобства мы получаем вместе с ActionScript. Встроенные функции позволяют делать все – от математических вычислений до управления клипами фильмов. Все встроенные функции перечислены в части II «Справочник по языку». Мы также столкнемся с многими из них, изучая основы ActionScript.

Объекты и объектно-ориентированное программирование

Как мы уже знаем, *инструкция* (*statement*) – это команда ActionScript, которая заставляет Flash что-то сделать. И мы знаем, что *функция* объединяет несколько инструкций в одну удобную команду. Функции вносят в код структуру, благодаря которой можно строить более сложные программы. Следующий логический шаг делают *объекты* – они упаковывают ряд связанных между собой функций и переменных в единый модуль кода.

По мере того как растет сложность программы, становится все труднее описывать ее работу в виде одной линейной последовательности команд (инструкций). Объекты помогают концептуально моделировать поведение программы как ряда взаимосвязанных самодостаточных компонентов, а не длинного списка команд. В примере с прыгающим шариком, приведенном в начале главы, можно было представить шарик и квадратную область, в которой он движется, в виде объектов. Объект шарика содержал бы функции, которые относятся к шарику, такие как *startMoving( )* (начать движение), *stopMoving( )* (прекратить движение) и *checkForWall( )* (проверить стенку). Он также содержал бы информацию о шарике, хранящуюся в переменных, таких как `velocity` (скорость), `color` (цвет) и `diameter` (диаметр). Объект пространства мог бы содержать такие функции, как *setRoomSize( )* (задать размер пространства) и *addBall( )* (добавить шарик), а также переменные `width`

(ширина) и `height` (высота). Благодаря объектам концептуальная модель программы отражается в интуитивно понятном синтаксисе:

```
room.addBall();
ball.diameter = 5;
ball.startMoving();
```

В общем виде, сначала идет объект (существительное), затем точка (`.`), далее функция (глагол) или переменная (прилагательное). Функция совершает с объектом какие-то действия, тогда как переменная что-то сообщает нам об объекте. Функции, связанные с объектом, часто называют *методами*. Переменные, связанные с объектом, часто называют *свойствами* (*properties*). Вот несколько примеров:

```
object.method();
boy.run();
someMovieClip.play();

object.property = value;
boy.speed = 5;
someMovieClip._width = 60;
```

Объектно-ориентированное программирование (ООП) – это подход, в соответствии с которым элементарными конструктивными элементами программы являются объекты. В зависимости от языка и склонностей разработчиков уровни ООП могут различаться. Пуристы настаивают на том, что каждая часть программы должна содержаться в объекте (для языка программирования Java это правило обязательно). Другие разработчики довольствуются тем, что используют объекты для структурирования лишь некоторых частей программы (это принято в ActionScript).

В любом случае знакомство с объектами необходимо, потому что они являются важной частью структуры ActionScript. Почти все, что есть во Flash – от кнопок до текстовых полей и клипов, – представляется в ActionScript объектами. Даже если отказаться от организации своего собственного кода в виде объектов, технологией объектно-ориентированного программирования придется пользоваться для того, чтобы управлять кнопками, текстом и клипами фильма. В главе 12 объектно-ориентированное программирование описывается с исчерпывающей полнотой. А пока можете представлять себе элементы Flash, например клипы, как объекты, способные выполнять определенные задачи (например, воспроизводиться методом *play( )*) или хранить информацию (например, в переменной `_width`).

Экземпляры клипов

Я надеюсь, что за всеми этими разговорами об основах программирования вы не забыли об основах Flash. Одним из ключевых понятий визуального программирования во Flash являются *экземпляры* (*instances*) клипов.[1] Как

[1] Здесь и далее «movie clip» упоминается как «клип» или «символ». – *Примеч. науч. ред.*

Flash-разработчик вы должны быть уже знакомы с клипами, но возможно, не воспринимаете их как инструменты программирования.

Для каждого клипа в библиотеке фильма Flash имеется исходный символ. Можно добавлять несколько копий, или *экземпляров*, одного символа в фильм Flash, перетаскивая клип из библиотеки в рабочую область. С помощью ActionScript можно также создавать клипы в процессе воспроизведения ролика. В любом визуальном программировании во Flash в той или иной степени присутствует управление экземпляром клипа. Например, прыгающий мячик – это не что иное, как экземпляр клипа, положение которого в рабочей области периодически изменяется. ActionScript позволяет изменять местоположение экземпляра, его размер, текущий кадр, угол поворота и т. д. во время воспроизведения нашего фильма.

Если вы не знакомы с клипами и экземплярами, то прежде чем продолжить работу с данной книгой, почитайте документацию по работе с клипами и кнопками, открыв пункт Help → Using Flash.

Модель исполнения, управляемая событиями

Последняя тема, которую мы должны рассмотреть в нашем обзоре основ ActionScript, – это модель исполнения, которая определяет, в какой момент выполняется код в фильме. Код может быть прикреплен к различным кадрам, кнопкам и клипам по всему фильму. Но когда фактически он выполняется? Чтобы ответить на этот вопрос, совершим краткую прогулку по истории компьютерных вычислений.

На заре развития программирования инструкции программы выполнялись последовательно в порядке своего появления, начиная с первой строки и заканчивая последней. Программа должна была совершить какие-то действия и затем остановиться. Такого рода программы, называемые *пакетными* (*batch*), не могут обеспечить интерактивность, необходимую для программной среды, *управляемой событиями* (*event-based*), такой как Flash.

Программы, управляемые событиями, не выполняются линейным образом. Они выполняются непрерывно (в *цикле событий*, *event loop*) в ожидании, когда что-нибудь произойдет (*событие*, *event*), и в ответ на эти события выполняются определенные фрагменты кода. В языках, предназначенных для использования в интерактивной визуальной среде (например, ActionScript или JavaScript), событиями обычно являются действия пользователя, такие как щелчок мыши или нажатие клавиши.

Когда происходит событие, интерпретатор извещает о нем программу. Программа может реагировать на этот сигнал тем, что попросит интерпретатор выполнить некоторый участок кода. Например, если пользователь щелкает мышью в фильме, можно выполнить какой-нибудь код, который покажет другую часть фильма (навигация) или передаст переменные в базу данных (подача формы).

Но программы будут реагировать на события только в том случае, если мы создадим *обработчики событий* (*event handlers*) и/или *приемники событий*

(*event listeners*), которые сообщают интерпретатору, какую функцию надо выполнить при возникновении определенного события. Во Flash MX обработчик события в общем виде создается так:

```
someObject.onSomeEvent = someFunction;
```

Flash MX допускает такой синтаксис событий даже для клипов и кнопок, тогда как во Flash 5 для клипов и кнопок требовался особый синтаксис событий. Подробности см. в главе 10.

События всегда происходят по отношению к какому-то объекту в программе. Например, кнопка может поворачивать клип, в котором она расположена, с помощью следующего обработчика события:

```
rotateButton.onRelease = rotate;
```

На обычном языке это указание интерпретатору можно описать так: «Когда кнопка мыши отпускается над объектом `rotateButton`, выполнить функцию *rotate*». Функция *rotate* может выглядеть так:

```
function rotate () {
  this._parent._rotation = 45;
}
```

Функция, которая выполняется, когда происходит событие, называется *функцией обратного вызова* (*callback function*). Как мы узнаем в главе 10, внутри функции обратного вызова ключевое слово `this` указывает на объект, в котором определен обработчик события (в данном случае это `rotateButton`). В случае кнопки, реагирующей на щелчок мыши, `this` указывает на кнопку, по которой был сделан щелчок. В объектно-ориентированном синтаксисе к клипу, в котором находится кнопка, обращаются как к `this._parent` (клип является родителем кнопки, поскольку содержит ее). Наконец, мы задаем угол поворота родительского клипа равным 45°, присваивая значение 45 свойству `this._parent._rotation`:

```
this._parent._rotation = 45;
```

Это буквально переводится как «установить угол поворота родительского клипа данной кнопки равным 45°».

Зачастую наш пример обработчика события кнопки более кратко записывают так:

```
rotateButton.onRelease = function () {
  this._parent._rotation = 45;
};
```

Программы, управляемые событиями, всегда выполняют цикл ожидания событий, находясь в готовности реагировать на очередное событие. События играют важнейшую роль в фильмах Flash. Без событий наши сценарии ничего бы не выполняли, за одним исключением: Flash выполняет любой код в кадре, когда воспроизводящая головка (playhead) входит в этот кадр. Такое событие настолько свойственно для Flash, что для него не требуется непременно явный обработчик – достаточно прикрепить код к кадру.

События буквально являются движущей силой всего, поэтому они и появились в конце первого дня обучения в школе ActionScript. Теперь вы знаете, в чем состоит написание сценариев и что управляет временем выполнения этих сценариев (события). Можно сказать, что вы готовы к попытке провести первый настоящий разговор.

Создание вопросника с вариантами ответов

Теперь, когда мы изучили основные принципы ActionScript, попробуем применить эти принципы в контексте реального фильма Flash. Мы начнем наше изучение прикладного программирования во Flash с создания вопросника с вариантами ответов, используя очень простые приемы, по большей части вам уже известные. В последующих главах мы будем возвращаться к нашему вопроснику, чтобы показать, как его можно усовершенствовать за счет более развитых концепций программирования, которые мы изучим. В итоге мы сделаем код более элегантным, что облегчит его развитие и сопровождение, а также добавим в наш вопросник новые характеристики, чтобы он мог легко обрабатывать любое количество вопросов.

Готовый файл *.fla* для этого вопросника можно найти в сетевом хранилище кода, о котором говорилось в предисловии. Это урок по программированию Flash, а не продукт Flash. Предполагается, что вы уже можете без труда создавать и использовать кнопки, слои, кадры, ключевые кадры и инструмент Text. Вопросник показывает практическое применение следующих аспектов программирования на ActionScript:

- Переменные
- Управление порядком воспроизведения кадров фильма с помощью функций
- Обработчики событий кнопок
- Простые условные инструкции
- Переменные текстовых полей для вывода информации на экран

Обзор вопросника

Наш вопросник, часть которого показана на рис. 1.4, будет содержать только два вопроса. На каждый вопрос есть три возможных ответа. Пользователь представляет свои ответы, щелкая по кнопкам, соответствующим выбранным вариантам. Результаты выбора, сделанного пользователем, записываются в переменную, чтобы с ее помощью оценить потом результат пользователя. Когда даны ответы на все вопросы, подсчитывается количество правильных ответов и выводится балл, набранный пользователем.

Создание структуры слоев

При создании фильмов Flash важно разделить содержимое на контролируемые части путем размещения различных элементов содержимого в отдельных слоях. Разложение содержимого по слоям вообще является хорошим

Рис. 1.4. *Вопросник*

технологическим приемом, но для программирования Flash оно особенно важно. В нашем вопроснике, как и в большинстве наших фильмов со сценариями, мы будем хранить все сценарии временных диаграмм в одном отдельном слое с именем *scripts*. Я держу слой *scripts* первым в своем стеке слоев, чтобы его было легко найти.

Кроме того, все метки кадров мы будем хранить в отдельном слое, который назовем *labels*. Слой *labels* на всех ваших временных диаграммах должен располагаться ниже слоя *scripts*. Помимо этих двух стандартных слоев (*scripts* и *labels*) наш вопросник имеет несколько слоев, по которым мы распределим свои активы содержимого.

Начните построение вопросника с создания и именования следующих слоев (с помощью Insert → Layer) и расположите их в указанном порядке:

scripts
labels
quiz end
question 2
question 1
choice buttons
housing

Теперь добавьте по 30 кадров в каждый из этих слоев, выделив 30 кадров на временной диаграмме и выбрав Insert → Frame или нажав клавишу F5. Ваша временная диаграмма должна иметь вид, соответствующий рис. 1.5.

Рис. 1.5. *Начальная настройка временной диаграммы вопросника*

Создание интерфейса и вопросов

Прежде чем заниматься сценариями, управляющими вопросником, надо создать вопросы и интерфейс, который позволит пользователю этот вопросник пройти.

Вот шаги, которые требуется выполнить:

1. Выбрав кадр 1 в слое *housing*, с помощью инструмента Text введите название вопросника прямо на сцене.
2. В кадре 1 слоя *question 1* добавьте номер вопроса «1» и текст вопроса Question 1: «When were movie clips introduced into Flash?» (Когда во Flash впервые появились клипы?). Оставьте ниже вопроса место для текста ответов и кнопок.
3. Под текстом вопроса (по-прежнему в слое *question 1*) добавьте текст трех ответов: «Version 1», «Version 2» и «Version 3» – для каждого на отдельной строке.
4. Question 1 будет выступать в качестве шаблона для Question 2. Выделите первый кадр слоя *question 1* и выберите Edit → Copy Frames (не Edit → Copy).

5. Выделите кадр 10 в слое *question 2* и выберите Edit → Paste Frames (а не Edit → Paste). В слое *question 2* в кадре 10 появится копия вашего вопроса Question 1.
6. Чтобы под вопросом 2 не появился вопрос 1, добавьте пустой ключевой кадр в кадре 10 слоя *question 1* с помощью Insert → Blank Keyframe.
7. В том же кадре 10 слоя *question 2* измените номер вопроса с «1» на «2» и текст вопроса на «When was MP3 audio support added to Flash?» (Когда во Flash была введена поддержка аудиоформата MP3?). Ответы поменяйте на «Version 3», «Version 4» и «Version 5».

Вопросы почти готовы, но надо еще добавить кнопки для ответов пользователя на каждый вопрос:

1. Создайте символ простой кнопки (Insert → New Symbol → Button) типа флажка или переключателя размером не выше строки текста (см. рис. 1.4).
2. В кадре 1 слоя *choice buttons* рядом с тремя ответами на вопрос 1 поместите три экземпляра кнопки-флажка.
3. Выберите самую верхнюю кнопку (рядом с первым ответом) и с помощью инспектора свойств (Property inspector) установите для <Instance Name> значение `choice1_btn`.
4. Повторите шаг 3 и переименуйте оставшиеся две кнопки ответов `choice2_btn` и `choice3_btn` (сверху вниз).

На рис. 1.6 показано, как будет выглядеть временная диаграмма после помещения двух вопросов в вопросник. Обратите внимание, как мы используем кадры на временной диаграмме для представления так называемых *состояний приложения* (*application states*). Каждый «экран» приложения получает свой собственный кадр. Позднее мы добавим в кадры вопросов метки и увидим, как это способствует перемещению между кадрами в нашей программе, позволяя с легкостью показывать нужное состояние – в данном случае соответствующий вопрос.

Рис. 1.6. *Временная диаграмма вопросника с двумя вопросами*

Инициализация вопросника

Первейшей задачей для нашего сценария вопросника (и большинства сценариев) является создание переменных главной временной диаграммы, которые будут использоваться на протяжении всего фильма.

В нашем вопроснике мы делаем это в первом кадре фильма, но в других фильмах мы обычно будем делать это после частичной или полной загрузки фильма. В любом случае нам надо инициализировать свои переменные прежде, чем будут осуществляться какие-либо действия сценария.

Когда наши переменные будут определены, мы вызовем функцию *stop( )*, чтобы пользователь остановился на первом кадре (в котором начинается вопросник).

В более сложных случаях можно устанавливать начальное состояние путем вызова функций, осуществляя подготовку к оставшейся части фильма. Этот этап называется *инициализацией*. Функции, запускающие в движение процессы или определяющие начальные условия действия системы, часто называют *инициализирующими* (*init*).

Прежде чем прикреплять инициализирующий код нашего вопросника, показанный в примере 1.1, к кадру 1 слоя *scripts* нашего фильма, рассмотрим, какие действия он выполняет.

Пример 1.1. Инициализирующий код вопросника

```
// Инициализировать переменные главной временной диаграммы
var q1answer;          // Ответ пользователя на вопрос 1
var q2answer;          // Ответ пользователя на вопрос 2
var totalCorrect = 0;  // Счетчик правильных ответов

// Остановить фильм на первом вопросе
stop();
```

Строка 1 нашей инициализирующей последовательности представляет собой *комментарий*. Комментарии – это заметки, добавляемые в код для пояснения производимых действий. Однострочный комментарий начинается с двух символов косой черты и (желательно) пробела, за которыми следует строка текста:

```
// Это комментарий
```

Заметьте, что комментарии можно помещать в конце строки с кодом, например:

```
x = 5; // Это тоже комментарий
```

В строке 2 примера 1.1 создается переменная с именем `q1answer`. Вспомните, что для создания переменной применяется ключевое слово *var*, за которым следует имя переменной, например:

```
var favoriteColor;
```

Итак, строки нашего кода со второй по четвертую объявляют переменные, которые нам понадобятся, и дополнены комментариями, объясняющими их назначение:

- `q1answer` и `q2answer` будут содержать значение ответа пользователя (1, 2 или 3, указывая на номер варианта, выбранный для каждого вопроса). Эти значения понадобятся для проверки правильности ответов пользователя.
- `totalCorrect` будет использоваться в конце опроса для подсчета количества правильных ответов.

Взгляните внимательнее на строку 4 примера 1.1:

```
var totalCorrect = 0;  // Счетчик правильных ответов
```

Строка 4 выполняет двойную задачу. Сначала она объявляет переменную `totalCorrect`, а затем присваивает этой переменной значение 0 с помощью оператора присваивания =. Нам нужно, чтобы значением `totalCorrect` был 0, поскольку в начале пользователь не ответил правильно ни на один вопрос. Другим переменным значения по умолчанию не требуются, потому что они получат значения явным образом во время опроса.

Определив переменные, мы вызываем функцию *stop( )*, которая останавливает воспроизведение фильма на текущем кадре – в данном случае на кадре 1, с которого начинается вопросник:

```
// Остановить фильм на первом вопросе
stop();
```

Снова обратите внимание на комментарий перед вызовом функции *stop( )*. Он поясняет, к какому результату должен привести последующий код.

Комментарии могут отсутствовать, но они помогают прояснить код, когда мы некоторое время не работаем с ним, а затем снова возвращаемся к нему и должны вспомнить его, либо если передаем код другому разработчику. Кроме того, комментарии облегчают просмотр кода, что важно во время отладки.

Итак, вы разобрались, что делает инициализирующий код, и пора добавить его в кадр 1 слоя *scripts* нашего фильма-вопросника:

1. Выделите кадр 1 слоя *scripts*.
2. Выберите Window → Actions (F9). Появится окно Frame Actions.
3. Убедитесь, что вы находитесь в режиме Expert Mode, который можно сделать постоянной настройкой с помощью всплывающего меню Options в правом верхнем углу окна Actions.
4. В правой части окна Frame Actions введите инициализирующий код, показанный выше в примере 1.1.

Добавление меток кадров

Мы написали инициализирующий сценарий для нашего вопросника и создали вопросы. Теперь мы добавим метки для некоторых кадров, чтобы управлять воспроизведением нашего вопросника.

Для того чтобы пользователь в каждый момент времени работал только с одним вопросом, мы разделили содержимое вопросов 1 и 2 по кадрам 1 и 10. Перемещая воспроизводящую головку в эти ключевые кадры, мы создаем эффект показа слайдов, причем каждый слайд содержит свой вопрос. Мы знаем, что вопрос 2 находится в кадре 10, поэтому чтобы показать вопрос 2, можно воспользоваться функцией *gotoAndStop( )*:

```
gotoAndStop(10);
```

что вызовет перемещение воспроизводящей головки в кадр 10, где находится вопрос 2. Разумный код, правда? А вот и нет. Использование конкретного числа 10 в нашей функции *gotoAndStop( )* допустимо, но это негибкое решение (подобное использование литералов называют *жестким кодированием* (*hard-coding*) и часто осуждают). Если, например, поместить на временную диаграмму пять новых кадров перед кадром 10, то вопрос 2 сместится в кадр 15, и наша команда *gotoAndStop(10)* приведет пользователя не туда, куда требуется. Чтобы наш код мог работать даже в случае последующих изменений во временной диаграмме, мы будем использовать вместо номеров кадров *метки кадров*. Метки кадров – это описательные имена, например `q2` или `quizEnd`, с помощью которых можно ссылаться на конкретные точки временной диаграммы.

Как только точка помечена, можно обращаться к кадру по имени, а не по номеру. Метки кадров обеспечивают незаменимую гибкость. Я практически никогда не использую номера кадров с такими функциями управления воспроизведением, как *gotoAndStop( )*.

Давайте теперь добавим в наш вопросник все необходимые метки, с помощью которых будем потом проводить пользователя через вопросы:

1. В слое *labels* щелкните по кадру 1.
2. В инспекторе свойств задайте для <Frame Label> значение `init`.
3. В кадр слоя 10 *labels* добавьте пустой ключевой кадр.
4. В инспекторе свойств задайте для <Frame Label> значение `q2`.
5. В кадр 20 слоя *labels* добавьте пустой ключевой кадр.
6. В инспекторе свойств задайте для <Frame Label> значение `quizEnd`.

Сценарии для кнопок ответов

Наши вопросы на месте, наши переменные инициализированы, наши кадры помечены. Если мы сейчас протестируем свой фильм, то увидим, что появляется вопрос 1 с тремя кнопками ответа, которые ничего не делают при щелчке по ним, а попасть на вопрос 2 пользователь не может. Надо добавить в кнопки ответов какой-то код, который будет перемещать пользователя по вопроснику и попутно запоминать его ответы.

Стиль именования переменных

Вы уже видели достаточно много имен переменных, и у вас мог возникнуть вопрос об использовании заглавных букв. Тем, кто никогда не занимался программированием, прописная буква в середине слова, как в `firstName` или `totalCorrect`, может показаться необычной. Такое выделение первой буквы второго и последующих слов в имени переменной позволяет зрительно разделить слова, образующие имя. К этому приему прибегают потому, что наличие пробелов и дефисов в именах переменных недопустимо. Но не делайте прописной первую букву имени переменной: начальная прописная буква обычно используется в именах классов объектов, а не переменных.

Если вы используете для разделения слов в именах не заглавные буквы, а символы подчеркивания, как в `first_name` и `total_correct`, то будьте последовательны, то есть не используйте для одних переменных `firstName`, а для других `second_name`. Придерживайтесь какого-либо одного стиля, чтобы ваш код могли понять другие программисты. В некоторых языках программирования имена переменных чувствительны к регистру, то есть `firstName` и `firstname` рассматриваются как разные переменные. Однако ActionScript рассматривает их как одну переменную. Но плохо, когда на одну и ту же переменную ссылаются с использованием разных регистров: если вы назвали переменную `xPOS`, не следует где-либо ссылаться на нее как на `xpos`.

Всегда давайте переменным и функциям осмысленные имена, которые напоминают о том, для чего они используются. Избегайте бессмысленных имен типа *foo*, а однобуквенными переменными типа *x* или *i* обозначайте только что-то очень простое, например индекс (счетчик) в цикле. Не путайте *x* и *y* (как в наших вводных примерах, без подчеркивания) с *_x* и *_y*. В то время как *x* и *y* – произвольные имена переменных, выбираемые программистом, *_x* и *_y* представляют собой встроенные свойства, представляющие положение чего-либо по горизонтали и вертикали, например клипа. См. подробности в статье *MovieClip._x* «Справочника по языку».

Для некоторых видов данных ActionScript рекомендуется суффикс вида *_suffix* (см. таблицу 2.1 в главе 2). Например, клипы обозначаются суффиксом `_mc`, текстовые поля – суффиксом `_txt`, а кнопки – суффиксом `_btn`. Для ясности добавляйте эти суффиксы в конец имен переменных, например: `ball_mc` или `submit_btn`.

Вспомним, что мы назвали кнопки выбора `choice1_btn`, `choice2_btn` и `choice3_btn`, как показано на рис. 1.7.

Чтобы заставить кнопки вопроса 1 сохранять ответ пользователя и показывать следующий вопрос, применим код примера 1.2.

Рис. 1.7. *Кнопки ответов*

Пример 1.2. Код для кнопок вопроса 1

```
// Код, выполняемый при нажатии кнопки 1.
choice1_btn.onRelease = function () {
  this._parent.q1answer = 1;
  this._parent.gotoAndStop("q2");
};

// Код, выполняемый при нажатии кнопки 2.
choice2_btn.onRelease = function () {
  this._parent.q1answer = 2;
  this._parent.gotoAndStop("q2");
};

// Код, выполняемый при нажатии кнопки 3.
choice3_btn.onRelease = function () {
  this._parent.q1answer = 3;
  this._parent.gotoAndStop("q2");
};
```

Код для каждой кнопки состоит из двух инструкций, выполняемых только при щелчке мыши. На обычном языке код для каждой кнопки говорит: «Если пользователь щелкнет по этой кнопке, записать, что он выбрал ответ 1, 2 или 3, а затем перейти к вопросу 2». Разберем, как работает этот код, на примере кнопки 1:

```
choice1_btn.onRelease = function () {
  this._parent.q1answer = 1;
  this._parent.gotoAndStop("q2");
};
```

Строка 1 служит началом *обработчика события*. Она задает:

- Имя кнопки, которая должна реагировать на событие: `choice1_btn`.
- Имя события, на которое должна реагировать кнопка: *onRelease* (событие происходит, когда пользователь наводит курсор мыши на кнопку, щелкает и отпускает мышь над кнопкой).
- Функцию, которая должна быть выполнена при фактическом возникновении события: `function () { ... }`.

Функция, выполняемая в ответ на событие, называется функцией обратного вызова (*callback function*).

Открывающая фигурная скобка ({) обозначает начало блока инструкций, которые должны быть выполнены при возникновении события *onRelease*. Конец блока кода помечен закрывающей фигурной скобкой (}) и означает конец функции обратного вызова обработчика.

```
choice1_btn.onRelease = function () {
  ...
}
```

Обработчик события спокойно ждет, пока пользователь не щелкнет по кнопке 1. Когда происходит щелчок по кнопке, Flash выполняет нашу функцию обратного вызова, содержащую следующие две строки:

```
this._parent.q1answer = 1;
this._parent.gotoAndStop("q2");
```

Первая строка устанавливает переменную `q1answer` равной 1 (другие кнопки ответа устанавливают ее равной 2 или 3). Переменная `q1answer` запоминает ответ пользователя на первый вопрос. Однако правильный формат задает не только имя переменной `q1answer`, но и ее местонахождение по отношению к кнопке. Выражение `this._parent` означает «родительский клип этой кнопки», или «клип, содержащий эту кнопку». Ключевое слово `this` представляет саму кнопку, а `_parent` представляет клип родительского объекта кнопки. Поэтому полная инструкция `this._parent.q1answer = 1` означает: «в родительском клипе кнопки установить переменную `q1answer` равной 1». Если такой синтаксис вам внове, он может показаться слишком многословным. Однако, как мы узнаем из главы 2, у каждой переменной есть свое место (как правило, клип), и обычно требуется указывать не только имя переменной, но и ее местонахождение (например, `this._parent.q1answer`, а не просто `q1answer`).

Запомнив ответ пользователя на первый вопрос в `q1answer`, мы переходим ко второму вопросу с помощью кода, находящегося в строке 2 нашей функции обратного вызова:

```
this._parent.gotoAndStop("q2");
```

Строка 2 вызывает функцию *gotoAndStop( )*, передавая ей в качестве аргумента метку кадра «`q2`», в результате чего воспроизводящая головка перемещается в кадр `q2`, где находится вопрос 2. Однако мы снова должны указать относительно нажатой кнопки, в каком именно клипе воспроизводящая головка должна переместиться на метку `q2`. Так же, как и для нашей переменной, мы хотим управлять родительским клипом кнопки, поэтому вызов функции *gotoAndStop( )* мы предваряем `this._parent`.

Все наши кнопки ответа выполняют одни и те же базовые функции (устанавливают `q1answer` и показывают кадр `q2`). Следовательно, эти три обработчика событий кнопок отличаются только в двух отношениях:

Рис. 1.8. *Код в кадре 1*

- Каждый указывает свое имя экземпляра (`choice1_btn`, `choice2_btn`, `choice3_btn`).
- Каждый устанавливает свое значение `q1answer`, записывая фактический ответ пользователя на вопрос.

Для того чтобы применить обработчики событий к кнопкам вопроса 1, введите в свой сценарий код примера 1.2 под имеющимся кодом *init* в кадре 1 слоя *scripts*. Когда вы это сделаете, окно Actions для кадра 1 слоя *scripts* должно будет выглядеть, как на рис. 1.8.

Все это относилось к коду для кнопок вопроса 1. Займемся кнопками вопроса 2. Они действуют точно так же, как кнопки вопроса , но должны записывать ответы, выбранные пользователем, на вопрос 2, а не на вопрос 1, и показывать не следующий вопрос, а экран конца вопросника. К счастью, можно повторно использовать уже имеющиеся в фильме кнопки. Достаточно изменить функции обработчиков событий, выполняемые при нажатии кнопок. Чтобы переопределить обработчики событий для кнопок вопроса 2, выполните следующие действия:

1. В кадр 10 слоя *scripts* добавьте пустой ключевой кадр.

2. Не снимая выделения с кадра 10 слоя *scripts*, введите код примера 1.3 в окно Actions.

Любой сценарий, помещенный в ключевой кадр на временной диаграмме, автоматически выполняется, когда воспроизводящая головка входит в этот кадр. Следовательно, когда Flash будет показывать кадр 10, новый код кнопок будет применен к кнопкам вопроса 2.

Пример 1.3. Код кнопок вопроса 2

```
// Код, выполняемый при нажатии кнопки 1.
choice1_btn.onRelease = function () {
  this._parent.q2answer = 1;
  this._parent.gotoAndStop("quizEnd");
};

// Код, выполняемый при нажатии кнопки 2.
choice2_btn.onRelease = function () {
  this._parent.q2answer = 2;
  this._parent.gotoAndStop("quizEnd");
};

// Код, выполняемый при нажатии кнопки 3.
choice3_btn.onRelease = function () {
  this._parent.q2answer = 3;
  this._parent.gotoAndStop("quizEnd");
};
```

Наши обработчики событий кнопок для вопроса 2 такие же, какими они были для вопроса 1, за исключением того, что место переменной `q1answer` в них занимает переменная `q2answer`, поскольку мы хотим, чтобы кнопки отслеживали ответ пользователя на вопрос 2. А в качестве аргумента *gotoAndStop( )* мы используем `"quizEnd"`, чтобы переместить воспроизводящую головку в конец вопросника (то есть в кадр с меткой `quizEnd`) после того, как пользователь ответит на вопрос 2.

Добавив этот код для шести кнопок, вы, несомненно, обратите внимание на большие повторения в нем. Код одной кнопки отличается от кода другой лишь несколькими символами. Такое программирование не очень эффективно. Наш код для кнопок настоятельно требует наличия какой-то общей команды записи ответа и перехода к следующему экрану вопросника. В главе 9 мы узнаем, как централизовать наш код с помощью *функций*.

Создание концовки вопросника

Вопросник почти готов. У нас теперь есть два вопроса и сценарий, который запоминает ответы пользователя и позволяет ему перемещаться по вопроснику. Но нам еще нужен завершающий экран, чтобы оценить ответы и сообщить пользователю, каковы его успехи.

Чтобы создать завершающий экран нашего вопросника, надо еще немного поработать во Flash и со сценарием. Сначала поработаем во Flash:

Рис. 1.9. *Судный день*

1. В кадр 20 слоя *question 2* добавим пустой ключевой кадр. Это помешает содержимому вопроса 2 появиться под содержимым завершающего экрана.
2. К кадру 20 слоя *choice buttons* добавим пустой ключевой кадр. Это помешает нашим кнопкам появиться под содержимым завершающего экрана.
3. К кадру 20 слоя *quiz end* добавим пустой ключевой кадр.
4. Оставаясь в этом кадре, поместите на сцену такой текст: «Thank you for taking the quiz» (Спасибо за участие в опросе). Постарайтесь оставить внизу место для показа набранных пользователем очков.
5. К кадру 20 слоя *scripts* добавим пустой ключевой кадр.

Это обеспечит создание нашего завершающего экрана. Он должен выглядеть примерно так, как на рис. 1.9.

Теперь займемся сценарием конца вопросника. Когда воспроизводящая головка окажется в кадре `quizEnd`, нам потребуется сосчитать результат, полученный пользователем. Необходимо, чтобы при достижении воспроизводящей головкой кадра 20 был выполнен сценарий расчета, показанный в примере 1.4. Выберите кадр 20 в слое *scripts* и введите код примера в окно Actions.

Пример 1.4. Код окончания вопросника

```
// Подсчитаем правильные ответы пользователя
if (q1answer == 3){
  totalCorrect = totalCorrect + 1;
}
if (q2answer == 2){
  totalCorrect++;
}

// Создать на экране текстовое поле для показа результатов пользователя.
this.createTextField("totalOutput_txt", 1, 150, 200, 200, 20);

// Показать результат пользователя в текстовом поле на экране.
totalOutput_txt.text = "Your final score is: " + totalCorrect + "/2.";
```

В сценарии для расчетов мы сначала определим очки, набранные пользователем, а затем выведем его результат на экран. Строки 1, 9 и 12 (если считать промежуточные пустые строки) представляют собой комментарии, кратко описывающие назначение частей сценария. В строке 2 начинается первая из двух условных инструкций этого сценария. В нем мы видим, как найдено применение нашей переменной `q1answer`. Обратите внимание, что поскольку этот код прикреплен непосредственно к временной диаграмме нашего фильма, не требуется указывать местонахождение переменной `q1answer`.

```
if (q1answer == 3){
```

Ключевое слово *if* сообщает интерпретатору, что мы собираемся передать ему список инструкций, которые должны быть выполнены, только если удовлетворено некоторое условие. Суть этого условия описывается в круглых скобках за ключевым словом *if*: (*q1answer == 3*), а открывающаяся фигурная скобка начинает список условно выполняемых инструкций. Поэтому строка 2 означает: «Если значение `q1answer` равно 3, то выполнить инструкции, заключенные в фигурные скобки».

Но как в точности действует условие *q1answer == 3*? Давайте разобьем эту конструкцию на части. Мы узнаем в `q1answer` переменную, в которой мы сохранили ответ пользователя на вопрос 1. Число 3 означает правильный ответ на вопрос 1, потому что клипы впервые появились во Flash 3. Удвоенный знак равенства (==) между переменной и числом 3 является оператором сравнения *равенства*, который сравнивает два выражения. Если выражение в левой части (`q1answer`) равно выражению в правой части (3), то наше условие выполнено, и выполняются инструкции в фигурных скобках. Если нет, то условие не выполнено, и инструкции в фигурных скобках пропускаются.

Flash не может знать о том, какие ответы на наши вопросы являются правильными. Проверка переменной `q1answer` на равенство 3 служит способом сообщить Flash о необходимости проверки правильности ответа пользователя на вопрос 1. Если ответ правильный, то Flash должен добавить единицу к своему суммарному счету:

```
totalCorrect = totalCorrect + 1;
```

В строке 3 сказано: «Сделать новое значение totalCorrect равным прежнему значению totalCorrect плюс один» (то есть *инкрементировать* totalCorrect). Инкрементирование переменной встречается столь часто, что для него есть специальный собственный оператор, ++.

Поэтому вместо кода:

```
totalCorrect = totalCorrect + 1;
```

мы обычно пишем:

```
totalCorrect++;
```

что делает то же самое, но записывается короче.

В строке 4 фигурная скобка завершает блок инструкций, подлежащих выполнению, если удовлетворено наше первое условие:

```
}
```

Строки с 5 по 7 представляют собой другую условную инструкцию:

```
if (q2answer == 2){
  totalCorrect++;
}
```

Здесь мы проверяем, правильно ли пользователь ответил на вопрос 2 (поддержка MP3-аудио впервые появилась во Flash 4). Если пользователь выбрал второй ответ, мы добавляем единицу к totalCorrect с помощью оператора инкрементирования ++.

Так как в нашем вопроснике есть только два вопроса, то подсчет баллов пользователя на этом завершен. Для каждого вопроса, на который дан правильный ответ, мы добавляли единицу к totalCorrect, поэтому totalCorrect теперь содержит окончательный результат пользователя. Осталось только показать пользователю его результат в экранном текстовом поле. Flash MX позволяет создать текстовое поле непосредственно кодом:

```
this.createTextField("totalOutput_txt", 1, 150, 200, 200, 20);
```

Ключевое слово this относится здесь к главной временной диаграмме фильма, где должно появиться новое текстовое поле. Затем мы выполняем *createTextField( )*, что сообщает Flash о необходимости поместить новое поле с именем totalOutput_txt на глубину 1 с координатой x, равной 150, и координатой y, равной 200, установив для него ширину 200 и высоту 20. Мы можем показать значение totalCorrect в новом текстовом поле следующим образом:

```
totalOutput_txt.text = "Your final score is: " + totalCorrect + "/2";
```

Новичку в программировании не обязательно до конца понимать, как работает этот код. Но код должен дать представление о том, какие замечательные вещи можно делать с помощью ActionScript. Попробуйте самостоятельно изменить, скажем, ширину и высоту текстового поля или текст, который в нем отображается. Текстовые поля исчерпывающе рассматриваются в описании класса *TextField* в «Справочнике по языку».

Тестирование нашего вопросника

Ну, вот и все. Наш вопросник завершен. Теперь вы можете проверить, работает ли вопросник, с помощью меню Control → Test Movie. Пощелкайте по кнопкам ответов в разных сочетаниях, чтобы убедиться в правильности подсчета очков. Можно даже создать кнопку перезапуска, создав экземпляр кнопки с именем `restart_btn` и поместив его в кадр 1 слоя *scripts*:

```
restart_btn.onRelease = function () {
  this._parent.totalOutput_txt.removeTextField();
  this._parent.gotoAndStop("init");
}
```

Так как `totalCorrect` устанавливается в 0 в коде кадра `init`, счет будет сбрасываться каждый раз, когда вы начинаете воспроизведение с `init`.

Обнаружив, что вопросник не работает, попробуйте сравнить его с образцом вопросника, имеющимся в сетевом хранилище кода.

Вперед!

Ну как? Вы выучили несколько фраз, немного грамматики, некоторые слова и даже имели продолжительный разговор с Flash (вопросник с вариантами ответов). Я бы сказал, весьма насыщенный первый день обучения иностранному языку.

Как видите, в ActionScript надо многое изучать, но и при минимуме знаний можно сделать достаточно много. Даже то, что вам известно сейчас, открывает массу возможностей. В оставшейся части книги мы упрочим наши знания основ, изучая их более глубоко и показывая в сочетании с практическими примерами. Конечно, мы расскажем также о некоторых темах, о которых еще и речи не шло.

Запомните: общение, сотрудничество и ясность мысли. Если вы обнаружите, что делаете что-то чрезвычайно интересное и вам хочется поделиться информацией с другими, посылайте ее мне на адрес *http://www.moock.org/contact/*.

Теперь, получив практический опыт, вы сможете лучше понять и запомнить базовые знания, подробно излагаемые в нескольких следующих главах. Они дадут вам возможность глубже понять ActionScript и создавать более сложные фильмы.

2

Переменные

В типичном фильме, управляемом сценариями, приходится отслеживать и обрабатывать все – от номеров кадров до пароля пользователя и скорости фотонной торпеды, которой стреляет космический корабль. Для управления всей этой информацией и ее извлечения надо запоминать данные в *переменных* (*variables*), главных контейнерах для хранения информации в ActionScript.

Переменная подобна банковскому счету, но хранит не деньги, а информацию (*данные*). Создание новой переменной подобно открытию нового счета: мы определяем место для хранения чего-то, что может понадобиться нам в будущем. И так же как у каждого счета есть свой номер, у каждой переменной есть имя, которое используется для доступа к находящимся в ней данным.

Создав переменную, в нее можно поместить новые данные сразу, как только это потребуется, как деньги на счет. Можно также узнать, что находится в переменной, указав ее имя, – это напоминает проверку состояния счета. Если переменная нам больше не нужна, можно «закрыть счет», удалив переменную.

Надо отметить главное – возможность обращаться с помощью переменных к данным, изменяющимся или вычисляемым во время воспроизведения фильма. Так же как номер банковского счета остается одним и тем же, даже если изменяется остаток на нем, имя переменной постоянно, даже если содержащиеся в ней данные изменяются. Такие постоянные ссылки для доступа к переменному содержимому позволяют выполнять сложные вычисления, следить за картами в карточной игре, сохранять записи в гостевой книге или направлять воспроизводящую головку в различные точки в зависимости от меняющихся условий.

Заблестел ли восторг в ваших глазах? Отлично, а то я боялся, что вы потеряли интерес к теме за время всех этих разговоров о банках. Начнем наше изучение переменных с того, что посмотрим, как они создаются.

Создание переменных (объявление)

Создание переменной называется ее *объявлением* (*declaration*). Объявление – это, пользуясь нашей банковской метафорой, этап «открытия счета», то есть

этап формального создания переменной. Впервые объявленная переменная пуста – это чистая страница, ожидающая заполнения. В таком состоянии переменная содержит особое значение, называемое `undefined` (указывающее на отсутствие данных).

Для объявления новой переменной используется инструкция *var*. Например:

```
var speed;
var bookTitle;
var x;
```

Слово *var* говорит интерпретатору, что мы объявляем переменную, а последующий текст, например `speed`, `bookTitle` или `x`, становится именем новой переменной. Создавать переменные можно везде, где допускается прикрепление кода: в ключевом кадре, кнопке или клипе.

При вводе кода в окне Actions, когда выбранный кадр не является ключевым, код прикрепляется к ближайшему предшествующему ключевому кадру.

С помощью одной инструкции *var* можно объявить несколько переменных, например:

```
var x, y, z;
```

Однако в этом случае мы не сможем поместить комментарии рядом с каждой переменной.

Создав переменную, можно присвоить ей значение, но прежде чем узнать, как это делается, рассмотрим некоторые более тонкие детали объявления переменной.

Автоматическое создание переменных

Многие языки программирования требуют, чтобы переменные были объявлены прежде, чем в них будут помещены данные, в противном случае возникает ошибка. ActionScript не так строг. Если присваивать значение несуществующей переменной, то интерпретатор создаст ее сам. Если продолжить аналогию, то банк автоматически откроет счет при попытке сделать первый вклад.

Такое удобство обходится, однако, не бесплатно. Если мы не объявим переменные сами, то у нас не будет их общего перечня, удобного при изучении кода. Более того, прямое объявление переменной с помощью инструкции *var* может иногда давать результаты, отличные от тех, которые получаются при *неявном* (автоматическом) объявлении переменной. Самое надежное – сначала объявить переменную, а затем ее использовать (*явное объявление*), как и делается на протяжении этой книги.

Допустимые имена переменных

Прежде чем начать создавать переменные, запомните, что имена переменных:

- должны состоять исключительно из букв, цифр, знаков доллара ($) и символов подчеркивания (пробелы, дефисы и символы пунктуации не допустимы);
- должны начинаться с буквы или символа подчеркивания (например, `_someVar`) либо со знака доллара (например, `$someVar`);
- не должны превышать в длину 255 символов (ладно, это неправда, но пересмотрите свою систему именования переменных, если длина их имен превосходит 255 символов);
- нечувствительны к регистру (верхний и нижний регистры воспринимаются одинаково, но несмотря на это следует соблюдать единообразие).

Вот допустимые имена переменных:

```
var first_name;
var counter;
var reallyLongVariableName;
```

Вот недопустимые имена переменных, которые вызовут ошибку:

```
var 1first_name;                // Начинается с цифры
var variable name with spaces;  // Содержит пробелы
var another-illegal-name;       // Содержит дефис
```

Следуя хорошему стилю программирования, стоит добавлять к именам переменных суффиксы, говорящие о типе данных, которые они содержат:

```
var firstName_str;              // _str означает, что переменная содержит текст
var variable name with spaces;  // _array означает, что переменная содержит массив
```

Во Flash MX некоторые суффиксы также активизируют подсказку кода (*code hinting*) в окне Actions. Например, суффикс `_txt` в имени переменной `output_txt` не только указывает на то, что в переменной хранится текстовое поле, но также вызывает при вводе имени этой переменной в окне Actions показ всплывающей подсказки по текстовым полям (так называемый *code hint*). В табл. 2.1 перечислены встроенные суффиксы, активизирующие подсказку кода во Flash MX.

Таблица 2.1. Суффиксы для подсказки кода во Flash MX

Суффикс	Тип представляемых объектов
`_mc`	*MovieClip*
`_array`	*Array*
`_str`	*String*
`_btn`	*Button*
`_txt`	*TextField*
`_fmt`	*TextFormat*

Суффикс	Тип представляемых объектов
_date	*Date*
_sound	*Sound*
_xml	*XML*
_xmlsocket	*XMLSocket*
_color	*Color*
_video	*Video*
_ch	*FcheckBox**
_pb	*FpushButton**
_rb	*FradioButton**
_lb	*FListBox**
_sb	*FScrollBar**
_cb	*FcomboBox**
_sp	*FscrollPane**

* Компоненты Flash UI (см. приложение G).

В индустрии программирования широко распространена система именования переменных, альтернативная той, которую предлагает Flash. Она известна как *венгерская нотация* и разработана инженером венгерского отделения Microsoft Чарльзом Симони (Charles Simonyi). Информацию о ней можно получить на странице:

http://msdn.microsoft.com/library/techart/hunganotat.htm

Достоинства и недостатки венгерской нотации обсуждаются по адресу:

http://ootips.org/hungarian-notation.html

Создание переменных с динамическими именами

Хотя вы едва ли будете когда-нибудь использовать динамически создаваемые имена переменных, следует знать, что имена переменных можно генерировать программно. Этот прием применялся в основном до того, как во Flash 5 появились массивы, поэтому его можно встретить в старом коде, предназначенном для Flash версии 4 или более ранних. Чтобы создать имя переменной из любого выражения, применяется инструкция *set*. Например, в следующем коде мы присваиваем значение «bruce» переменной `player1name`:

```
var i = 1;
set ("player" + i + "name", "bruce");
```

Массивы и объекты, о которых рассказывается в последующих главах, следует предпочесть, поскольку они предоставляют более мощные средства хранения данных с динамическими именами.

Объявление переменных в начале

Хорошим стилем считается объявление переменных приложения в начале главной области сценариев каждого фильма, то есть обычно в первом ключевом кадре, следующем за предзагрузчиком фильма (preloader). Добавьте комментарии, поясняющие назначение каждой переменной, чтобы облегчить чтение кода в дальнейшем. Начало хорошо структурированного сценария может выглядеть так:

```
// ^^^^^^^^^^^^^^^^^^^^^^^^^
// Инициализация переменных
// ^^^^^^^^^^^^^^^^^^^^^^^^^
var ballSpeed;   // Скорость шарика, максимум 10
var score;       // Текущий счет игрока
var hiScore;     // Высший результат (не сохраняемый между сеансами)
var player1;     // Имя игрока 1, сообщенное пользователем
```

Одновременно с созданием переменных можно дать им начальные значения, например:

```
var ballSpeed = 5;    // Скорость шарика, по умолчанию 5, максимум 10
var score = 0;        // Текущий счет игрока
var hiScore = 0;      // Высший результат (не сохраняемый между сеансами)
var player1 = "1P"    // Имя игрока по умолчанию 1P
```

В объектно-ориентированных программах для управления переменными лучше всего хранить их в классе, как описывается в главе 12.

Присваивание значений переменным

Теперь переходим к самому интересному: размещению данных в переменных. Если вы еще помните аналогию с банком, то это этап «внесения средств на счет». Для присваивания значения переменной запишем:

```
variableName = value;
```

где *variableName* — имя переменной, а *value* — данные, присваиваемые этой переменной. Приведем пример:

```
bookTitle = "ActionScript: The Definitive Guide";
```

Слева от знака равенства находится слово bookTitle, представляющее собой *имя* переменной (ее *идентификатор*). Справа от знака равенства расположена фраза «ActionScript: The Definitive Guide», представляющая собой *значение* переменной (размещаемые данные). Сам знак равенства называется оператором *присваивания* (*assignment*). Он сообщает Flash, что значение справа от знака равенства (вклад) вы хотите присвоить переменной, указанной слева. Если переменная слева еще не существует, то Flash создаст ее (хотя полагаться на интерпретатор в создании переменных не рекомендуется).

Вот еще два примера присваивания значений переменным:

```
speed = 25;
output = "thank you";
```

В первом примере переменной speed присваивается значение 25, что демонстрирует возможность хранения в переменных не только текста, но и чисел. Вскоре мы увидим, что они могут содержать и другие типы данных. Во втором примере текст «thank you» присваивается переменной output. Обратите внимание, что в ActionScript для ограничения текстовых строк используются вертикальные двойные кавычки (" ").

Теперь рассмотрим несколько более сложный пример, в котором y получает значение выражения 1 + 5:

```
y = 1 + 5;
```

При выполнении предложения *y = 1 + 5;* сначала 1 складывается с 5, что дает 6, а затем 6 присваивается переменной y. Выражение справа от знака равенства *вычисляется* (*evaluated*, calculated или resolved) до присваивания результата переменной в левой части. Ниже мы присваиваем выражение, содержащее переменную y, другой переменной, z:

```
z = y + 4;
```

Повторюсь, вычисляется выражение справа от знака равенства, а результат присваивается переменной z. Интерпретатор извлекает текущее значение y (так сказать, проверяет состояние этого счета) и прибавляет к нему 4. Так как y имеет значение 6, то z получит значение 10.

Синтаксис присваивания любых данных переменным любого типа одинаков (будь то числа, текст или данные любого другого типа). Например, мы еще не изучали массивы, но вы должны уже определить, что следующий код является инструкцией присваивания значения переменной:

```
myList = ["John", "Joyce", "Sharon", "Rick", "Megan"];
```

Как и раньше, мы помещаем имя переменной слева, оператор присваивания (знак равенства) в середину, а новое значение, присваиваемое переменной, – справа.

Для того чтобы быстро присвоить одно и то же значение нескольким переменным, можно расположить присваивания одно за другим:

```
x = y = z = 10;
```

Присваивание переменным всегда выполняется справа налево. Предыдущая инструкция присваивает z значение 10, затем присваивает y значение z и, наконец, присваивает x значение y.

Не путайте знак равенства, применяемый для присваивания значения переменной, с алгебраическим знаком равенства, знакомым вам по урокам математики. Более того, не путайте знак равенства (=), являющийся оператором присваивания, со знаком двойного равенства (==), применяемым для логического сравнения двух выражений.

В алгебре следующее выражение не имеет смысла, поскольку нечто не может быть равно самому себе плюс единица:

```
x = x +1;
```

Но в языках программирования такая инструкция совершенно допустима и даже часто встречается. Она говорит о том, что надо взять прежнее значение x, добавить к нему 1 и записать новое значение обратно в переменную x. В данном случае, если прежнее значение x было 4, то эта инструкция изменит x на 5.

Следующая инструкция обычно некорректна, потому что она изменяет значение x на 5:

```
if (x = 5) { //что-то выполнить};
```

Скорее всего, программист собирался сравнить текущее значение x с 5:

```
if (x == 5) { //что-то выполнить};
```

Применение оператора сравнения (==) подробно рассмотрено в главе 5.

Изменение и извлечение значений переменных

Создав переменную, можно при желании присваивать ей новые и новые значения, как показано в примере 2.1.

Пример 2.1. Изменение значений переменных

```
var firstName;                    // Объявить переменную firstName
firstName = "Graham";             // Присвоить значение firstName
firstName = "Gillian";            // Изменить значение firstName
firstName = "Jessica";            // Снова изменить firstName
firstName = "James";              // Снова изменить firstName
var x = 10;                       // Объявить x и присвоить числовое значение
x = "loading...please wait...";   // Присвоить x текстовое значение
```

Обратите внимание, что мы изменили *тип данных* переменной x с числового на текстовый, просто присвоив ей значение требуемого типа. Некоторые языки программирования не позволяют изменять тип данных переменной, но в ActionScript это делать можно.

Конечно, создавать переменные и присваивать им значения бесполезно, если позднее эти значения нельзя извлечь. Для того чтобы извлечь значение переменной, достаточно указать ее имя там, где, по вашему мнению, должно быть ее значение. Везде, где появляется имя переменной (за исключением объявления и левой части оператора присваивания), оно заменяется ее значением.

Вот несколько примеров. Обратите внимание, что _x является встроенным свойством, представляющим расположение клипа по горизонтали и не имеющим никакого отношения к переменной x в предыдущих примерах (см. врезку «Стиль именования переменных» в главе 1 и описание *MovieClip._x* в «Справочнике по языку»).

```
newX = oldX + 5;    // Установить newX равным значению oldX плюс 5
ball._x = newX;     // Установить позицию по горизонтали клипа ball равной значению newX
trace(firstName); // Вывести значение firstName в окно Output
```

Обратите внимание, что `ball` в выражении `ball._x` представляет собой имя клипа, а `_x` задает свойство его координаты x (горизонтального положения на сцене). Подробнее о свойствах мы поговорим позднее. Последняя строка, *trace(`firstName`)*, выводит значение переменной в окно Output во время выполнения сценария, что удобно для отладки кода.

Проверка наличия значения у переменной

Иногда прежде чем ссылаться на переменную, желательно проверить, присваивалось ли ей значение. Как мы уже знаем, объявленная переменная, которой не присваивалось значение, содержит особое «неопределенное значение» `undefined`. Чтобы определить, присваивалось ли переменной значение, мы сравним тип ее значения с типом данных *undefined*, например:

```
if (typeof someVariable != "undefined") {
  // Любой расположенный здесь код выполняется,
  // только если someVariable не является undefined
}
```

Обратите внимание на *оператор неравенства* `!=`, который определяет, что два значения *не* равны, и кавычки, в которые заключается «undefined», поскольку оператор *typeof* возвращает строку. Во Flash MX есть другая возможность проверить, не равно ли значение `undefined` (не имеет ли оно тип *undefined*) – использовать оператор строгого неравенства (`!==`).

```
if (someVariable !== undefined) {
  // Любой расположенный здесь код выполняется,
  // только если someVariable не является undefined
}
```

Оба показанных приема – *typeof* и строгое неравенство – предотвращают во Flash автоматическое преобразование типа данных при проверке переменной *`someVariable`* на неопределенность. Напротив, благодаря автоматическому преобразованию типа данных обычный оператор неравенства (`!=`) проверяет не только равенство *`someVariable`* значению `undefined`, но и равенство значению `null`. Например:

```
if (someVariable != undefined) {
  // Любой расположенный здесь код выполняется,
  // только если someVariable не является undefined или null
}
```

С помощью значения `null` программисты часто показывают, что переменная сделана пустой умышленно, но все же существует.

Типы значений

Используемые в программировании ActionScript данные могут иметь разные типы. До сих пор мы видели числа и текст, а другие типы включают в себя булевы значения, массивы, функции и объекты. Прежде чем подробно рассказывать о каждом типе данных, обсудим некоторые проблемы типов данных, имеющих особое отношение к использованию переменных.

Автоматическая типизация

Любая переменная ActionScript может содержать данные любого типа. Это может показаться малопримечательным, но способность хранить данные *любого* типа в *любой* переменной на самом деле не совсем обычна. Переменные в таких языках, как C++ и Java, *строго типизированы*: каждая переменная может принимать данные только одного типа, который должен быть указан при ее объявлении. Переменные ActionScript получают тип *автоматически* – интерпретатор назначает тип переменной, присваивая ей значение.

Переменные ActionScript не только могут содержать данные любого типа, но также могут динамически *изменять* тип данных. Когда переменной присваивается значение иного типа, чем у предшествующего значения переменной, тип переменной автоматически изменяется. Поэтому в ActionScript допустим такой сценарий:

```
x = 1;                  // x является числом
x = "Michael";          // x является теперь строкой
x = [4, 6, "hello"];    // x является теперь массивом
x = 2;                  // x снова число
```

В языках, которые не поддерживают автоматическое изменение типа, таких как C++ или Java, данные неверного типа преобразуются к тому типу, который имеется у переменной (а если преобразование осуществить нельзя, происходит ошибка). Для сравнения, VB.NET-программист имеет возможность решить, должен ли компилятор осуществлять строгую проверку типов. Строгая типизация может показаться обременительной, но она помогает избежать ошибок, с которыми может неожиданно столкнуться программист из-за автоматического и динамического преобразования типов, которое будет рассмотрено в следующих разделах.

Автоматическое преобразование типа значения

В некоторых контекстах ActionScript ожидает данные конкретного типа. Если мы используем переменную, тип значения которой не соответствует ожидаемому, то интерпретатор пытается преобразовать данные к нужному типу. Например, если мы используем текстовую переменную там, где требуется число, интерпретатор попытается преобразовать текстовое значение переменной в числовое, необходимое для текущей операции. В примере 2.2 переменная `z` устанавливается равной 2. Почему? Поскольку оператор вычитания ждет число, значение `y` преобразуется из строки «4» в число 4, которое вычитается из 6 (значения `x`), что дает в результате 2.

Пример 2.2. Автоматическое преобразование строки в число

```
x = 6;      // x - число 6
y = "4";    // y - строка "4"
z = x - y;  // Устанавливает значением z число 2
```

И наоборот, если числовая переменная используется там, где ожидается строка, то интерпретатор пытается преобразовать число в строку. В примере 2.3

значением z становится строка «64», а не число 10. Почему? Потому что второй операнд в выражении x + y является строкой. В результате оператор + осуществляет конкатенацию строк, а не математическое сложение. Значение x (6) преобразуется в строку «6», а затем объединяется со строкой «4» (значением y), результатом чего и является «64».

Пример 2.3. Автоматическое преобразование числа в строку

```
x = 6;       // x - число 6
y = "4";     // y - строка "4"
z = x + y;   // Устанавливает значением z строку "64"
```

Автоматическое преобразование типа, выполняемое при вычислении переменной как части выражения, осуществляется над *копией* данных переменной – оно не влияет на тип исходной переменной. Тип переменной изменяется только при присваивании ей значения, тип которого не соответствует типу предшествующего значения. Поэтому в конце примеров 2.2 и 2.3 y остается строкой, а x остается числом.

Обратите внимание, что оператор в строке 3 («-» в примере 2.2, «+» в примере 2.3) существенно влияет на значение, присваиваемое z. В примере 2.2 строка «4» становится числом 4, тогда как в примере 2.3 происходит противоположное (число 6 становится строкой «6»), поскольку правила преобразования типов для операторов + и - различны. О правилах преобразования данных мы расскажем в главе 3, а об операторах – в главе 5.

Определение типа данных вручную

Автоматическое определение и преобразование типа данных удобно, но, как показывают примеры 2.2 и 2.3, может приводить к непредсказуемым результатам. До выполнения команд, оперирующих смешанными типами данных, может оказаться полезным определить тип данных переменной с помощью оператора *typeof*:

```
productName = "Macromedia Flash";  // Строковое значение
trace(typeof productName);         // Выводит: "string"
```

Если известен тип данных переменной, можно воспользоваться условными инструкциями. Здесь, например, прежде чем продолжить работу, мы проверяем, является ли переменная числом:

```
if (typeof age == "number"){
  // Можно продолжить
} else {
  trace ("Age isn't a number");  // Выввести сообщение об ошибке
}
```

Полное описание оператора *typeof* представлено в главе 5.

Область видимости переменной

Ранее мы узнали, как создавать переменные и получать их значения. Но все наши переменные прикреплялись к отдельному кадру главной временной

диаграммы документа Flash. Если документ содержит несколько кадров и временных диаграмм клипов, создание переменных и извлечение их значений становится несколько более сложным.

Переменные временной диаграммы

Чтобы проиллюстрировать использование переменных временной диаграммы, рассмотрим два простых сценария.

Сценарий 1: доступ к значению, определенному ранее на той же временной диаграмме

Что произойдет, если мы определим переменную в некотором кадре временной диаграммы и позднее попытаемся к ней обратиться?

Предположим, что мы создаем переменную `x` в кадре 1 главной временной диаграммы. После создания переменной `x` мы устанавливаем ее значение равным 10:

```
var x;
x = 10;
```

Затем прикрепляем к следующему кадру (кадр 2) следующий код:

```
trace(x);
```

Если мы воспроизведем этот фильм, появится ли что-либо в окне Output? Мы создали переменную в кадре 1, а пытаемся извлечь ее значение в кадре 2. Существует ли там наша переменная? Да. В окне Output появится число 10.

При определении переменной во временной диаграмме она становится доступной из всех других сценариев в ее кадрах.

Сценарий 2: доступ к значению, определяемому позднее на той же временной диаграмме

Что произойдет, если мы попытаемся обратиться к переменной, значение которой устанавливается в более позднем кадре временной диаграммы?

Допустим, что в кадр 30 главной временной диаграммы мы помещаем следующий код:

```
password = "let_me_in";
gotoAndStop(15);
```

А в кадр 15 такой:

```
trace(password);
```

Тогда при воспроизведении фильма в окне Output будет показано:

```
undefined
let_me_in
```

Это объясняется следующим. Когда воспроизводящая головка достигает кадра 15, Flash показывает значение `password` в окне Output. В этот момент `password` не существует, поэтому Flash показывает `undefined`. Однако когда воспроизводящая головка достигает кадра 30, `password` создается и получает значение «let_me_in». После этого воспроизводящая головка направляется снова в кадр 15 (*gotoAndStop(15)*), вызывая новое выполнение кода кадра 15. На этот раз, хотя *password* определяется в более позднем кадре, чем команда *trace(password)*, она все же принадлежит той же самой временной диаграмме и у нее есть значение, поэтому Flash показывает его в окне Output.

Любая переменная, объявленная на временной диаграмме, доступна для всех сценариев своей временной диаграммы, пока существует эта временная диаграмма. Однако значение переменной остается неопределенным, пока не будет выполнен тот сценарий, в котором ей присваивается значение. Когда значение переменной установлено, оно сохраняется, даже если головка воспроизведения перемещается по временной диаграмме назад. Иными словами, значение переменной определяется порядком выполнения сценариев, который зависит от перемещения головки, а не от расположения кадров, к которым прикреплены сценарии, на временной диаграмме.

Доступность переменных (область видимости)

Представленные выше два сценария иллюстрируют проблемы *области видимости* (*scope*). Область видимости переменной означает, где и когда эта переменная может обрабатываться кодом в фильме. Область видимости переменной определяет продолжительность ее существования и доступность из других блоков кода в сценариях. Чтобы определить область видимости переменной, мы должны ответить на два вопроса: а) в течение какого времени существует переменная и б) в каких частях нашего кода можно извлекать и устанавливать значения переменной?

В традиционном программировании переменные часто разбивают на две общие категории видимости: *глобальную* и *локальную*. Переменные, которые доступны на протяжении всей программы, называются *глобальными переменными*. Переменные, доступные только в некоторых частях программы, называются локальными. В дополнение к двум обычным типам переменных Flash вводит *переменные временной диаграммы*, область видимости которых ограничивается отдельными экземплярами клипов. Flash 5 поддерживал обычные локальные переменные, но не поддерживал подлинно глобальные переменные. Во Flash MX добавлена поддержка подлинно глобальных переменных.

Во Flash все переменные относятся к одной из следующих областей видимости:

- Функция (локальная переменная)
- Главная временная диаграмма или клип (переменная временной диаграммы)
- Во Flash MX – объект `_global` (глобальная переменная)

Переменные клипов и глобальные переменные

Как мы видели в двух предшествующих сценариях, переменная, определенная на временной диаграмме, доступна во всех сценариях на этой временной диаграмме – от первого до последнего кадра – независимо от того, объявлена она в кадре или в кнопке. Но что если в фильме не одна временная диаграмма, как в сценарии 3?

Сценарий 3: переменные, определенные в разных временных диаграммах

Предположим, что есть две базовые геометрические фигуры, квадрат и круг, помещенные в отдельные клипы. Вспомним, что каждый клип снабжен собственной временной диаграммой.

В кадре 1 клипа с *квадратом* присвоим переменной `x` значение 3:

```
var x;
x = 3;
```

В кадре 1 клипа с *кругом* присвоим переменной `y` значение 4:

```
var y;
y = 4;
```

Экземпляры этих клипов мы поместим в кадр 1 слоя 1 главной временной диаграммы и назовем наши экземпляры `square` и `circle`.

Первый вопрос: если следующий код прикрепить к кадру 1 *главной* временной диаграммы фильма (в которую помещены `square` и `circle`), что появится в окне Output при воспроизведении фильма? Вот этот код:

```
trace(x);
trace(y);
```

Ответ:

```
undefined
undefined
```

Переменная `x` определена во временной диаграмме `square`, а переменная `y` – во временной диаграмме `circle`; ни та ни другая не определены на главной временной диаграмме. Поскольку Flash не может найти на главной временной диаграмме переменные с именами `x` или `y`, он выводит в окне Output значение `undefined`.

Для переменных, прикрепленных к временной диаграмме клипа (например, `square` или `circle`), область видимости ограничена этой диаграммой. Они не доступны непосредственно сценариям в других временных диаграммах, таких как наша главная временная диаграмма фильма.

Второй вопрос: если мы поместим инструкции *trace(x)* и *trace(y)* в кадр 1 клипа `square` вместо кадра 1 главной временной диаграммы, то что появится в окне Output?

Ответ: значение x, которое равно 3, и значение y, undefined.

```
3
undefined
```

Значение x отображается равным 3, потому что x определена на временной диаграмме square и, следовательно, доступна команде *trace( )*, расположенной на той же временной диаграмме. Но значение y не появляется в окне Output, потому что y определена в circle, а это отдельная временная диаграмма.

Во Flash переменная, прикрепленная к отдельной временной диаграмме (например, к любому клипу), непосредственно доступна только сценариям на той же временной диаграмме. Кроме того, два или более клипа могут законным образом определять переменные с одинаковыми именами. Например, можно создать переменную x в circle со значением 25, хотя уже существует переменная x в square со значением 3. Результатом выполнения *trace(x)* из square будет 3, а из circle – 25.

Переменные, прикрепленные к временным диаграммам, будем называть *переменными временных диаграмм* или *переменными клипов*. Теперь создадим *глобальную переменную*, которая непосредственно доступна всем сценариям фильма (на любых временных диаграммах). Следующий код мы поместим в кадр 1 временной диаграммы square, хотя тот же результат этот код даст, будучи помещен в любой клип:

```
_global.day = "Monday";
```

Теперь поместим следующий код в кадр 1 главной временной диаграммы:

```
trace(day);
```

Поскольку переменная day глобальна, Flash найдет ее на главной временной диаграмме, хотя она была создана в square, и выведет ее значение в окно Output:

```
Monday
```

Как мы увидим ниже в разделе «Цепочка областей видимости», если у переменной временной диаграммы и глобальной переменной одинаковые имена, приоритет отдается переменной временной диаграммы. Например, если мы создадим переменную временной диаграммы day внутри клипа circle:

```
var day = "Friday";
```

и проверим значение day, находясь в circle:

```
trace(day);
```

то окно Output покажет «Friday» (значение переменной временной диаграммы), а не «Monday» (значение глобальной переменной). Подробнее о глобальных переменных рассказано в описании _global в «Справочнике по языку».

Хотя во Flash 5 не поддерживаются подлинно глобальные переменные, там есть возможность эмулировать их с помощью класса *Object* или класса *MovieClip*. Чтобы создать во Flash 5 переменную, доступную на всех временных диаграммах, применяется следующая конструкция:

```
MovieClip.prototype.myGlobalVariable = myValue;
```

Например:

```
MovieClip.prototype.msg = "Hello world";
```

Другая возможность состоит в том, чтобы назначить переменную свойством *Object*, например:

```
Object.msg = "Hello world";
```

После этого на любой временной диаграмме доступ к `msg` можно получить так:

```
trace(Object.msg);
```

Мы обсудим этот прием (и причины, по которым он действует) в разделе «Конец цепочки наследования» главы 12. Во Flash MX предпочтительнее хранить глобальные переменные в объекте `_global`.

Доступ к переменным других временных диаграмм

Несмотря на то что переменные одной временной диаграммы не доступны непосредственно сценариям других временных диаграмм, доступ к ним может быть получен косвенным образом. Для создания, извлечения и присваивания значений переменным другой временной диаграммы можно использовать *синтаксис с точкой*, стандартный способ обозначений, часто применяемый в объектно-ориентированных языках, таких как Java, C++ и JavaScript. Вот общий вид конструкции, использующей синтаксис с точкой для обращения к переменной на другой временной диаграмме:

```
movieClipInstanceName.variableName
```

Это значит, что к переменной на другой временной диаграмме мы обращаемся по имени клипа, содержащего эту переменную, за которым следует точка, а затем само имя переменной. Например, в приведенном выше сценарии из главной временной диаграммы можно обратиться к переменной `x` в клипе `square` так:

```
square.x
```

А вот пример обращения из главной временной диаграммы к переменной `y` в клипе `circle`:

```
circle.y
```

Ссылка на переменную с указанием местонахождения этой переменной называется *квалифицированной ссылкой*[1] (*qualified reference*). Квалифицированные ссылки сообщают Flash, где он должен искать нужную нам переменную. Неквалифицированные ссылки требуют от Flash делать предположения относительно того, где находится эта переменная, о чем говорится далее в разделе «Цепочка областей видимости». Поэтому всегда лучше иметь дело с квалифицированными ссылками на переменные.

---

[1] В русскоязычной литературе квалифицированные ссылки иногда называют «уточненными», что поясняет их предназначение. – *Примеч. науч. ред.*

Квалифицированные ссылки из главной временной диаграммы фильма можно использовать для присваивания и извлечения значений переменных в square:

```
square.z = 5;        // Присвоить значение 5 переменной z в клипе square
var mainZ;           // Создать переменную mainZ на главной временной диаграмме
mainZ = square.z;    // Присвоить переменной mainZ значение z из клипа square
```

Однако одного синтаксиса *movieClipInstanceName.variableName* недостаточно, чтобы ссылаться на переменные в square из клипа circle. Если мы поместим ссылку square.x в кадр circle, то интерпретатор станет искать клип с именем square ***внутри*** кадра circle, в то время как square находится на главной временной диаграмме. Поэтому необходим механизм, позволяющий ссылаться на временную диаграмму, содержащую клип square (в данном случае главную временную диаграмму), из клипа circle. Такой механизм предоставляется в виде двух специальных свойств: _root (корень) и _parent (родитель).

Свойства _root и _parent

Свойство _root является прямой ссылкой на главную временную диаграмму фильма. Находясь на любой глубине вложенности структуры клипа, можно обращаться к переменным главной временной диаграммы с помощью свойства _root:

```
_root.mainZ       // Обращение к переменной mainZ главной временной диаграммы
_root.firstName   // Обращение к переменной firstName главной временной диаграммы
```

Можно даже объединять ссылку на _root со ссылками на экземпляры клипов, вовлекая в процесс многоуровневую структуру фильма. Например, можно обратиться к переменной x клипа square, который находится на главной временной диаграмме, таким образом:

```
_root.square.x
```

Эта ссылка действует в любом месте фильма независимо от глубины вложенности клипов, потому что ссылка начинается с главной временной диаграммы фильма, _root. Вот еще один пример вложенности, показывающий обращение к переменной area клипа triangle, находящегося на временной диаграмме клипа shapes:

```
_root.shapes.triangle.area
```

Ссылки на переменные, начинающиеся с ключевого слова _root, называются *абсолютными*, или *полностью квалифицированными ссылками*, потому что они описывают положение переменной относительно фиксированной неизменной точки фильма – главной временной диаграммы.

Бывают, однако, случаи, когда надо сослаться на переменные других временных диаграмм, не обращаясь к главной временной диаграмме фильма. Для этого предназначено свойство _parent, ссылающееся на временную диаграмму, на которой лежит текущий экземпляр клипа. Например, из кода, прикрепленного к кадру клипа square, мы можем ссылаться на переменные временной диаграммы, которая содержит square, используя синтаксис:

```
_parent.myVariable
```

Ссылки, начинающиеся с ключевого слова _parent, называются *относительными ссылками*, так как они составлены *относительно* местоположения клипа, в котором находятся.

Предположим, что у нас есть переменная size, определенная на главной временной диаграмме фильма. Поместим экземпляр клипа с именем shapes на главную временную диаграмму, а на временной диаграмме shapes определим переменную color. Кроме того, на временную диаграмму shapes поместим клип с именем triangle, как показано на рис. 2.1.

Рис. 2.1. *Пример иерархии клипов*

Чтобы отобразить значение переменной color (которая находится в клипе shapes) из кода, прикрепленного к временной диаграмме triangle, можно использовать абсолютную ссылку, начинающуюся на главной временной диаграмме:

```
trace(_root.shapes.color);
```

Но это привязывает наш код к главной временной диаграмме. Чтобы сделать код более гибким, можно использовать свойство _parent и создать относительную ссылку:

```
trace(_parent.color);
```

Наш первый подход (основанный на применении _root) действует в нисходящем порядке: он начинается на главной временной диаграмме и опускается вниз по иерархии клипов, пока не достигнет переменной color. Второй подход (использующий _parent) действует в восходящем порядке: он начинает с клипа, содержащего инструкцию *trace( )* (клип triangle), затем поднимается на один уровень структуры клипов, где и находит переменную color.

Свойство _parent можно использовать два раза подряд (разделитель – точка), чтобы подняться по иерархии клипов на два уровня и обратиться к переменной size на главной временной диаграмме. Прикрепим к triangle код, который ссылается на size, находящуюся на главной временной диаграмме:

```
trace(_parent._parent.size);
```

Используя свойство _parent два раза подряд, мы поднимаемся вверх на два уровня, что в данном контексте доставляет нас к главной временной диаграмме фильма.

Выбор подхода к адресации переменных зависит от того, чего мы хотим достичь при помещении экземпляров клипа на разные временные диаграммы. Если требуется, чтобы ссылка на color всегда указывала на переменную color, находящуюся в клипе shapes, то следует выбрать синтаксис с _root. Но если надо, чтобы ссылка на color указывала на разные переменные color в зависимости от того, на какой временной диаграмме лежит данный экземпляр triangle, то следует выбрать синтаксис _parent.

Значение _root изменяется, когда файл *.swf* загружается в клип. Допустим, например, что мы загрузим *myPictures.swf* в клип, находящийся в *container.swf*. Когда *myPictures.swf* выполняется самостоятельно, _root означает его собственную главную временную диаграмму. Но когда *myPictures.swf* выполняется внутри *container.swf*, _root означает главную диаграмму *container.swf*.

Flash MX позволяет легко решить эту проблему с помощью глобальной переменной. Чтобы сохранить целостность ссылки на главную временную диаграмму фильма, даже когда этот фильм загружен в клип, поместите на главную временную диаграмму следующий код:

```
_global.movieNameMainTimeline = this;
```

Например:

```
_global.myPicturesMainTimeline = this;
```

Эта команда создает глобальную ссылку на главную временную диаграмму, что позволяет ссылаться на переменные главной временной диаграммы из любой другой временной диаграммы вот так:

```
myPicturesMainTimeline.variableName
```

вместо

```
_root.variableName
```

Доступ к переменным на различных уровнях документа

Свойство _root ссылается на главную временную диаграмму фильма текущего уровня (то есть текущего документа), но Flash Player может разместить в своем *стеке документов* несколько документов. На главную временную диаграмму любого фильма, загруженного в стек документов плеера, можно ссылаться с помощью _level*n*, где *n* является номером уровня, на котором расположен фильм. Номера уровней начинаются с 0, то есть _level0, level1, _level2, _level3 и т. д. Сведения о загрузке нескольких фильмов можно найти в главе 13. Вот несколько примеров, демонстрирующих обращение к переменным разных уровней:

```
_level1.firstName          // firstName главной временной диаграммы _level1
```

```
_level4.ball.area          // area в клипе ball главной временной диаграммы _level4
_level0.guestBook.email    // email в клипе guestBook временной диаграммы _level0
```

При обращении к переменным временных диаграмм других клипов надо дать имена экзеплярам клипов с помощью поля <Instance Name> в Инспекторе свойств и указать те же имена при ссылке в коде. Не путайте имя символа в библиотеке с именем экземпляра на сцене. Если экземпляры безымянные, код не сможет обращаться к ним по имени. Неименованные экземпляры и ошибки при вводе имени очень часто оказываются источником проблем.

До того как во Flash 5 был введен синтаксис с точкой, во Flash был принят синтаксис в стиле «пути», в котором место точки занимала косая черта. Flash MX по-прежнему поддерживает этот старый синтаксис, но следует пользоваться более современным синтаксисом с точкой, как описано в таблице C.2 приложения C.

Время жизни переменной клипа

Выше мы сказали, что область видимости переменной отвечает на два вопроса: 1) в течение какого времени существует переменная и 2) в каких частях нашего кода можно извлекать и устанавливать значения переменной? Для переменных клипов нам теперь известны факторы, определяющие ответ на второй вопрос. Но мы еще не ответили на первый вопрос. Вернемся к нему сейчас, рассмотрев еще один, последний сценарий использования переменных.

Сценарий 4: срок жизни переменных клипа

Допустим, мы создаем новый фильм с ключевыми кадрами 1, 2 и 3. В кадр 1 мы поместим экземпляр клипа, `ball`. На временной диаграмме `ball` создадим переменную `radius`. Кадр 3 нашей главной временной диаграммы будет пуст (в нем нет экземпляра `ball`).

Из кадра 2 главной временной диаграммы фильма мы можем получить значение `radius` с помощью кода:

```
trace(ball.radius);
```

Вопрос звучит так: если переместить эту строку кода из кадра 2 в кадр 3 главной временной диаграммы, то что появится в окне Output при воспроизведении фильма?

Ответ: `undefined`. Когда клип `ball` исчезает со сцены в кадре 3, то все его переменные тоже удаляются.

Переменные клипа существуют, только пока клип, в котором они находятся, присутствует на сцене. Переменные, определенные на главной временной диаграмме документа Flash, сохраняются на протяжении всего фильма, но теряются, как только документ выгружается из плеера (с помощью функции *unloadMovie( )* или в результате загрузки другого фильма на тот же уровень).

Срок жизни переменных важен при создании сценариев фильмов, содержащих клипы, распределенные по нескольким кадрам в нескольких временных диаграммах. Всегда следите за тем, чтобы клип, к которому вы обращаетесь, присутствовал на временной диаграмме, когда вы собираетесь использовать его переменные.

Локальные переменные

Переменные клипов имеют область видимости, ограниченную клипами, и сохраняются до тех пор, пока существует клип, в котором они определены. Иногда это дольше, чем требуется. Для ситуаций, в которых переменные нужны нам лишь временно, в ActionScript есть переменные с *локальной* областью видимости (*локальные переменные*), срок жизни которых значительно короче, чем у обычных переменных клипов.

Локальные переменные применяются в функциях. Те, кто не работал ранее с функциями, могут пропустить этот раздел и вернуться к нему после прочтения главы 9.

Функции часто используют переменные, которые нужны только внутри них. Допустим, например, что у нас есть функция, отображающая элементы массива:

```
function displayElements(theArray) {
   var counter = 0;
   while(counter < theArray.length) {
      trace("Element " + counter + ": " + theArray[counter]);
      counter++;
   }
}
```

Переменная `counter` нужна для вывода элементов массива, но после этого не используется. Можно было бы оставить ее определение на временной диаграмме, но это нехорошо по двум причинам: а) если `counter` продолжает существовать, она занимает память на протяжении оставшейся части нашего фильма; б) если `counter` доступна вне нашей функции, она может вступить в конфликт с другими переменными с именем `counter`. Поэтому желательно уничтожить `counter` после завершения работы функции *displayElements( )*.

Чтобы вызвать автоматическое удаление `counter` в конце нашей функции, мы определим ее как *локальную переменную*. В отличие от переменных клипов, локальные переменные автоматически помечаются для удаления их из памяти интерпретатором, когда определившая их функция завершает работу.

Чтобы указать на локальность переменной, объявите ее с помощью ключевого слова *var* внутри своей функции, как в предыдущем примере функции *displayElements( )*.

Однако будьте внимательны: когда инструкция *var* помещена *вне* функции, она создает обычную переменную временной диаграммы, а не локальную переменную. Как показано в примере 2.4, расположение инструкции *var* оказывает решающее влияние.

Переменные внутри функции не обязательно должны быть локальными. Внутри функции можно создавать или изменять переменные клипа, если опустить ключевое слово *var*. Если вместо ключевого слова *var* мы просто присваиваем значение переменной, Flash при некоторых условиях рассматривает ее как нелокальную. Рассмотрим такое присваивание переменной внутри функции:

```
function setHeight(){
   height = 10;
}
```

Результат выполнения инструкции `height = 10;` зависит от того, является ли `height` локальной переменной или переменной клипа. Если `height` была ранее определена как локальная переменная (а в нашем случае это не так), то инструкция `height = 10;` просто модифицирует значение локальной переменной. Если локальной переменной с именем `height` нет, как в данном случае, то интерпретатор создает переменную клипа (не локальную) с именем `height` и присваивает ей значение 10. Не будучи локальной переменной, `height` продолжает существовать и после завершения работы функции.

Пример 2.4 иллюстрирует применение локальных и нелокальных переменных.

Пример 2.4. Локальные и нелокальные переменные

```
var x = 5;                        // Новая нелокальная переменная x теперь равна 5
function variableDemo(){
   x = 10;                        // Нелокальная переменная x теперь равна 10
   y = 20;                        // Новая нелокальная переменная y теперь равна 20
   var z = 30;                    // Новая локальная переменная z теперь равна 30
   trace(x + "," + y + "," + z);  // Послать значения переменных в окно Output
}
variableDemo();   // Вызов нашей функции. Выводит: 10,20,30
// Проверим значения x, y и z по завершении работы функции.
trace(x);         // Выводит: 10 (присваивание в нашей функции сохранилось)
trace(y);         // Выводит: 20 (нелокальная переменная y продолжает существовать)
trace(z);         // Выводит "undefined" в Flash MX и ничего в Flash 5
                  // (бывшая локальная переменная z не определена)
```

Обратите внимание, что можно (хотя это создает путаницу и не рекомендуется) иметь в сценарии локальную и нелокальную переменные с одинаковыми именами, но разными областями видимости. В примере 2.5 показан такой случай.

Пример 2.5. Локальная и нелокальная переменные с одинаковыми именами

```
var myColor = "blue";
function hexRed(){
   var myColor = "#FF0000";
   return myColor;
}
trace(hexRed());   // Выводит: #FF0000 (локальную переменную myColor)
trace(myColor);    // Выводит: "blue" (установка локальной переменной
                   // myColor в #FF0000 не повлияла на нелокальную версию)
```

Цепочка областей видимости

Flash обрабатывает каждую неквалифицированную ссылку, встретившуюся в коде, с помощью цепочки *областей видимости* (*scope chain*), представляющей собой иерархию мест (областей видимости), которые надо просматривать для разрешения ссылки на переменную (для того чтобы однозначно найти нужную переменную и получить ее значение). Когда Flash встречает в сценарии или функции переменную, он ищет значение этой переменной в цепочке областей видимости, связанной с этим кодом. Поиск переменной завершается, когда она обнаруживается в каком-либо месте этой цепочки или ее вообще не удается найти (переменная имеет значение `undefined`.) Цепочка областей видимости для сценария кадра (кода, прикрепленного непосредственно к ключевому кадру клипа) содержит две области видимости, которые принято читать в восходящем порядке:

- Глобальный объект (просматривается интерпретатором в последнюю очередь)
- Клип, содержащий этот код (сначала интерпретатор ведет поиск здесь)

Когда к переменной обращается сценарий кадра, Flash ищет ее в текущем клипе. Если переменная не найдена в клипе или его прототипе, Flash ищет глобальную переменную с тем же именем. Если глобальная переменная не найдена, Flash возвращает `undefined`. Если определены две переменные с одинаковыми именами – переменная в клипе и глобальная переменная, Flash извлекает переменную клипа, поскольку она первая в соответствии с порядком поиска в цепочке областей видимости. Следовательно, переменная клипа всегда преобладает над глобальной переменной с тем же именем.

С другой стороны, цепочка областей видимости для кода в функции, не являющейся вложенной, содержит три области видимости (здесь просмотр тоже осуществляется в восходящем порядке):

- Глобальный объект (`_global`)
- Клип, в котором определена функция
- Локальные переменные

Когда обращение к переменной происходит в функции, Flash сначала ищет локальную переменную, определенную внутри функции с помощью инструкции *var*, как показано в примере 2.5. Если локальная переменная не обнаружена, Flash ищет переменную в том клипе, в котором функция была определена. (В главе 9 мы подробнее обсудим эту тему, включая поведение вложенных функций.) Если переменная не найдена в этом клипе (или его прототипе), Flash ищет глобальную переменную с тем же именем. Если и глобальная переменная не найдена, Flash возвращает `undefined`.

С помощью инструкции *with* можно поместить на первое место в цепочку областей видимости любой объект, в котором Flash должен осуществлять поиск переменных. Подробности можно найти в главе 5.

Пока вы не освоились с цепочкой областей видимости, возможно, следует квалифицировать ссылки на имена переменных (например, `myGameMainTime-`

`line.numPlayers` вместо `numPlayers`), чтобы явно сообщить Flash, где искать переменную. Квалифицированные ссылки однозначно определяют переменную, и потому их легче всего понимать программистам любого уровня. См. раздел «Доступ к переменным других временных диаграмм» выше в этой главе.

Не спутайте цепочку областей видимости с *цепочкой прототипов*. Flash применяет цепочку областей видимости для поиска переменных, а цепочку прототипов он использует при поиске свойств и методов. О цепочке прототипов будет рассказано в главе 12.

Обратите внимание, что с технической точки зрения каждая цепочка областей видимости внутренне реализована в виде ряда объектов, используемых интерпретатором для поиска переменных. См. главу 9 данной книги и красноречивое описание области видимости переменных JavaScript, данное Дэвидом Флэнаганом в «JavaScript: The Definitive Guide» (O'Reilly).[1] Не забывайте, однако, что в ActionScript к глобальному и локальному контекстам исполнения JavaScript добавлены клипы.

Область видимости обработчиков событий

Как будет показано в главе 10, у различных типов обработчиков событий во Flash разные области видимости. В табл. 2.2 приведены суммарные сведения об областях видимости обработчиков событий. Для каждого типа обработчика объявление неквалифицированной нелокальной переменной (без инструкции *var*) создает новую переменную в клипе, приведенном в колонке «Цепочка областей видимости» этой таблицы. Более глубоко мы рассмотрим эту тему в главе 10.

Таблица 2.2. Область видимости обработчиков событий

Тип обработчика событий	Цепочка областей видимости
Функция обратного вызова	Глобальный объект Клип, в котором была определена функция Локальные переменные
Метод приемника (listener)	Глобальный объект Клип, в котором была определена функция Локальные переменные
Клип с обработчиком *on( )* или *onClipEvent( )*	Глобальный объект Клип, в котором физически размещен обработчик во время разработки
Кнопка с обработчиком *on( )*	Глобальный объект Клип, на временной диаграмме которого находится кнопка

[1] Д. Флэнаган «JavaScript. Подробное руководство», 4-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2004.

Загрузка внешних переменных

Большинство переменных создаются непосредственно внутри Flash, но часто переменные загружаются из внешнего текстового файла, сценария сервера или веб-страницы. Загрузка внешних переменных не означает простого присваивания существующим переменным значений из внешнего источника. Загрузка внешних переменных фактически создает новые переменные на этапе исполнения. Различные приемы загрузки переменных разъясняются в следующих статьях «Справочника по языку»:

- *LoadVars*
- *loadVariables( )*
- *fscommand( )*

Кроме того, в некоторых броузерах переменные Flash могут устанавливаться на этапе исполнения с помощью JavaScript и следующего синтаксиса:

```
movieObj.SetVariable("/someClip:firstName", "Colin");
```

где `movieObj` – ссылка на объект фильма Flash, встроенного в страницу. Этот прием действует только в следующих броузерах:

- Internet Explorer под Windows
- Netscape 4 под Windows, Macintosh и Linux
- Netscape 6.2 с Flash Player 6.0.40.0 (или выше) под Windows, Macintosh и Linux

Подробности можно найти по адресу:

http://www.moock.org/webdesign/flash/fscommand/

Аналогично переменные в фильме Flash могут создавать теги `<OBJECT>` и `<EMBED>` документа HTML во время начальной загрузки фильма (через параметр `FlashVars` или строку запроса). Переменные, передаваемые фильму Flash через `FlashVars` или строку запроса, должны быть заданы в кодировке URL согласно правилам, описанным для класса *LoadVars* в «Справочнике по языку». Следующий код использует параметр `FlashVars` для создания переменных `authorName` и `bookName` со значениями «Colin Moock» и «ASDG» на главной временной диаграмме фильма с именем *main.swf*:

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
 CODEBASE="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab"
 WIDTH="550"
 HEIGHT="400">
 <PARAM NAME=movie VALUE="main.swf">
 <PARAM NAME=FlashVars VALUE="authorName=Colin+Moock&bookName=ASDG">

 <EMBED src="main.swf"
 FlashVars="authorName=Colin+Moock&bookName=ASDG"
 WIDTH="550"
 HEIGHT="400"
 TYPE="application/x-shockwave-flash"
 PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
```

```
 </EMBED>
</OBJECT>
```

Параметр FlashVars требует наличия Flash Player 6 или выше и может передать во Flash не более 63 Кбайт данных. Для передачи данных из HTML в более старые плееры Flash допишите параметры к URL файла *.swf*, как показано в следующем примере. В этом случае допустимый объем передаваемых данных зависит от веб-сервера.

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
 codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab"
 WIDTH="550"
 HEIGHT="400">
 <PARAM NAME=movie VALUE="main.swf?authorName=Colin+Moock">

 <EMBED src="main.swf?authorName=Colin+Moock"
 WIDTH="550"
 HEIGHT="400"
 TYPE="application/x-shockwave-flash"
 PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
 </EMBED>
</OBJECT>
```

Передаваемые переменные всегда создаются на главной временной диаграмме фильма. Нет прямого способа установить переменные во вложенном клипе с помощью параметра FlashVars или строки запроса. Дополнительные сведения о тегах <OBJECT> и <EMBED> представлены в приложении Н.

Некоторые практические примеры

Мы очень много занимались теоретическими аспектами переменных. Ниже приводится три примера кода, демонстрирующих использование переменных. Пояснения представлены в комментариях.

В примере 2.6 случайным образом выбирается кадр, в который перемещается головка воспроизведения фильма.

Пример 2.6. Отправить головку воспроизведения в случайный кадр текущей временной диаграммы

```
var randomFrame;            // Хранит случайно выбранный номер кадра
var numFrames;              // Хранит общее количество кадров во временной диаграмме
numFrames = this._totalframes;   // Записать свойство_totalframes
                                 // текущего клипа в переменную numFrames
// Выбрать случайный кадр
randomFrame = Math.floor(Math.random() * numFrames + 1);
this.gotoAndStop(randomFrame);   // Переместить головку воспроизведения
                                 // текущего клипа в случайно выбранный кадр
```

В примере 2.7 определяется расстояние между двумя клипами. Рабочий вариант этого примера можно получить из сетевого хранилища кода.

Пример. 2.7. Расчет расстояния между двумя клипами

```
var c;          // Удобная ссылка на объект клипа circle
var s;          // Удобная ссылка на объект клипа square
var deltaX;     // Расстояние по горизонтали между c и s
var deltaY;     // Расстояние по вертикали между c и s
var dist;       // Кратчайшее расстояние между c и s
c = _root.circle;        // Получить ссылку на клип circle
s = _root.square;        // Получить ссылку на клип square
deltaX = c._x - s._x;    // Вычислить расстояние между клипами по горизонтали
deltaY = c._y - s._y;    // Вычислить расстояние между клипами по вертикали
// Расстояние является корнем квадратным из выражения:
// (deltaX в квадрате плюс deltaY в квадрате).
dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));
// Аккуратные ссылки значительно легче читать, чем следующие:
dist = Math.sqrt(((_root.circle._x - _root.square._x) * (_root.circle._x -
_root.square._x)) + ((_root.circle._y - _root.square._y) * (_root.circle._y -
_root.square._y)));
```

В примере 2.8 значения по шкале Фаренгейта преобразуются в значения по шкале Цельсия. Рабочий вариант этого примера с преобразованием в обе стороны можно получить из сетевого хранилища кода.

Пример 2.8. Преобразователь температуры из шкалы Фаренгейта в шкалу Цельсия

```
// Создать переменные
var fahrenheit;           // Температура по Фаренгейту
var result;               // Результат преобразования

// Инициализировать переменные
fahrenheit = 451;         // Задать температуру по Фаренгейту

// Вычислить эквивалентную по Цельсию
result = (fahrenheit - 32) / 1.8;
// Вывести результат
trace (fahrenheit + " degrees Fahrenheit is " + result + " degrees Celsius.");
```

Вперед!

После знакомства с основами хранения информации в переменных и более сложными темами, такими как область видимости переменных, настало время узнать больше о содержимом, хранящемся в переменных: о *данных*. В следующих трех главах мы узнаем о том, что такое данные, как можно их обрабатывать и почему они являются существенной частью почти всего, что мы создаем с помощью ActionScript.

3

Данные и типы данных

Работая со значениями переменных в главе 2 «Переменные», мы уже сталкивались с данными, то есть с информацией, которая обрабатывается в сценариях. В этой главе мы более глубоко исследуем данные и узнаем, как ActionScript определяет, распределяет по категориям и хранит данные. Кроме того, мы увидим, как создавать и классифицировать данные.

Данные и информация

В самом широком смысле *данные* (*data*) охватывают все, что может храниться в компьютере, от слов и чисел до изображений, видео и звуков. Все данные хранятся в компьютере как последовательность единиц и нулей, которую вы могли видеть в рекламе высоких технологий:

```
0101010101010101101010110110101010101010100000101010101011010101010
1010101010101010111010101010101010101010101010101011111010101010101010101
0101010101010101010101010111010101010101010101010101010101010101010101010
```

Данные – это информация в необработанном состоянии, сырая и бессмысленная. Смысл информации придает семантика. Рассмотрим, например, число 8008898969. В исходном виде оно не слишком осмысленно, но если классифицировать его семантически как телефонный номер (800) 889-8969, то данные становятся полезной информацией.

В этой главе будет показано, как придать смысл необработанным компьютерным данным, чтобы они стали информацией, понятной человеку.

Сохранение смысла данных с помощью типов

Как хранится информация в компьютере в виде необработанных данных, не теряя при этом своего смысла? Путем категорирования данных и определения их типа мы даем им контекст, определяющий их значение.

Допустим, например, что у нас есть три числа: 5155534, 5159592 и 4593030. Устанавливая категорию наших данных, скажем, считая их телефонными

номерами, номерами факсов или регистрационными номерами почтовых посылок, мы сохраняем контекст, а следовательно, и смысл наших данных. Когда установлена категория, каждое из неопределенных до того семизначных чисел обретает смысл.

В языках программирования для создания элементарных категорий пользовательских данных применяются *типы данных*. Например, почти во всех языках программирования определены типы данных для хранения и обработки текста (или *строк*) и чисел. Чтобы отличать числа друг от друга, можно использовать осмысленные имена, такие как `phoneNumber` или `faxNumber`. В более сложных случаях можно создавать собственные категории данных с помощью *объектов* и *классов объектов*, о чем будет рассказано в главе 12. Прежде чем заниматься созданием собственных категорий данных, посмотрим, какие категории встроены в ActionScript.

Типы данных ActionScript

В процессе программирования нам может потребоваться запомнить название продукта, цвет фона или количество звезд, которые нужно поместить на ночное небо. Для хранения данных в ActionScript предназначены следующие типы данных:

string (*строка*)

Для текстовых последовательностей, таких как `"hi there"`. *Строка* является последовательностью символов (букв, цифр и знаков пунктуации).

number (*число*)

Для чисел, например 351 или 7,5. Числа используются для счета и математических уравнений.

Boolean

Для принятия логических решений. С помощью булевых данных можно представлять или записывать состояние какого-либо условия или результат сравнения. Для булевых данных допустимы только два значения: `true` (истина) и `false` (ложь).

null и undefined

Чтобы показать *отсутствие* данных, в ActionScript есть два специальных значения данных: `null` и `undefined`. Можно рассматривать их как единственные разрешенные значения типов данных *null* и *undefined*.

array (*массив*)

Для списков из одного или более элементов данных.

movieclip (*клип*)

Для управления экземплярами клипов.

object (*объект*)

Произвольные встроенные или определенные пользователем классы данных.

Любой элемент данных, которые мы храним в ActionScript, попадает в одну из этих категорий. Перед тем как изучить все типы данных в главе 4, мы рассмотрим общие вопросы, связанные с использованием данных в целом.

Создание данных и задание их категорий

Есть два способа создать новый элемент данных с помощью ActionScript, и в обоих используются *выражения* – конструкции кода, представляющие данные в сценариях.

Символьное *выражение* (или *литерал*) является последовательностью букв, цифр и знаков пунктуации, которая и *есть* элемент данных. Литерал – это буквальное описание данных в исходном коде программы. Он – противоположность *переменной*, представляющей собою лишь контейнер, содержащий данные. Для каждого типа данных есть собственные правила создания литералов. Например, строковые литералы заключаются в кавычки, а числовые – нет. Вот несколько примеров литералов:

```
"loading...please wait"  // Строковый литерал
1.51                     // Числовой литерал
["jane", "jonathan"]     // Литерал массива
{x: 10, y: 15}           // Литерал объекта
```

Заметьте, что клипы не могут быть представлены литералами, а обращение к ним происходит по именам экземпляров.

Мы можем также программно создавать данные с помощью *сложных выражений*. Сложные выражения – это фразы программного кода, значение которых не буквально, а должно быть вычислено. Вычисленное значение и представляет собой элемент данных. Например, каждое из следующих сложных выражений имеет результатом один элемент данных:

```
1999 + 1       // Получаем числовое данное 2000
"1999" + "1"   // Получаем строковое данное "19991"
"hi " + "ma!"  // Получаем строковое данное "hi ma!"
firstName      // Дает значение переменной firstName
_currentframe  // Дает номер текущего кадра
new Date()     // Дает новый объект Date с текущими датой и временем
```

Обратите внимание, что отдельное символьное выражение, такое как 1999 или 1, может быть частью большего сложного выражения, как в *1999 + 1*.

Если данные создаются посредством символьных или сложных выражений, то мы должны сохранить каждый элемент данных, которым собираемся воспользоваться позднее. Результат выражения `"hi" + "ma!"` будет потерян, если не сохранить его, скажем, в переменной. Например:

```
// Этот элемент данных недолговечен и умирает сразу после своего создания
"hi " + "ma";
// Этот элемент данных хранится в переменной, и к нему
// можно потом обратиться через перменную welcomeMessage
var welcomeMessage = "hi " + "ma";
```

Каким образом мы устанавливаем для данных надлежащий тип? То есть как указать, что данное является числом, строкой, массивом или чем-то еще? В большинстве случаев мы не определяем сами категорию данных: интерпретатор ActionScript автоматически присваивает тип каждому элементу данных или определяет его, основываясь на наборе внутренних правил.

Автоматическая типизация литералов

Интерпретатор определяет тип литерала, исследуя его синтаксис, как разъясняется в комментариях к следующему фрагменту кода:

```
"animal"     // Кавычки идентифицируют "animal" как строку
1.35          // Если в литерале только целые числа и десятичная точка, это число
true         // Специальное ключевое слово true указывает на булев тип
null         // Специальное ключевое слово null указывает на тип null
undefined    // Специальное ключевое слово undefined указывает на тип undefined
["hello", 2, true]   // Квадратные скобки и разделяемые запятыми
                     // значения указывают, что это массив
{x: 234, y: 456}     // Фигурные скобки и пары имя/значение свойства,
                     // разделенные запятыми, указывают, что это объект
```

Как видите, использование правильного синтаксиса с литералами очень важно. Неверный синтаксис может привести к ошибке или неправильной интерпретации содержимого элемента данных, например:

```
animal   // Отсутствуют кавычки - animal интерпретируется
         // как переменная, а не строка текста
"1.35"   // Числа в кавычках рассматриваются как строки, а не числа
1. 35    // Пробел перед 3 вызывает ошибку
"animal  // Отсутствие закрывающей кавычки вызывает ошибку
```

Автоматическая типизация сложных выражений

Интерпретатор вычисляет значение выражения, чтобы определить его тип данных. Рассмотрим такой пример:

```
pointerX = _xmouse;
```

Поскольку `_xmouse` хранит расположение указателя мыши в виде числа, типом выражения `_xmouse` всегда будет число, поэтому переменная `pointerX` тоже становится числом.

Обычно тип данных, автоматически определяемый интерпретатором, соответствует типу, который нам нужен. Однако в некоторых двусмысленных случаях требуется понимать правила, которыми руководствуется интерпретатор при определении типа данных выражения (см. примеры 2.2 и 2.3). Рассмотрим следующее выражение:

```
"1" + 2;
```

Операнд слева от «+» представляет собой строку («1»), а операнд справа – число (2). Оператор «+» может применяться как с числами (сложение), так и со строками (конкатенация). Что будет значением выражения `"1" + 2` – число 3 или строка «12»? Чтобы разрешить неоднозначность, интерпретатор ис-

пользует фиксированное правило: оператор плюс (+) всегда отдает предпочтение строкам, а не числам, поэтому выражение `"1" + 2` вычисляется как строка «12», а не число 3. Это правило установлено произвольно, но оно позволяет единообразно интерпретировать код. Правило было выбрано с учетом типичного применения оператора «плюс»: если один из операндов представляет собой строку, то, скорее всего, мы хотим конкатенировать операнды, а не сложить их численные значения, как в следующем примере:

```
trace ("The value of x is: " + x);
```

Соединение несопоставимых типов данных или использование данных в контексте, не соответствующем их типу, вызывает неоднозначность. Это вынуждает интерпретатор производить автоматическое *преобразование* типа данных в соответствии с предсказуемыми правилами. Давайте рассмотрим случаи, в которых происходит автоматическое преобразование типа данных, и результаты преобразования данных из одного типа в другой.

Преобразование типов данных

Посмотрите внимательнее на пример из предыдущего раздела. В этом примере каждый элемент данных – «1» или 2 – принадлежит своему типу данных: первое является строкой, а второе – числом. Мы видели, что интерпретатор объединил два значения в строку «12». Обратите внимание, что интерпретатор сначала должен был *преобразовать число* 2 в *строку* «2». Только после этого автоматического преобразования стало возможным присоединить (конкатенировать) значение «2» к строке «1».

Преобразование типа данных означает просто изменение типа данных. Не все преобразования типа данных происходят автоматически: можно изменять тип данных явным образом, чтобы не полагаться на автоматическое преобразование, которое ActionScript осуществил бы по умолчанию. Явное преобразование типа данных называют также *приведением типов данных* (*typecasting*), или просто *приведением* (*casting*).

Автоматическое преобразование типа

В тех случаях, когда мы используем значение в контексте, не соответствующем типу данных этого значения, интерпретатор пытается выполнить преобразование. Это означает, что если интерпретатор рассчитывает получить данные типа A, а мы предоставляем ему данные типа B, то интерпретатор будет пытаться преобразовать данные типа B в данные типа A. Например, в следующем коде мы используем строку «Flash» как правый операнд в операторе вычитания. Так как в операторе вычитания могут использоваться только числа, интерпретатор попытается преобразовать строку «Flash» в число:

```
999 - "Flash";
```

Конечно, строку «Flash» нельзя преобразовать в допустимое число, поэтому она преобразуется в специальное числовое значение `NaN` (то есть Not-a-Number – «нечисло»). `NaN` является допустимым значением *числового* типа данных,

предназначенным специально для действий в такой ситуации. Если строка «Flash» преобразована в NaN, то наше выражение в итоге представляется интерпретатору в следующем виде (хотя этого промежуточного шага мы никогда не видим):

```
999 - NaN;
```

Оба операнда оператора вычитания теперь являются числами, поэтому операцию можно продолжить: 999 - NaN дает значение NaN, которое и является окончательным значением нашего выражения.

Выражения, приводящие к числовому значению NaN, не слишком полезны: большинство преобразований имеет более конструктивные результаты. Например, если в строке содержатся только числовые символы, оно может быть преобразовано в пригодное число. Выражение:

```
999 - "9";  // Число 999 минус строка "9"
```

интерпретируется как:

```
999 - 9;    // Число 999 минус число 9
```

что при вычислении выражения дает значение 990. Автоматическое преобразование чаще всего происходит с оператором «плюс», оператором равенства, операторами сравнения, а также с условными инструкциями или инструкциями цикла. Чтобы быть уверенным в результате выражения, в котором происходит автоматическое преобразование, нужно ответить на три вопроса: 1) какой тип данных предполагается в данном контексте, 2) что произойдет, если в этом контексте появится неожиданный тип данных и 3) каким будет результирующее значение, если произойдет преобразование.

Для ответа на первый и второй вопросы необходимо обратиться к соответствующим темам в данной книге (например, чтобы определить тип данных, ожидаемый в условной инструкции, см. главу 7).

В следующих трех таблицах, перечисляющих правила автоматического преобразования, содержится ответ на третий вопрос: «Если произойдет преобразование, каким будет результирующее значение?» В табл. 3.1 показаны результаты преобразования каждого типа данных в число.

Таблица 3.1. Преобразование в число

Исходные данные	Результат после преобразования
Undefined	0
Null	0
Булево значение	1, если исходное значение true; 0, если исходное значение false
Числовая строка	Эквивалентное числовое значение, если строка состоит только из чисел в десятичной системе счисления, пробельных символов, экспоненты, десятичной точки, знака «плюс» или знака «минус» (например, "-1.485e2")
Прочие строки	Пустые строки, нечисловые строки, в том числе строки, начинающиеся с "x", "0x" или "FF", преобразуются в NaN

Таблица 3.1 (продолжение)

Исходные данные	Результат после преобразования
"Infinity"	Infinity (бесконечность)
"-Infinity"	-Infinity
"NaN"	NaN
Массив	NaN
Объект	Значение, возвращаемое методом *valueOf( )* объекта
Клип	NaN

В табл. 3.2 показаны результаты преобразования каждого типа данных в строку.

Таблица 3.2. Преобразование в строку

Исходные данные	Результат после преобразования
undefined	"" (пустая строка)
null	"null"
Булево значение	"true", если исходное значение true; "false", если исходное значение false
NaN	"NaN"
0	"0"
Infinity	"Infinity" (бесконечность)
-Infinity	"-Infinity"
Другие числовые значения	Строка, эквивалентная числу. Например, 944.345 становится строкой "944.345"
Массив	Список значений элементов, разделенных запятыми
Объект	Значение, возвращаемое методом *toString( )* объекта. По умолчанию метод объекта *toString( )* возвращает "[object Object]". Метод *toString( )* может быть модифицирован для получения более полезного результата (например, *toString( )* объекта *Date* возвращает: "Sun May 14 11:38:10 EDT 2000")
Клип	Путь к экземпляру клипа, заданный в абсолютном виде, начиная с уровня документа в плеере, например "_level0.ball"

В табл. 3.3 показаны результаты преобразования каждого типа в тип булев.

Таблица 3.3. Преобразование в тип булев

Исходные данные	Результат после преобразования
Undefined	false
Null	false

Исходные данные	Результат после преобразования
NaN	false
0	false
Infinity	true
-Infinity	true
Прочие числовые значения	true
Непустая строка	true, если строка может быть преобразована в допустимое число, отличное от нуля, false, если нет; в ECMA-262 непустая строка всегда преобразовывается в true (Flash нарушает стандарт ECMA для сохранения совместимости с Flash 4)
Пустая строка ("")	false
Массив	true
Объект	true
Клип	true

Явное преобразование типов

Если правила автоматического (неявного) преобразования типов не подходят для наших целей, можно вручную (явно) изменить тип данных. Беря дело в свои руки, нужно помнить, что правила, перечисленные в предшествующих таблицах, по-прежнему применяются.

Преобразование в строку с помощью метода toString()

Чтобы преобразовать любой элемент данных в строку, можно вызвать метод *toString( )*, например:

```
x.toString();       // Получить строковое значение переменной x.
(523).toString();   // Возвращает "523". Обратите внимание на использование круглых
                    // скобок, чтобы "." не воспринималась как десятичная точка
```

При вызове метода *toString( )* с числом можно передать числовой аргумент, задающий основание системы счисления, в которой мы хотим представить преобразованную строку. Это дает удобный способ переключаться между шестнадцатеричными, десятичными и восьмеричными числами. Например:

```
var myColor = 255;
var hexColor = myColor.toString(16); // Присваивает hexColor значение "ff"
```

Преобразование в строку с помощью функции String()

Функция *String( )* дает такой же результат, как и метод *toString( )*, но использует иную грамматику:

```
String(x);     // Преобразует x в строку
String(523);   // Преобразует 523 в строку "523"
```

Не путайте глобальную функцию *String( )* с одноименным встроенным конструктором класса. Функция и конструктор описаны в «Справочнике по языку».

Преобразование в строку при конкатенации с пустой строкой

Поскольку оператор «плюс» (+) в правилах автоматического преобразования отдает предпочтение строкам, конкатенация пустой строки (`""`) с любым элементом данных преобразует этот элемент в строку.

```
// Преобразовать x в строку.
x + "";
// Определим положение 2 в числе 523. Сначала
// конкатенируем 523 и "", затем применяем метод String к преобразованному значению.
trace((523 + "").indexOf(2));
```

Преобразование в число с помощью функции Number()

Так же, как функция *String( )* преобразует данные в *строковый* тип, функция *Number( )* преобразует свой аргумент в числовой тип. Если преобразование в действительное число невозможно или бессмысленно, функция *Number( )* возвращает особое числовое значение, как описано в табл. 3.1. Вот некоторые примеры:

```
Number(age);      // Возвращает значение age, преобразованное в число
Number("29");     // Возвращает число 29
Number("sara");   // Возвращает NaN
```

Не путайте глобальную функцию *Number( )* с одноименным конструктором встроенного класса. Функция и конструктор описаны в «Справочнике по языку».

Поскольку введенные пользователем данные в экранных текстовых полях всегда принадлежат текстовому типу, при выполнении математических операций необходимо преобразовывать значения в текстовых полях в числа. Например, для нахождения суммы чисел, введенных в текстовые поля с переменными `price1_txt` и `price2_txt`, используем код:

```
totalCost = Number(price1_txt) + Number(price2_txt);
```

В противном случае `price1_txt` и `price2_txt` будут конкатенированы как строки, а не сложены как числа. Дополнительные сведения о текстовых полях можно найти в описании *TextField* в «Справочнике по языку».

Преобразование в число путем вычитания нуля

Чтобы заставить интерпретатор преобразовать элемент данных в число, можно вычесть нуль из этого элемента данных. При этом применяются правила, описанные в табл. 3.1:

```
"953" - 0      // Возвращает 953
"molly" - 0    // Возвращает NaN
x - 0          // Возвращает значение x, преобразованное в число
```

Преобразование в число с помощью функций parseInt() и parseFloat()

Функции *parseInt( )* и *parseFloat( )* преобразуют строку, содержащую числа и буквы, в число. Функция *parseInt( )* извлекает первое целое число, содержащееся в строке, при условии, что первый отличный от пробела символ строки является допустимым числовым символом. В противном случае *parseInt( )* возвращает NaN. Число, извлеченное с помощью *parseInt( )*, начинается с первого отличного от пробела символа строки и заканчивается символом перед первым нечисловым символом строки или первой десятичной точкой.

Вот некоторые примеры применения *parseInt( )*:

```
parseInt("1a")                // Возвращает 1
parseInt("1.3a"               // Возвращает 1
parseInt("     1a")           // Возвращает 1
parseInt("I am 14 years old") // Возвращает NaN (первый отличный от
                              // пробела символ - нечисло)
parseInt("14 years old")      // Возвращает 14

// Преобразовать десятичное число в шестнадцатеричное.
(255).toString(16);           // Возвращает: ff
// Преобразовать шестнадцатеричное число в десятичное.
parseInt("0xFF");             // Возвращает 255
```

Функция *parseFloat( )* возвращает первое число с плавающей точкой, которое находится в строке, при условии, что первый отличный от пробела символ является допустимым числовым символом. (Число с плавающей точкой – это положительное или отрицательное число, содержащее десятичное значение, например 10,5 или 345,678.) Подобно *parseInt( )*, *parseFloat( )* возвращает специальное числовое значение NaN, если первый отличный от пробела символ строки не является допустимым числовым символом. Число, извлекаемое *parseFloat( )*, представляет собой результат преобразования последовательности символов, начинающейся с первого отличного от пробела символа и заканчивающейся символом перед первым нечисловым символом (любым символом, отличным от +, -, 0–9, десятичной точки либо *e* или *E* для числа в экспоненциальном виде).

Вот некоторые примеры применения *parseFloat( )*:

```
parseFloat("1.3a");             // Извлекает 1.3
parseFloat("2.75 years old")    // Извлекает 2.75
parseFloat("1nce upon a time")  // Извлекает 1
parseFloat("I'm 3.5 feet tall") // Возвращает NaN
```

Дополнительные сведения о *parseInt( )* и *parseFloat( )*, включая задание основания при преобразовании из одной системы счисления в другую, можно найти в «Справочнике по языку».

Преобразование в булев тип

Когда требуется преобразовать элемент данных в булев тип, можно воспользоваться глобальной функцией *Boolean( )*, синтаксис которой аналогичен синтаксису функций *String( )* и *Number( )*. Например:

```
Boolean(5);  // Результатом является true
Boolean(x);  // Преобразует значение x в булеву величину
```

Не путайте глобальную функцию *Boolean( )* с одноименным конструктором встроенного класса. Функция и конструктор описаны в «Справочнике по языку».

Срок действия преобразования

Все преобразования типа, осуществляемые над переменными, элементами массивов и свойствами объектов, являются временными, если только преобразование не происходит при присваивании. Вот примеры временных преобразований:

```
var x = "10";      // x является строкой
y = x - 5;         // y теперь 5; значение x было временно преобразовано в число
trace(typeof x);   // Выводит: "string"; преобразование было временным,
                   // поскольку произошло случайно при вычислении выражения
```

А вот постоянные преобразования как результат присваивания:

```
x = "10";          // x является строкой
x = x - 5;         // x преобразуется в число
trace(typeof x);   // Выводит: "number". Преобразование стало
                   // постоянным, поскольку произошло при присваивании
```

Определение типа существующего элемента данных

Для определения типа данных, содержащихся в указанном выражении, например перед выполнением участка кода, применяется оператор *typeof*:

```
typeof expression;
```

Оператор *typeof* возвращает строку, указывающую тип данных *expression*, согласно табл. 3.4.

Таблица 3.4. Значения, возвращаемые typeof

Исходный тип данных	Значение, возвращаемое typeof
Число	"number"
Строка	"string"
Булев	"boolean"
Объект	"object"
Массив	"object"
null	"null"
Клип	"movieclip"
Функция	"function"
undefined	"undefined"

Вот несколько примеров:

```
trace(typeof "game over");    // Выводит: "string" в окне Output
var x = 5;
trace(typeof x);              // Выводит: "number"
var now = new Date();
trace(typeof now);            // Выводит: "object"
```

Как показано в примере 3.1, при использовании в инструкции *for-in* функция *typeof* дает удобный способ нахождения всех экземпляров клипа на временной диаграмме. Когда клипы найдены, ссылки на них можно записать в массив и осуществить программную обработку. (Если пример 3.1 не совсем вам понятен, вернитесь к нему, закончив изучение части I.)

Пример 3.1. Заполнение массива динамически идентифицируемыми клипами

```
// Создать массив для хранения клипов.
var childClips = new Array();

// Проверить все свойства главной временной диаграммы.
for (prop in _root) {
  // Если текущее свойство является клипом...
  if (typeof _root[prop] == "movieclip") {
    // ...добавить его в массив клипов.
    childClips.push(_root[prop]);
  }
}

// После заполнения массива можно использовать его
// для управления клипами, ссылки на которые в нем содержатся.
childClips[0]._x = 0;  // Поместить первый клип у левого края сцены
childClips[1]._y = 0;  // Поместить второй клип у верхнего края сцены
```

Элементарные и сложные типы данных

До сих пор мы работали преимущественно с числами и строками – чаще всего встречающимися *элементарными* (*primitive*) типами данных. Элементарные типы данных являются базовыми единицами языка: каждое элементарное значение представляет один элемент данных (в противоположность массиву, содержащему несколько элементов) и непосредственно содержит эти данные, а не адрес памяти, по которому они находятся.

ActionScript поддерживает следующие элементарные типы данных: *числовой*, *строковый*, *булев*, *undefined* и *null*. В ActionScript нет отдельного типа данных для одиночных символов, типа *char* в C/C++ (строки в ActionScript – это элементарный тип данных, а не массивы символов, как в C/C++).

Элементарные типы данных могут содержать текстовые сообщения, номера кадров, секретные пароли и т. д., но их не так легко приспособить для хранения более сложных данных. Для более сложной обработки данных, например, при моделировании движения десятка прыгающих мячиков или организации вопросника с 500 вопросов и ответов, мы обращаемся к *слож-*

ным (*composite*) типам данных. Сложные типы данных позволяют работать с несколькими фрагментами взаимосвязанных данных как с одним элементом данных.

ActionScript поддерживает следующие сложные типы данных: *массив* (*array*), *объект* (*object*) и *клип* (*movieclip*). Функции формально являются особыми бъектами и потому рассматриваются как сложные данные, но в таком качестве они используются редко. Подробнее о функциях как типе данных рассказано в главе 9.

В то время как отдельное число является простым элементом данных, список (то есть *массив*) нескольких чисел является *сложным* элементом данных. Вот практический пример, показывающий пользу сложных типов данных. Допустим, мы хотим сохранить имя клиента Derek. Можно создать переменную, которая хранит его имя как значение элементарного типа:

```
var custName = "Derek";
```

Однако такой подход становится довольно утомительным, если нужно добавить много клиентов. Для хранения имен всех клиентов потребуется использовать последовательно именуемые переменные – `cust1Name`, `cust2Name`, `cust3name` и т. д. Но с помощью массива можно сохранить данные гораздо более эффективно:

```
customers = ["Derek", "James", "Joe"];
```

Мило и аккуратно. В следующих главах мы узнаем о сложных типах значительно больше.

Копирование, сравнение и передача данных

Существует три основных способа обращения с данными. Можно *копировать* (*copy*) данные (например, присвоить значение переменной `x` переменной `y`), *сравнивать* (*compare*) данные (например, проверить равенство `x` и `y`) и *передавать* (*pass*) данные (например, передать переменную в функцию в качестве аргумента). Значения элементарных типов копируются, сравниваются и передаются совсем иначе, чем значения составных типов. Когда элементарные данные копируются в переменную, эта переменная получает собственный уникальный и личный экземпляр данных, хранящийся в памяти отдельно от других данных. Поэтому следующие строки кода приводят к тому, что строка «Dave» сохраняется в памяти дважды: один раз в пространстве памяти, отведенном для `name1`, и второй раз – в пространстве памяти, отведенном для `name2`:

```
name1 = "Dave";
name2 = name1;
```

Мы говорим, что элементарные данные копируются *по значению* (*by value*), потому что буквальное значение данных сохраняется в пространстве памяти, выделенном для переменной. Напротив, когда в переменную копируются составные данные, то в памяти, отведенной переменной, сохраняется

только *ссылка* на данные (а не фактические данные). Эта ссылка сообщает интерпретатору, где хранятся фактические данные (то есть их адрес в памяти). Когда переменная, содержащая составные данные, копируется в другую переменную, то копируется *ссылка* (часто называемая указателем), а не сами данные. Поэтому говорят, что составные данные копируются *по ссылке* (*by reference*).

Это хорошее решение, потому что дублирование больших массивов и других составных типов данных было бы весьма неэффективно. Но для кода программы это имеет важные последствия. Когда нескольким переменным в качестве значения присваивается один и тот же фрагмент составных данных, все эти переменные *не* сохраняют уникальный экземпляр данных (как произошло бы в случае элементарных данных). Вместо этого все переменные указывают на один экземпляр составных данных. Если изменяется значение данных, это отразится на всех переменных, которые указывают на эти данные.

Посмотрим, какую из этого можно извлечь пользу. Когда две переменные ссылаются на одни и те же элементарные данные, каждая переменная получает собственный экземпляр этих данных. Присвоим переменной `x` значение 12:

```
var x = 12;
```

Теперь присвоим значение `x` новой переменной, `y`:

```
var y = x;
```

Как можно догадаться, теперь `y` равна 12. Но `y` обладает собственным экземпляром значения 12, отличным от экземпляра в `x`. Если изменить значение `x`, на значении `y` это не отразится:

```
x = 15;
trace(x); // Выводит 15
trace(y); // Выводит 12
```

Значение `y` не изменилось в результате изменения `x`, потому что при присваивании `x` переменной `y` последняя получила собственный экземпляр числа 12 (то есть *элементарных* данных, содержащихся в `x`).

Теперь попробуем сделать то же самое с *составными* данными. Создадим новый массив из трех элементов и присвоим его переменной `x`:

```
var x = ["first element", 234, 18.5];
```

Теперь, как и ранее, присвоим значение `x` переменной `y`:

```
var y = x;
```

Теперь значение `y` такое же, как значение `x`. Но что является значением `x`? Запомните: поскольку `x` ссылается на массив, являющийся составным элементом данных, значением `x` является не буквально массив `["first element", 234, 18.5]`, а просто ссылка на эти данные. Поэтому, присваивая `x` переменной `y`, мы копируем в `y` не сам массив, а ссылку, содержащуюся в `x` и указывающую на этот массив. Поэтому как `x`, так и `y` указывают на один и тот же массив, содержащийся где-то в памяти.

Если изменить этот массив через переменную x, например, так:

```
x[0] = "1st element";
```

то изменение отразится и на y:

```
trace(y[0]);  // Выводит: "1st element"
```

Аналогично, если модифицировать массив через y, то изменение можно увидеть через x:

```
y[1] = "second element";
trace (x[1]);  // Выводит: "second element"
```

Эту связь можно разорвать с помощью функции *slice( )*, которая создает совершенно новый массив:

```
var x = ["first element", 234, 18.5];
// Копировать каждый элемент x в новый массив, хранящийся в y
var y = x.slice(0);
y[0] = "hi there";
trace(x[0]);  // Выводит: "first element" (а не "hi there")
trace(y[0]);  // Выводит: "hi there" (а не "first element")
```

Разовьем этот пример и посмотрим, как *сравниваются* элементарные и составные значения данных. Присвоим x и y одинаковое элементарное значение и затем сравним обе переменные:

```
x = 10;
y = 10;
trace(x == y);  // Выводит: true
```

Поскольку x и y содержат элементарные данные, они сравниваются по значению. При сравнении по значению данные сравниваются буквально. Число 10 в x считается равным числу 10 в y, поскольку эти числа составлены из одинаковых байтов.

Теперь присвоим x и y идентичные варианты одинаковых составных данных и снова сравним две переменные:

```
x = [10, "hi", 5];
y = [10, "hi", 5];
trace(x == y);  // Выводит: false
```

На этот раз x и y содержат составные данные, поэтому они сравниваются по ссылке. Массивы, присвоенные x и y, состоят из одинаковых байтов, но переменные x и y не равны между собой, потому что они не хранят ссылку на один и тот же составной элемент данных. Однако посмотрим, что происходит при копировании ссылки, находящейся в x, в переменную y:

```
x = y;
trace(x == y);  // Выводит: true
```

Теперь, когда ссылки одинаковые, значения считаются равными. Таким образом, результат сравнения зависит от ссылок, содержащихся в переменных, а не фактических значений байтов в массивах, на которые они указывают.

Элементарные и составные данные по-разному обрабатываются и при передаче в функции, о чем говорится в разделе «Передача параметров по значению и по ссылке» главы 9. Самое важное, что при передаче элементарной переменной в качестве аргумента в функцию любые изменения этого элемента данных, осуществленные функцией, не отражаются на исходной переменной. Однако при передаче составной переменной изменения в функции *влияют* на исходную переменную. Это означает, что при передаче в функцию целочисленной переменной `x` изменения, производимые с ней в функции, не влияют на ее первоначальное значение в вызвавшей функцию программе. Но если передать в функцию массив `y`, любые изменения этого массива внутри функции изменят исходное значение `y` вне функции (потому что изменения в массиве касаются данных, на которые указывает `y`).

Вперед!

Мы познакомились с представлением данных в ActionScript и готовы к дальнейшему изучению. В главе 4 мы изучим типы данных: *числовой*, *строковый*, *булев*, *undefined* и *null*, а в главе 5 научимся обрабатывать данные с помощью операторов. Затем мы изучим сложные типы данных, такие как *клипы*, *массивы* и *объекты*.

4

Элементарные типы данных

Элементарные данные состоят из простых символов или ключевых слов, например чисел 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 или строк «a», «b», «c». Как мы узнали в предыдущей главе, ActionScript поддерживает простые типы данных: *числовой*, *строковый*, *булев*, *undefined* и *null*. В этой главе мы научимся определять, просматривать и изменять данные каждого из этих типов.

Числовой тип

Числа используются для счета, математических действий и хранения числовых свойств в фильмах (таких как номер текущего кадра или его местоположение на сцене). Рассмотрим, как в ActionScript определяются и обрабатываются числа.

Целые числа и числа с плавающей точкой

В большинстве языков программирования различаются два вида чисел: *целые числа* (*integers*) и *числа с плавающей точкой* (*floating-point numbers*). Целые числа не имеют дробной части. Они могут быть положительными, отрицательными или нулем. Числа с плавающей точкой могут содержать дробную часть, представленную после десятичной точки, например 0.56, 199.99 или 3.14159. Таким образом, 1, 34523, –3, 0 и –9999999 представляют собой целые, а 223.45 и –0.56 – числа с плавающей точкой.

Числовые литералы

Ранее мы узнали, что *литерал* является непосредственным представлением отдельного фиксированного значения. Числовой тип данных поддерживает три вида литералов: целочисленные литералы, литералы чисел с плавающей точкой и специальные числовые значения. Первые две категории представляют *действительные числа* (числа, имеющие фиксированную математическую величину); в третью категорию входят числа, представляющие числовые понятия, например «бесконечность».

Целочисленные литералы

Целочисленные литералы, такие как 1, 2, 3, 99 и –200, должны соответствовать правилам:

- Целые числа не могут содержать десятичную точку или дробное значение.
- Целые числа не должны выходить за пределы максимальных и минимальных числовых значений, допустимых в ActionScript. См. свойства `MIN_VALUE` и `MAX_VALUE` объекта *Number* в «Справочнике по языку».
- Целые числа в системе по основанию 10 не должны начинаться с нуля (например, недопустимы числа 002, 000023 или 05).

Целочисленные значения не всегда записываются в системе счисления по основанию 10 (то есть как десятичные). ActionScript поддерживает также числовые литералы по основанию 8 (восьмеричные) и по основанию 16 (шестнадцатеричные). Введение в десятичные, восьмеричные и шестнадцатеричные числа можно найти по адресу:

http://www.moock.org/asdg/technotes/basePrimer/

Ведущий ноль используется для обозначения восьмеричного числа. Например, восьмеричное число 723 представляется в ActionScript так:

```
0723  // десятичное 467 (7*64 + 2*8 + 3*1)
```

Для обозначения шестнадцатеричного литерального целого мы помещаем перед числом 0x (или 0X), например:

```
0x723  // десятичное 1827 (7*256 + 2*16 + 3*1)
0xFF   //  десятичное 255 (15*16 + 15*1)
```

Шестнадцатеричные числа часто используются для задания значений цветов, но в простейших программах нужны только числа по основанию 10. Следите за тем, чтобы при преобразовании строк в числа нежелательные ведущие нули удалялись, как показано в примере 4.1.

Пример 4.1. Удаление ведущих нулей

```
function trimZeros(theString) {
  while (theString.charAt(0) == "0" || theString.charAt(0) == " ") {
    theString = theString.substring(1, theString.length);
  }
  return theString;
}
testString = "00377";
trace(trimZeros(testString));  // Выводит: 377
```

Литералы с плавающей точкой

Литералы с плавающей точкой представляют числа, содержащие дробную часть. В литерале с плавающей точкой могут содержаться все или некоторые из следующих составляющих:

целое десятичное число
десятичная точка (.)
дробная часть (представленная как десятичное число)
порядок

Первые три составляющие довольно просты: в числе 3,14 «3» является десятичным числом, «.» – это десятичная точка, а «14» (то есть 14 сотых) представляет собой дробную часть. Однако четвертую составляющую (порядок) надо рассмотреть более внимательно.

Чтобы представить очень большое положительное или отрицательное число как число с плавающей точкой, можно присоединить к числу порядок с помощью буквы E (или e). Чтобы определить величину числа с порядком, надо умножить число на 10 в степени, заданной порядком. Например:

```
12e2   // 1200 (10 в квадрате равно 100, умножить на 12 дает 1200)
143E-3 // 0.143 (10 в степени -3 дает .001, умножить на 143 дает 0.143)
```

Возможно, вы узнали стандартную *научную нотацию*, или *экспоненциальное представление*. Если вы не очень сильны в математике, вот простой совет по преобразованию: если порядок положительный, передвиньте десятичную точку вправо на равное ему количество разрядов; если порядок отрицательный, передвиньте десятичную точку влево на равное ему количество разрядов.

Иногда ActionScript возвращает в качестве результата вычислений число с порядком, если результат является очень большим или очень маленьким числом. Заметьте, однако, что порядок E (или e) служит просто для удобства обозначений. Для возведения числа в произвольную степень применяется встроенная функция *Math.pow( )*, документированная в «Справочнике по языку».

Точность представления чисел с плавающей точкой

Flash использует числа с плавающей точкой *двойной точности* (*double-precision*), которые обеспечивают точность около 15 значащих цифр. (Ведущие и замыкающие нули, а также порядок не входят в число этих значащих цифр.) Другими словами, Flash может представить число 123456789012345, но не 1234567890123456. Точность ограничивает не величину числа, а точность его представления. 2e16 больше, чем 123456789012345, но содержит всего одну значащую цифру.

Иногда результаты вычислений в ActionScript округляются нежелательным образом, создавая числа вроде 0,14300000000000001 вместо 0,143. Дело в том, что компьютеры преобразуют числа по любому основанию во внутреннее двоичное представление, в результате чего могут образовываться бесконечные двоичные дроби (типа десятичной 0,3333333...). Однако точность компьютеров конечна, и они не могут точно представлять бесконечные дроби. Чтобы приспособиться к незначительным расхождениям, следует вручную округлять числа, если различие отрицательно влияет на выполнение кода. Например, округлим `myNumber` до трех десятичных разрядов:

```
myNumber = Math.round(myNumber * 1000) / 1000;
```

А вот функция, которую можно использовать для округления произвольного числа до заданного числа десятичных разрядов:

```
function roundNum(theNumber, decPlaces) {
  if (decPlaces >= 0) {
    var temp = Math.pow(10, decPlaces);
    return Math.round(theNumber * temp) / temp;
  }
}

// Округлить число до двух десятичных разрядов
trace(roundNum(1.12645, 2));  // Выводит: 1.13
```

Специальные значения числового типа данных

Целые числа и числа с плавающей точкой исчерпывают почти все допустимые значения для *числового* типа, но есть специальные ключевые слова, представляющие следующие понятия: «нечисло», «минимальное разрешенное значение», «максимальное разрешенное значение», «бесконечность», «отрицательная бесконечность».

Каждое из этих особых значений можно присваивать переменным и свойствам либо использовать в литеральных выражениях, как любой другой числовой литерал. Чаще, однако, специальные числовые значения возвращаются интерпретатором как результат вычисления каких-либо выражений.

Нечисло: NaN

Иногда математическое вычисление или попытка преобразования типа данных возвращает значение, которое просто не является числом. Например, вычисление 0/0 невозможно, а следующее выражение нельзя преобразовать в конечное число:

```
23 - "go ahead and try!"
```

Чтобы охватить данные, имеющие *числовой* тип, но не являющиеся действительными числами, в ActionScript есть ключевое слово NaN. Хотя NaN не представляет никакое число, это допустимое значение *числового* типа, что демонстрирует следующий код:

```
x = 0/0;
trace(x);          // Выводит: NaN
trace(typeof x);   // Выводит: "number"
```

Так как NaN не является конечным числовым значением, при сравнении оно не считается равным самому себе. Если две переменные содержат значение NaN, они не считаются равными (хотя нам они кажутся равными). Это препятствие можно обойти при помощи встроенной функции *isNaN()*, проверяющей, содержит ли переменная значение NaN:

```
x = 12 - "this doesn't make much sense";  // x теперь NaN
trace(isNaN(x));                          // Выводит: true
```

Минимальное и максимальное допустимые значения: MIN_VALUE и MAX_VALUE

ActionScript предоставляет широкий, но не безграничный диапазон чисел. Максимально допустимым значением является 1,7976931348623157e+308, а минимально допустимое значение – 4.94065645841247e–324. Очевидно, работать с этими числами не очень удобно, поэтому вместо них применяются специальные значения `Number.MAX_VALUE` и `Number.MIN_VALUE`.

Значение `Number.MAX_VALUE` удобно, если требуется проверить, является ли результат вычислений допустимым положительным числом:

```
z = x*y;
if (z <= Number.MAX_VALUE && z >= -Number.MAX_VALUE) {
  // Число допустимо
}
```

Обратите внимание, что `Number.MIN_VALUE` представляет собой самое маленькое допустимое *положительное* значение, а не самое большое по абсолютной величине *отрицательное* значение. Самым большим по абсолютной величине отрицательным значением является `–Number.MAX_VALUE`.

Бесконечность и отрицательная бесконечность: Infinity и –Infinity

Если вычисление дает результат, превосходящий `Number.MAX_VALUE`, ActionScript использует для представления результата вычислений ключевое слово `Infinity` (такая ситуация называется *переполнением*). Аналогично, если вычисление приводит к результату, более отрицательному, чем самое «большое» (по абсолютной величине) отрицательное число, ActionScript представляет результат ключевым словом `–Infinity`. `Infinity` и `–Infinity` можно использовать непосредственно как литеральные числовые выражения.

Иррациональные числа

Кроме специальных числовых значений `NaN`, `Infinity`, `–Infinity`, `Number.MAX_VALUE` и `Number.MIN_VALUE`, ActionScript предоставляет удобный доступ к математическим константам через объект *Math*. Например:

```
Math.E        // Значение e, основания натуральных логарифмов
Math.LN10     // Натуральный логарифм 10
Math.LN2      // Натуральный логарифм 2
Math.LOG10E   // Логарифм e по основанию 10
Math.LOG2E    // Логарифм e по основанию 2
Math.PI       // Число Пи (то есть 3.1415926...)
Math.SQRT1_2  // Квадратный корень из 1/2
Math.SQRT2    // Квадратный корень из 2 (то есть 1.4142135...)
```

Константы представляют собой сокращенную запись приближенных значений часто встречающихся иррациональных чисел. (Иррациональными называются числа, которые нельзя представить в виде простой дроби, такой как 1/3. Не следует путать их с `NaN`, `Infinity` и т. д.) Используются иррациональные числа точно так же, как любые другие свойства объекта:

```
area = Math.PI*(radius*radius);
```

Полный список поддерживаемых констант можно найти в описании объекта *Math* в «Справочнике по языку».

Действия с числами

Для работы с числами можно объединять их в математические выражения с помощью операторов и вызывать встроенные функции для выполнения сложных математических операций.

Использование операторов

Основные арифметические операции – сложение, вычитание, умножение и деление – выполняются с помощью операторов +, –, * и /. Операторы можно использовать с любыми числовыми литералами или контейнерами данных, например переменными. Математические выражения вычисляются в порядке, который определяется приоритетами, приведенными в табл. 5.1. Например, умножение выполняется раньше, чем сложение. Вот примеры допустимого применения математических операторов:

```
x = 3 * 5;            // Присвоить x значение 15
x = 1 + 2 - 3 / 4;    // Присвоить x значение 2.25
x = 56;
y = 4 * 6 + x;        // Присвоить y значение 80
y = x + (x * x) / x;  // Присвоить y значение 112
```

Встроенные математические функции

Для выполнения более сложных математических действий применяются встроенные функции объекта *Math*. Например:

```
Math.abs(x)       // Абсолютная величина x (модуль x)
Math.min(x, y)    // Меньшее из значений x и y
Math.pow(x, y)    // x в степени y
Math.round(x)     // Значение x, округленное до ближайшего целого
```

Математические функции возвращают значения, которые можно использовать в выражениях так же, как числа, представленные литералами. Допустим, например, что мы хотим моделировать шестигранную игральную кость. Можно использовать функцию *random( )*, чтобы получить случайное число с плавающей точкой в интервале от 0 до 0,999...:

```
dieRoll = Math.random();
```

Это значение мы умножаем на 6, что дает число с плавающей точкой в интервале от 0 до 5,999, и добавляем к результату 1:

```
dieRoll = dieRoll * 6 + 1;  // Присваивает dieRoll число от 1 до 6.999
```

Наконец, мы используем функцию *floor( )* для округления до ближайшего снизу целого числа:

```
dieRoll = Math.floor(dieRoll);  // Присваивает dieRoll целое значение между 1 и 6
```

После объединения в одно выражение подсчет выпавшего на кости числа выглядит так:

```
// Присваивает dieRoll целое значение между 1 и 6 включительно
dieRoll = Math.floor(Math.random() * 6 + 1);
```

Строковый тип

Строковый тип данных используется для текстовых данных (букв, знаков пунктуации и других символов). Строковый литерал – это любая комбинация символов, заключенная в кавычки:

```
"asdfksldfsdfeoif"  // Расстроенная строка
"greetings"         // Дружественная строка
"unity@moock.org"   // Строка саморекламы
"123"               // Похоже на число, но это строка
'singles'           // Одинарные кавычки тоже допустимы
```

Прежде чем говорить о том, как образовывать строковые литералы, посмотрим, какие символы допускаются в строках.

Кодировка символов

Как и все данные, символы текста хранятся в компьютере с использованием числового кода. Они *кодируются* для хранения и *декодируются* для показа с использованием *набора символов* (*character set*), который *сопоставляет* символы и их численные коды. Наборы символов различны для разных языков и алфавитов. Первоначально в большинстве западных приложений применялись наборы, производные от ASCII – стандартного набора, содержащего лишь 128 символов: английский алфавит, цифры, основные символы пунктуации и особые *управляющие символы*. Современные западные приложения поддерживают семейство наборов символов, в совокупности называемых ISO-8859. Каждый из наборов символов ISO-8859 кодирует стандартный латинский алфавит (от «A» до «Z»), а также переменный набор букв, необходимых в целевых языках. В ActionScript в качестве основного набора символов принят ISO-8859-1, известный еще как *Latin 1* (он поддерживает также второй набор символов под названием *Shift-JIS* для японского языка).

Набор символов Latin 1 охватывает большинство западноевропейских языков, таких как французский, немецкий, итальянский, испанский, португальский и т. д., но не греческий, турецкий, славянские и русский. Для поддержки всего спектра мировых языков во Flash Player 6 используется кодировка *Unicode* – предпочтительный международный стандарт кодировки, который может содержать более миллиона символов. Flash Player 6 может отображать большинство поддерживаемых Unicode символов, независимо от национальных настроек, при условии, что соответствующий шрифт установлен на машине пользователя. Однако Flash Player 6 не поддерживает написание с обратным или двойным направлением, например арабское или иврит.

В приложении B перечислены кодовые *позиции* (*code points*) всех символов Latin 1 в Unicode, представляющие собой числовые позиции символов в наборе Unicode. Позднее мы увидим, как использовать эти коды для работы с символами в сценариях и отображения символов Unicode. Ресурсы для получения дополнительных сведений о Unicode приведены в приложении A.

Строковые литералы

Чаще всего строка образуется посредством заключения группы символов в одинарные или двойные кавычки:

```
"hello"
'Nice night for a walk.'
"The equation is 12 + 4 = 16, which programmers see as 12 + 4 == 16."
```

Если строка начинается с двойной кавычки, то и заканчиваться она должна тоже двойной кавычкой. Аналогично, если начало строки ограничено одинарной кавычкой, то завершить строку должна тоже одинарная кавычка. Однако строка в двойных кавычках может содержать одинарные кавычки, и наоборот. Следующие строки, например, содержат допустимое сочетание одинарных и двойных кавычек:

```
"Nice night, isn't it?"              // Апостроф внутри двойных кавычек
'I said, "What a pleasant evening!"'  // Двойные кавычки внутри одинарных
```

Пустая строка

Самая короткая возможная строка – это *пустая строка*, которая не содержит символов:

```
""
''
```

Пустая строка иногда удобна, если надо определить, содержит ли переменная строковое значение, которое можно использовать. Ниже проверяется, есть ли в текстовом поле ввода `firstName_txt` пригодное значение:

```
if (firstName == "") {
  trace("Вы забыли ввести свое имя!");
}
```

Пустая строка не считается равной одному или нескольким пробелам (например, "" – не то же самое, что " "). Модифицируем этот код, чтобы он проверял, есть ли в строке хоть один пригодный символ, то есть любой символ Unicode, код которого больше 32 (все, что ниже 32, представляет собой управляющие или пробельные символы). Рассмотрим пример 4.2.

Пример 4.2. Специальный метод String.isEmpty( )

```
// Проверяет, есть ли в строке символ Unicode с кодовой позицией больше 32.
String.prototype.isEmpty = function () {
  // Если найден пригодный символ...
  for (var i=0; i < this.length; i++) {
```

```
      if (this.charCodeAt(i) > 32) {
         // строка не "пустая"
         return false;
      }
   }

   // Пригодные символы не найдены. Строка “пуста”.
   return true;
}

// При помощи нового метода проверяем, есть ли пригодное значение
// в текстовом поле firstName_txt.
if (firstName_txt.text.isEmpty()) {
  trace("Вы забыли ввести свое имя!");
} else {
  trace("Welcome, " + firstName_txt.text);
}
```

Escape-последовательности

Ранее мы видели, что одинарные кавычки (') могут находиться внутри литералов, заключенных в двойные кавычки, а двойные кавычки (") – внутри литералов, заключенных в одинарные кавычки. А как быть, если нужны и те и другие? Например:

```
'I remarked "Nice night, isn't it?"'
```

В таком виде эта строка кода вызовет ошибку, поскольку компилятор ActionScript решит, что строковый литерал заканчивается апострофом в слове «isn’t». Интерпретатор прочтет это так:

```
'I remarked "Nice night, isn'          // Оставшаяся часть считается
                                       // бессмысленным мусором
```

Для того чтобы поместить одинарную кавычку внутрь строкового литерала, ограниченного одинарными кавычками, надо использовать *escape-последовательность*. Escape-последовательность представляет значение литеральной строки с помощью символа обратной косой черты (\), за которым следует нужный символ или представляющий его код. Вот escape-последовательности для одинарной и двойной кавычки:

```
\'
\"
```

Поэтому наше пожелание доброго вечера, надлежащим образом выраженное как строковый литерал, должно быть таким:

```
'I remarked "Nice night, isn\'t it?"'  // Обход апострофа!
```

Прочие escape-последовательности, которые можно использовать для представления различных специальных или зарезервированных символов, приведены в табл. 4.1.

Таблица 4.1. Escape-последовательности в ActionScript

Escape-последовательность	Значение
\b	Символ забоя (ASCII 8)
\f	Символ подачи листа (ASCII 12)
\n	Символ новой строки; вызывает перевод строки (ASCII 10)
\r	Символ возврата каретки (CR); вызывает перевод строки (ASCII 13)
\t	Символ табуляции (ASCII 9)
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Символ обратной косой черты; необходим при использовании обратной косой черты как литерального символа, чтобы помешать интерпретации обратного слэша (\) как начала escape-последовательности

Escape-последовательности в стиле Unicode

Не все символы Unicode есть на клавиатуре. Чтобы включить в строку отсутствующие на клавиатуре символы, используются escape-последовательности в стиле Unicode.

Escape-последовательности в стиле Unicode начинаются с обратной косой черты и строчной «*u*», то есть \u, а далее следует четырехзначное шестнадцатеричное число, соответствующее кодовой позиции символа в Unicode, например:

```
\u0040  // Знак @
\u00A9  // Символ copyright
\u0041  // Заглавная буква "A"
\u2014  // Длинное тире (em dash)
```

Кодовая позиция представляет собой уникальный идентификационный номер, присваиваемый каждому символу в наборе символов Unicode. Список кодовых позиций Unicode для Latin 1 приведен в приложении B. Кодовые позиции для других языков можно найти на сайте Unicode:

http://www.unicode.org/charts

При возникновении трудностей в поиске символа среди тысяч кодовых позиций, обратитесь к полезным предложениям Консорциума Unicode:

http://www.unicode.org/unicode/standard/where/

Если в escape-последовательность преобразуются только символы из набора Latin 1, то можно использовать сокращенную форму стандартной escape-последовательности Unicode. Сокращенная форма состоит из префикса \x, за которым следует двузначное шестнадцатеричное число, представляющее кодировку символа в Latin 1. Так как кодовые позиции Latin 1 совпадают

с первыми 256 позициями Unicode, можно пользоваться справочной таблицей приложения B, просто удаляя u00, как в следующих примерах:

```
\u0040  // Escape-последовательность Unicode
\x40    // сокращенный формат \x
\u00A9  // Unicode...
\xA9    // ...смысл ясен
```

Для вставки любого символа в строку кроме escape-последовательностей Unicode можно воспользоваться более громоздкой встроенной функции *fromCharCode( )*, описываемой ниже в разделе «Функции кодов символов». Функция *fromCharCode( )* принимает кодовую позицию в виде любого выражения, возвращающего число, тогда как escape-последовательность требует числового литерала.

Flash Player 5 поддерживал escape-последовательности в стиле Unicode, но не поддерживал Unicode. С помощью escape-последовательности Unicode во Flash 5 можно было задавать только кодовые позиции в наборах символов Latin 1 или Shift-JIS, а не кодовые позиции Unicode. Вставка кодовых позиций Unicode не даст правильного символа Unicode во Flash Player 5 или в файлах *.swf*, экспортированных во Flash 5.

Ввод многоязычных и специальных символов

Flash Player 6 полностью поддерживает Unicode, но в среде разработки Flash MX такой поддержки нет. Среда разработки Flash MX позволяет вводить символы из наборов символов Latin 1, Shift-JIS и MacRoman. Другие наборы символов вместе с клавиатурами и ОС для других языков могут вызвать трудности. Поэтому самый гибкий способ ввода текста Unicode во Flash MX состоит в загрузке его из внешнего файла в формате Unicode. В Windows такие программы, как Notepad, Word и WordPad, могут сохранять текст в формате Unicode. Проще всего импортировать текстовый файл в кодировке Unicode с помощью директивы #include, используемой вместе со специальным комментарием //!-- UTF8, в результате чего содержимое файла копируется и вставляется в фильм Flash на этапе компиляции. Сведения о применении #include с Unicode можно найти в описании этой директивы в «Справочнике по языку». Внешний текст Unicode можно также хранить в XML или переменных в кодировке URL. О загрузке внешних файлов XML или переменных идет речь в описании методов *XML.load( )* и *LoadVars.load( )* в «Справочнике по языку».

Обратите внимание, что конкретным форматом кодировки, поддерживаемым #include и *LoadVars*, служит UTF-8, тогда как *XML* поддерживает форматы UTF-8, UTF16-BE и UTF-16LE (то есть форматы с обратным или прямым порядком байтов). Файл в кодировке UTF-16 должен начинаться с маркера порядка байтов (BOM), указывающего на прямой или обратный порядок байтов. Большинство текстовых редакторов автоматически добавляет BOM. Если в файле остутствует BOM, предполагается использование коди-

ровки UTF-8. Если нет уверенности, следует использовать кодировку UTF-8, в которой порядок байтов не важен. Дополнительно о UTF и BOM можно прочесть по адресу:

http://www.unicode.org/unicode/faq/utf_bom.html

Чтобы вставить небольшое количество символов Unicode непосредственно в среде разработки Flash MX, можно воспользоваться стандартной шестнадцатеричной escape-последовательностью Unicode (описанной ранее в этой главе) или методом *String.fromCharCode( )* (описываемым ниже в этой главе).

Например, следующий код создает глобальную переменную `euro`, которая содержит знак «евро», и отображает этот символ в текстовом поле `price_txt`.

```
_global.euro = "\u20AC";
this.createTextField("price_txt", 1, 100, 100, 200, 20);
price_txt.text = "99 " + euro;
```

Во Flash Player 5 все escape-последовательности Unicode (`\u`) должны указывать на кодовые позиции в наборах символов Latin 1 или Shift-JIS; другие символы Unicode будут отображаться неправильно. Например, во Flash Player 5 код нашего примера не будет правильно показывать знак «евро».

Работа со строками

С помощью обработки строк можно запрограммировать что угодно – от проверки допустимости вводимых пользователем данных до игр со словами. Проявив некоторую изобретательность, можно реализовать интересные визуальные эффекты с текстом и другие интересные вещи.

Обрабатывать строки можно с помощью операторов и встроенных функций. Строковые операторы могут соединять вместе несколько строк или сравнивать символы двух строк. Встроенные функции могут изучать свойства и содержимое строк, извлекать часть строки, проверять кодовую позицию символа, создавать символы по их кодовой позиции, изменять регистр символов строки и даже преобразовывать строку в имя переменной или свойства.

Соединение строк

Соединение строк (создание новой строки из двух или более строк) называется *конкатенацией*. Как мы видели выше, две строки можно конкатенировать с помощью оператора «плюс» (+), например:

```
"Macromedia" + "Flash"
```

Этот код возвращает строку «MacromediaFlash». Ой! Мы забыли поместить пробел между словами. Чтобы добавить пробел, можно вставить его между кавычками, в которых задана одна из строк, например:

```
"Macromedia " + "Flash"  // Возвращает "Macromedia Flash"
```

Но это не всегда удобно. В большинстве случаев нежелательно добавлять пробел к имени компании или продукта. Поэтому мы соединим вместе три строки, средняя из которых состоит из одного пробела:

```
"Macromedia" + " " + "Flash"   // Тоже возвращает "Macromedia Flash"
```

Обратите внимание, что символ пробела не совпадает с пустой строкой, которую мы видели выше, потому что в пустой строке между кавычками нет никаких символов.

Обычно никто не конкатенирует литеральные строки, потому что проще записать результат одной строкой. В качестве примера, имеющего большее практическое значение, конкатенируем переменные, которые содержат строковые данные. Рассмотрим такой код:

```
var company = "Macromedia";
var product = "Flash";

// Присвоим переменной sectionTitle значение "Macromedia Flash"
var sectionTitle = company + " " + product;
```

В строках 1 и 2 мы запоминаем в переменных строковые значения. Затем мы соединяем эти значения через пробел. Два из наших строковых значений содержатся в переменных, а одно (пробел) – в строковом литерале. Проблем нет. Так часто бывает.

Иногда требуется добавить символы к существующей строке. Например, мы хотим создать индивидуальное приветственное сообщение для пользователя веб-сайта:

```
// Для данного примера жестко зашьем значение userName.
// (Обычно оно получается из текстового поля ввода или базы данных.)
var userName = "Karsten";
var greeting = "Welcome to GreatSite.com";  // Наше приветственное сообщение
greeting = greeting + " " + userName;       // Добавить к приветствию
                                            // имя пользователя
```

Это решает задачу, но обратите внимание, что нам пришлось дважды ссылаться на `greeting` в строке 2. Большую эффективность обеспечивает оператор +=, который добавляет строку, находящуюся справа от него, к строковой переменной, находящейся слева:

```
var greeting = "Welcome to GreatSite.com";  // Наше приветственное сообщение
greeting += (" " + username);               // Добавить к приветствию имя
                                            // пользователя
```

Изучая образцы кода для Flash 4, можно встретить устаревшие операторы конкатенации строк: & и *add*. Во Flash 5 и последующих версиях следует применять оператор +. Дополнительные сведения об обратной совместимости представлены в приложении C.

Функция concat()

Функция *concat( )* добавляет к строке символы, подобно +=. Поскольку *concat( )* представляет собой функцию, в ее вызове используется оператор точки:

```
var product = "Macromedia".concat(" Flash");
var sentence = "How are you";
var question = sentence.concat("?")
```

Однако будьте осторожны: в отличие от +=, функция *concat( )* не изменяет строку, к которой применяется, она лишь возвращает конкатенированное строковое значение. Для того чтобы с этим значением работать, необходимо присвоить его переменной или другому контейнеру данных. Изучите внимательно следующий код, чтобы понять разницу между += и *concat( )*:

```
// Здесь мы ошиблись, забыв сохранить значение, возвращаемое concat(),
// в переменной.
var userName = "Karsten";
var greeting = "Welcome to GreatSite.com";
greeting.concat(userName);
trace(greeting);  // Выводит "Welcome to GreatSite.com"
                  // concat() не изменила значение greeting
// Здесь мы правильно сохранили значение, возвращаемое concat(), в переменной.
finalGreeting = greeting.concat(" " + userName);
```

Функция *concat( )* может принимать несколько аргументов, то есть объединять несколько разделенных запятыми строк в одну строку:

```
var userName = "Karsten";
// Устанавливает finalGreeting в "Welcome to GreatSite.com, Karsten!"
finalGreeting = greeting.concat(", ", userName, "!");
```

что равносильно следующему:

```
finalGreeting = greeting;
finalGreeting += ", " + userName + "!";
```

Сравнение строк

Чтобы проверить, одинаковы ли две строки, используются операторы равенства (==) и неравенства (!=). Строки часто сравнивают при выполнении кода в зависимости от условия. Так, если пользователь вводит пароль, надо сравнить введенную им строку с фактическим паролем. Результат сравнения определяет поведение кода, например надо ли предоставить доступ к защищенной паролем функциональности.

Использование операторов равенства (==) и неравенства (!=)

Оператор равенства принимает два операнда – один слева, другой справа. Операнды могут быть строковыми литералами или любыми переменными, элементами массивов, свойствами объектов или выражениями, которые можно преобразовать в строку:

```
"hello" == "goodbye"       // Сравнить два строковых литерала
```

```
userGuess == "fat-cheeks"  // Сравнить переменную со строкой
userGuess == password      // Сравнить две переменные
```

Во Flash 4 имелись отдельные операторы для сравнения строк и нестроковых величин, однако во Flash 5 и далее операторы `==` и `!=` можно использовать для сравнения чисел и других типов данных (см. приложение C). Здесь мы ограничим обсуждение строками. Если операнд справа содержит те же самые символы в том же самом порядке, что и операнд слева, то две строки считаются равными, и результат представляет собой булево значение `true`.

Два отдельных символа считаются в ActionScript равными, только если они имеют одинаковую кодовую позицию Unicode. Это означает, что буквы верхнего и нижнего регистра (у которых разные кодовые позиции) не считаются равными. Аналогично буквы с диакритическими знаками и без них, например *ü* и *u*, имеют разные кодовые позиции и не считаются равными. Кроме того, ActionScript не считает символьные последовательности из нескольких глифов равными соответствующему одноглифовому символу. Например, *n~* не считается равным *c* (хотя в других языках они могут считаться равными).

Все следующие сравнения дают значение `false`:

```
"olive-orange" == "olive orange"   // Не равны
"nighttime" == "night time"        // Не равны
"Day 1" == "day 1"                 // Не равны
"n~" == "c"                        // Не равны
"ü " == "u"                        // Не равны
```

Поскольку результатом сравнения являются булевы значения `true` или `false`, можно использовать их в качестве проверочных выражений в условных инструкциях и циклах, например:

```
if (userGuess == password) {
  this.gotoAndStop("classifiedContent");
} else {
  this.gotoAndStop("entryFailed");
}
```

Если выражение (*userGuess == password*) имеет значение `true`, то выполняется инструкция *this.gotoAndStop("classifiedContent")*. Если выражение имеет значение `false`, секретное содержимое будет пропущено, а пользователь будет отправлен в кадр с сообщением об ошибке.

Далее в этой главе мы подробнее узнаем о булевых значениях, а условные инструкции мы изучим в главе 7.

Для проверки неравенства двух строк применяется оператор *неравенства*, дающий результат, противоположный оператору равенства. Например, следующие выражения представляют значения `false` и `true` соответственно:

```
"Jane" != "Jane"  // false, потому что две строки равны
"Jane" != "Biz"   // true, потому что две строки различны
```

Ниже мы используем оператор неравенства, чтобы выполнить некоторые действия, только если две строки не равны:

```
if (userGender != "boy") {
  // здесь идет код только для девочек...
}
```

Изучая код для Flash 4, можно встретить устаревшие операторы eq и ne, применявшиеся для проверки равенства и неравенства строк. Во Flash версии 5 и более поздних следует использовать операторы == и !=. В главе 5 рассматриваются операторы *строгого равенства* и *строгого неравенства*, осуществляющие сравнение без автоматического приведения типов.

Порядок символов и сравнение по алфавиту

Можно также сравнивать две строки, основываясь на порядке символов. Ранее мы видели, что каждому символу присвоен числовой код и что эти коды численно упорядочены в наборе символов (в случае ActionScript этим набором символов является Unicode). Можно узнать, который из символов следует раньше по порядку, с помощью операторов *сравнения*: «больше» (>), «больше или равно» (>=), «меньше» (<) и «меньше или равно» (<=). Все операторы сравнения сравнивают два операнда:

```
"a" < "b"      // true
"2" > "&"      // true
"r" <= "R"     // false
"$" >= "@"     // false
```

Подобно выражениям равенства и неравенства, выражения сравнения возвращают булевы значения true или false, в зависимости от соотношения между операндами. Каждый операнд должен возвращать строковое значение. Операторы сравнения можно применять для сравнения других типов данных, например чисел, но мы ограничим здесь изложение строковыми операндами.

Поскольку символы *A–Z* и *a–z* сгруппированы в наборах символов Unicode и Latin 1 в алфавитном порядке, то сравнение порядка символов часто используется для того, чтобы определить, которая из двух букв раньше следует в алфавите. Обратите, однако, внимание, что все прописные буквы в наборе Latin 1 предшествуют всем строчным буквам. Если забыть об этом, то можно получить неожиданные результаты:

```
"Z" < "a"       // Имеет значение true
"z" < "a"       // Имеет значение false
"Cow" < "bird"  // Имеет значение true
```

ActionScript не использует при сортировке символов алгоритм Unicode Collation Algorithm (UCA): символы сравниваются строго на основании кодовой позиции. UCA определяет правила для сравнения символов на основе национальных алфавитов с учетом регистра, диакритических знаков и многосимвольных последовательностей. Подробности можно найти по адресу:

http://www.unicode.org/unicode/reports/tr10

Рассмотрим каждый оператор сравнения более подробно; в следующих описаниях *сравниваемый символ* определяется как первый несовпадающий символ в двух операндах:

Больше (>)

Возвращает `true`, если сравниваемый символ левого операнда находится дальше в порядке символов Unicode, чем сравниваемый символ правого операнда. Если оба операнда полностью совпадают, > возвращает `false`:

```
"b" > "a"      // true
"a" > "b"      // false
"ab" > "ac"    // false (сравнивается второй символ)
"abc" > "abc"  // false (строки одинаковы)
"ab" > "a"     // true  (b является символом сравнения)
"A" > "a"      // false ("A" предшествует "a" в порядке символов)
```

Больше или равно (>=)

Возвращает `true`, если сравниваемый символ левого операнда находится дальше в порядке символов или если оба операнда полностью совпадают:

```
"b" >= "a"  // true
"b" >= "b"  // true
"b" >= "c"  // false
"A" >= "a"  // false ("A" и "a" имеют разные кодовые позиции)
```

Меньше (<)

Возвращает `true`, если сравниваемый символ левого операнда находится раньше в порядке символов Unicode, чем сравниваемый символ правого операнда. Если оба операнда полностью совпадают, < возвращает `false`:

```
"a" < "b"      // true
"b" < "a"      // false
"az" < "aa"    // false (сравнивается второй символ)
```

Меньше или равно (<=)

Возвращает `true`, если сравниваемый символ левого операнда находится раньше в порядке символов или если оба операнда полностью совпадают:

```
"a" <= "b"  // true
"a" <= "a"  // true
"z" <= "a"  // false
```

Чтобы определить, который из двух неалфавитных символов находится ближе к началу в наборе символов Latin 1 (его порядок соответствует порядку Unicode), обратитесь к приложению B. Узнать кодовую позицию любого символа в Unicode можно на *http://www.unicode.org/charts*.

В следующем примере мы определяем, является ли символ буквой латинского алфавита (а не цифрой, знаком пунктуации или еще каким-либо символом):

```
var theChar = "w";
if ((theChar >= "A" && theChar <= "Z") || (theChar >= "a" && theChar <= "z")) {
  trace("Символ принадлежит латинскому алфавиту.");
}
```

Обратите внимание, что логический оператор *ИЛИ* (||) позволяет проверить сразу два условия. Оператор *ИЛИ* рассмотрен в главе 5 «Операторы».

В коде для Flash 4 вам могут встретиться устаревшие операторы `gt`, `ge`, `lt` и `le`, применяемые для сравнения строк. Во Flash 5 и последующих версиях следует использовать операторы `>`, `>=`, `<` и `<=`.

Работа со встроенными строковыми функциями

За исключением функции *concat( )*, все рассмотренные нами средства работы со строками были операторами. Сейчас мы увидим, как использовать встроенные функции и свойства для осуществления более сложных действий со строками.

Чтобы применить к строке встроенную функцию, надо выполнить *вызов функции*, который имеет следующий вид:

```
string.functionName(arguments)
```

Вот пример применения функции *charAt( )* к `myString`:

```
myString.charAt(2)
```

Многие строковые функции возвращают новую строку, производную от исходной строки. Эти возвращаемые значения присваиваются переменным или свойствам объектов для использования в будущем, например:

```
thirdCharacter = myString.charAt(2);
```

Индекс символов

Многие строковые функции используют *индекс* символа – его числовую позицию относительно первого символа строки, начинающуюся с 0, а не с 1. Первый символ имеет номер 0, второй – номер 1, третий – номер 2 и т. д. Например, в строке «red» *r* имеет индекс 0, *e* имеет индекс 1, а *d* имеет индекс 2.

С помощью индексов символов задается часть строки. Можно, например, потребовать у интерпретатора получить символы с индекса 3 по индекс 7 либо узнать, какой символ находится по индексу 5.

Исследование строк

Можно исследовать строки или осуществлять поиск в них с помощью встроенного свойства `length` или функций *charAt( )*, *indexOf( )* и *lastIndexOf( )*.

Свойство length

Свойство `length` сообщает, сколько символов содержится в строке. Так как это свойство, а не функция, обращение к нему осуществляется без скобок и аргументов (прежняя функция Flash 4 *length( )* была объявлена устаревшей во Flash 5). Рассмотрим свойства `length` нескольких строк:

```
"Flash".length          // length равно 5
"skip intro".length     // length равно 10 (пробел является символом,
                        // считаются все символы)
```

```
"".length              // В пустой строке 0 символов
var axiom = "all that glisters will one day be obsolete";
axiom.length           // 42
```

Поскольку индексирование символов начинается с 0, индекс последнего символа всегда равен длине строки минус один.

Свойство строки `length` доступно для чтения, но не для записи. Нельзя удлинить строку таким способом:

```
axiom.length = 100;  // Чудесно, но не работает
```

Функция charAt()

Получить символ, находящийся в строке по любому индексу, можно с помощью функции *charAt( )*, которая имеет следующий вид:

```
string.charAt(index)
```

где *string* может быть любым литеральным строковым значением или идентификатором, содержащим строку, а *index* – это целое число или выражение, дающее в результате целое число и указывающее на позицию символа, который необходимо извлечь. Значение *index* должно находиться между 0 и *string*.length - 1. Если *index* выходит за пределы этого диапазона, то возвращается пустая строка. Вот несколько примеров:

```
"It is 10:34 pm".charAt(1)  // Возвращает "t", второй символ
var country = "Canada";
country.charAt(2);          // Возвращает "n", третий символ
var x = 4;
fifthLetter = country.charAt(x);                  // fifthLetter равно "d"
lastLetter = country.charAt(country.length - 1); // lastLetter равно "a"
```

Функция indexOf()

Функция *indexOf( )* применяется для поиска символов в строке. Если строка, в которой производится поиск, содержит искомую последовательность, то *indexOf( )* возвращает индекс (то есть позицию) ее первого вхождения в строку. В противном случае возвращается значение –1. Общий вид *indexOf( )* такой:

```
string.indexOf(character_sequence, start_index)
```

где *string* представляет собой любое литеральное строковое значение или идентификатор, содержащий строку; *character_sequence* – это строка, которую надо найти, и может быть литеральным строковым значением или идентификатором, содержащим строку, а *start_index* является начальной позицией для поиска (отсчет ведется с 0). Если *start_index* опущен, то поиск производится с начала строки *string*.

Давайте с помощью *indexOf( )* проверим, содержит ли строка символ *W*:

```
"GWEN!".indexOf("W");  // Возвращает 1
```

Да, *W* – второй символ в «GWEN!», поэтому мы получаем 1, индекс символа *W*. Вспомните, что нумерация индексов символов начинается с 0, поэтому второй символ имеет индекс 1.

Что произойдет, если мы будем искать символ *w* нижнего регистра? Посмотрим:

```
"GWEN!".indexOf("w");  // Возвращает -1
```

В строке «GWEN!» нет символа *w*, поэтому *indexOf( )* возвращает –1. Варианты буквы на нижнем и верхнем регистре – суть разные символы, а функция *indexOf( )* чувствительна к регистру!

Теперь проверим, есть ли символ @ в адресе электронной почты:

```
var email = "daniella2dancethenightaway.ca";  // Ой, кто-то забыл нажать Shift!
// Если знака @ нет, предупредить пользователя через текстовое поле formStatus_txt
if (email.indexOf("@") == -1) {
  formStatus_txt.text = "Адрес электронной почты неверен.";
}
```

Мы не ограничены поиском только одиночных символов. Можно искать в строке целую последовательность символов. Поищем «Canada» в адресе фирмы:

```
var iceAddress = "St. Clair Avenue, Toronto, Ontario, Canada";
iceAddress.indexOf("Canada");  // Возвращает 36, индекс буквы "C"
```

Обратите внимание, что *indexOf( )* возвращает позицию первого символа «Canada». Теперь сравним значение, возвращаемое выражением *iceAddress.indexOf("Canada")*, с –1 и присвоим результат переменной, в которой хранится страна размещения фирмы:

```
var isCanadian = iceAddress.indexOf("Canada") != -1;
```

Значением *iceAddress.indexOf("Canada") != –1* будет `true`, если *iceAddress.indexOf("Canada") не* равна –1 («Canada» найдена), и `false`, если *iceAddress.indexOf("Canada") равна* –1 («Canada» не найдена). Затем мы присваиваем это булево значение переменной `isCanadian`, которую можно использовать для создания формата почты, специфического для страны Северной Америки:

```
if (isCanadian) {
  mailDesc = "Please enter your postal code.";
} else {
  mailDesc = "Please enter your zip code.";
}
```

Посредством функции *indexOf( )* можно также определить, какую часть строки следует извлечь. Мы увидим, как это делается, когда изучим функцию *substring( )*.

Функция lastIndexOf()

Функция *indexOf( )*, как мы только что видели, возвращает местоположение *первого* вхождения последовательности символов в строку. Функция *lastIndexOf( )* возвращает местоположение *последнего* вхождения последовательности символов в строку или –1, если последовательность не найдена. Общий вид *lastIndexOf( )* точно такой же, как у *indexOf( )*:

```
string.lastIndexOf(character_sequence, start_index)
```

Единственное различие состоит в том, что поскольку *lastIndexOf( )* осуществляет поиск в обратном направлении, то *start_index* указывает на самый правый символ, который мы хотим включить в поиск, а не на самый левый. Если *start_index* опущен, он принимается равным string.length - 1 (последнему символу в строке).

Например:

```
paradox = "pain is pleasure, pleasure is pain";
paradox.lastIndexOf("pain");   // Возвращает 30; indexOf() возвратила бы 0
```

Следующий код возвращает 0 (индекс первого вхождения слова «pain»), потому что мы начинаем поиск в обратном направлении перед вторым вхождением «pain»:

```
paradox.lastIndexOf("pain",29);  // Возвращает 0
```

Регулярных выражений нет

Обратите внимание, что регулярные выражения (мощный инструмент поиска по шаблону в текстовых данных) по умолчанию не поддерживаются в ActionScript. Однако разработчик Пэвилс Юрьян (Pavils Jurjans) создал свой объект RegExp, который можно найти на *http://www.jurjans.lv/flash/RegExp.html*.

Извлечение частей строк

Иногда длинная строка содержит последовательность символов, к которой хотелось бы обращаться более удобным способом. Например, в строке «Steven Sid Mumby» нам может потребоваться фамилия, «Mumby». Чтобы извлечь более короткую строку (или *подстроку*), используется одна из следующих функций: *substring( )*, *substr( )*, *splice( )* или *split( )*. Сначала мы опишем работу каждой из них, а потом рассмотрим скорость их работы.

Функция substring()

Функция *substring( )* извлекает из строки последовательность символов, основываясь на индексах начального и конечного символов, и имеет следующий формат:

```
string.substring(start_index, end_index)
```

где *string* – любое литеральное строковое значение или идентификатор, который содержит строку, *start_index* – индекс первого символа, включаемого в подстроку, а *end_index* – символ, следующий *за* последним символом, который нужен нам в подстроке. Если *end_index* не задан, он принимается равным *string*.length. Следовательно:

```
var fullName = "Steven Sid Mumby";
middleName = fullName.substring(7, 10);  // Присваивает "Sid" переменной middleName
firstName = fullName.substring(0, 6);    // Присваивает "Steven"
                                         // переменной firstName
lastName = fullName.substring(11);       // Присваивает "Mumby" переменной lastName
```

В действительности нам может быть неизвестно, где начинаются и где заканчиваются имя, второе имя и фамилия, поэтому обычно отыскивают некоторый *разделитель*, например пробел, чтобы было проще определить, где находятся границы между словами (см. описание функции *split( )*, позволяющей сделать это еще проще). Вот как мы ищем последний пробел в строке, предполагая, что следующая после него часть строки является фамилией пользователя:

```
fullName = "Steven Sid Mumby";
lastSpace = fullName.lastIndexOf(" "); // Возвращает 10
// Предположительно, символы с 11 и до конца строки являются фамилией
lastName = fullName.substring(lastSpace+1);
trace ("Hello Mr. " + lastName);
```

Если *start_index* больше, чем *end_index*, оба аргумента автоматически меняются местами перед выполнением функции. Следующие два вызова функции дают одинаковый результат, но привыкать к применению *substring( )* с переставленными индексами не следует, потому что это затрудняет понимание кода:

```
fullName.substring(4, 6);  // Возвращает "en"
fullName.substring(6, 4);  // Возвращает "en"
```

Функция substr()

Функция *substr( )* извлекает из строки последовательность символов, основываясь на начальном индексе и длине (в отличие от *substring( )*, использующей начальный и конечный индексы). Общий формат *substr( )* следующий:

```
string.substr(start_index, subLength)
```

где *string*, как обычно, представляет собой любое литеральное строковое значение или идентификатор, содержащий строку; *start_index* – первый символ, который надо включить в подстроку; *subLength* указывает, сколько символов следует включить в нашу строку, начиная со *start_index* и ведя отсчет вправо. Если параметр *subLength* опущен, подстрока начинается с *start_index* и заканчивается последним символом исходной строки. Вот несколько примеров:

```
var fullName = "Steven Sid Mumby";
middleName = fullName.substr(7, 3);    // Присваивает "Sid" переменной middleName
firstName = fullName.substr(0, 6);     // Присваивает "Steven" переменной firstName
lastName = fullName.substr(11);        // Присваивает "Mumby" переменной lastName
```

Аргумент *start_index* можно задать относительно *конца* строки, если использовать отрицательное число. Последний символ соответствует –1, предпоследний соответствует –2 и т. д. Поэтому последние три примера *substr( )* можно переписать так:

```
middleName = fullName.substr(-9, 3);   // Присваивает "Sid" переменной middleName
firstName = fullName.substr(-16, 6);   // Присваивает "Steven" переменной firstName
lastName = fullName.substr(-5);        // Присваивает "Mumby" переменной lastName
```

Однако отрицательное значение *subLength* недопустимо.

Функция slice()

Подобно *substring( )*, функция *slice( )* извлекает из строки последовательность символов, основываясь на индексах начального и конечного символов. В то время как *substring( )* может задавать индексы только относительно *начала* исходной строки, *slice( )* может задавать их относительно начала или конца.

Функция *slice( )* имеет следующий формат:

```
string.slice(start_index, end_index)
```

где `string` – это любое литеральное строковое значение или идентификатор, содержащий строку, а `start_index` представляет собой индекс первого символа, включаемого в подстроку. Если `start_index` является положительным целым числом, то это обычный индекс символа, а если отрицательным целым числом, то эквивалентный индекс символа определяется отсчетом в обратном направлении от конца строки (то есть как `string.length + start_index`). Наконец, `end_index` является символом, следующим *за* последним символом, который надо включить в подстроку. Если `end_index` не задан, он принимается равным `string.length`. Если `end_index` отрицательный, то эквивалентный индекс символа определяется отсчетом в обратном направлении от конца строки (то есть как `string.length + end_index`).

При неотрицательных значениях индексов *slice( )* действует точно так же, как *substring( )*. Если же индексы принимают отрицательные значения, то следует помнить, что при этом не будет получена подстрока с символами, расположенными в обратном порядке (то есть строка, первым в которой будет символ с номером `end_index`, а вторым – с номером `start_index`). Вы просто задаете индексы относительно конца исходной строки. Помните также, что последний символ строки соответствует –1, а аргумент `end_index` задает символ *за* последним символом вашей подстроки, поэтому невозможно указать последний символ исходной строки с помощью отрицательного `end_index`. Внимательно посмотрите, каким образом отрицательные индексы используются для извлечения следующих подстрок:

```
var fullName = "Steven Sid Mumby";
middleName = fullName.slice(-9, -6);  // Присваивает "Sid" middleName
firstName = fullName.slice(-16, -10); // Присваивает "Steven" firstName
lastName = fullName.slice(-5, -1);    // Присваивает "Mumb" lastName:
                                      // не то, что нам нужно, но это лучшее,
                                     // что можно сделать с отрицательным end_index
lastName = fullName.slice(-5, 16)     // Присваивает "Mumby" lastName.
                                      // Обратите внимание на то, как сочетаются
                                      // отрицательный и положительный индексы.
lastName = fullName.slice(-5)         // Тоже присваивает "Mumby" lastName.
```

В следующем коде отрицательные индексы используются для проверки совпадения последних введенных в текстовое поле `guess_txt` букв с одним из специальных слов в массиве `specialWords`. Обратите внимание на применение обработчика *onChanged( )* объекта *TextField* для обнаружения ввода пользователем данных и метода *String.toLowerCase( )* для сравнения строк без уче-

та регистра. Подробности представлены в описании *TextField.onChanged( )* и *String.toLowerCase( )* в «Справочнике по языку».

```
var specialWords = ["MEGAN", "SING", "SUNDAY"];

guess_txt.onChanged = function () {
  for (var i=0; i < specialWords.length; i++) {
    var guess = this.text.slice(-specialWords[i].length).toLowerCase();
    var special = specialWords[i].toLowerCase();
    if (guess == special) {
      trace("The user entered a special word!");
      break;
    }
  }
}
```

Функция split()

Рассмотренные нами ранее функции извлечения строк возвращали только одну последовательность символов за одно обращение. Для того чтобы извлечь сразу много подстрок, можно воспользоваться мощной функцией *split( )*. (Она работает с массивами, поэтому кому-то, возможно, придется пропустить сейчас эту функцию и вернуться к ней после чтения главы 11.)

Функция *split( )* разбивает строку на серию подстрок и помещает эти подстроки в массив, который и возвращается. Функция *split( )* имеет следующий формат:

```
string.split(delimiter)
```

где *string* является любым литеральным строковым значением или идентификатором, который содержит строку, а *delimiter* представляет собой один или несколько символов, указывающих место расщепления строки *string*. Типичными разделителями служат запятые, пробелы и символы табуляции. Например, расщепить строку по каждой запятой можно так:

```
theString.split(",")
```

С помощью *split( )* можно без труда разбить предложение на отдельные слова. В описаниях функций *substring( )*, *substr( )* и *slice( )* каждую часть имени из строки «Steven Sid Mumby» приходилось вытаскивать вручную. Посмотрите, насколько легче это делается при помощи функции *split( )* с пробелом (" ") в качестве разделителя:

```
var fullName = "Steven Sid Mumby";
var names = fullName.split(" ");  // Это совсем просто!
// Теперь присвоим имена из массива отдельным переменным
firstName  = names[0];
middleName = names[1];
lastName   = names[2];
```

Эффективность извлечения подстрок во Flash 5

Функции Flash 5 *substr( )* и *slice( )* фактически реализованы путем наведения внешнего лоска на старую функцию *substring( )* из Flash 4 и потому выполняются немного дольше. Разница в скорости составляет миллисекунды, но при интенсивной обработке строк может стать ощутимой. При выполнении многократно повторяемых операций во Flash 5 для повышения производительности используйте функцию Flash 4 *substring( )*, например:

```
fullName = "Steven Sid Mumby";
// Присвоить "Sid" middleName с помощью функции Flash 5 substr()
middleName = fullName.substr(7, 3);
// Присвоить "Sid" middleName с помощью функции Flash 4 substring().
// Обратите внимание, что во Flash 4 индекс символа начинается с 1.
middleName = substring(fullname, 8, 3);
```

В интересах тех, кто работает во Flash 5, Брэнден Холл (Branden Hall) переписал методы класса *String*, чтобы повысить скорость их работы. См. файл по адресу:

http://chattyfig.figleaf.com/~bhall/code/string.as

Во Flash Player 6 эффективность обработки строк была заметно улучшена, поэтому необходимости в таком приеме в этой версии нет.

Сочетание исследования строки с извлечением подстроки

Мы посмотрели, как искать в строке символы и как извлекать символы из строки. Эти две задачи можно эффективно объединить.

В большинстве рассмотренных примеров в качестве аргументов выступали литеральные выражения, например:

```
var msg = "Welcome to my website!";
var firstWord = msg.substring(0, 7);  // 0 и 7 суть числовые литералы
```

Это хорошая демонстрация работы *substring( )*, но на практике *substring( )* обычно применяется иначе. Чаще нам не известно заранее содержимое строки, и аргументы должны генерироваться динамически. Так, обычно вместо чего-то статического вроде «вернуть подстроку с индекса 0 по индекс 7» мы имеем дело с изменяющимися фразами типа «вернуть подстроку, начиная с первого символа, и до первого пробела в этой строке». Этот подход характеризуется большей гибкостью и не требует предварительного знания содержимого строки. Извлечем первое слово из переменной `msg`, объединив *substring( )* с *indexOf( )*:

```
var firstWord = msg.substring(0, msg.indexOf(" "));
```

Выражение *msg.indexOf(" ")* вычисляет индекс первого пробела в `msg`. Наш метод действует независимо от того, где находится пробел. Это позволяет работать со строками, изменяющимися во время работы программы, и избежать подсчета символов, при котором легко допустить ошибку.

Сочетания исследования строки с ее извлечением могут быть практически бесконечными. В примере 4.3 извлекается второе слово переменной msg без жесткого кодирования индексов символов. На обычном языке это формулируется так: «извлечь из msg подстроку, начиная с символа после первого пробела и заканчивая символом перед вторым пробелом». Мы сохраняем местонахождение первого и второго пробела в виде переменных, что делает более очевидным происходящее.

Пример 4.3. Извлечение второго слова строки

```
var msg = "Welcome to my website!";
firstSpace = msg.indexOf(" ");                    // Найти первый пробел
secondSpace = msg.indexOf(" ", firstSpace + 1);  // Найти второй пробел

// Теперь извлечь второе слово
var secondWord = msg.substring(firstSpace + 1, secondSpace);
```

Преобразование регистра символов

Символы строки можно переводить в верхний или нижний регистр с помощью встроенных функций *toUpperCase( )* и *toLowerCase( )*. Обычно они применяются для форматирования строк при выводе и для сравнения строк, в которых есть символы разных регистров.

Функция toUpperCase()

Функция *toUpperCase( )* преобразует все символы строки к верхнему регистру (то есть в прописные буквы) и возвращает результат преобразования. Если для некоторого символа нет варианта на верхнем регистре, он возвращается в неизменном виде. Общий формат *toUpperCase( )* такой:

```
string.toUpperCase()
```

где *string* – это любое литеральное строковое значение или идентификатор, содержащий строку. Вот несколько примеров:

```
"listen to me".toUpperCase();       // Возвращает строку "LISTEN TO ME"
var msg1 = "Your Final Score: 234";
var msg2 = msg1.toUpperCase();      // Присваивает msg2
                                    // значение "YOUR FINAL SCORE: 234"
```

Обратите внимание, что функция *toUpperCase( )* не изменяет исходную строку, к которой применена, а возвращает копию этой строки в верхнем регистре. Следующий пример иллюстрирует это различие:

```
var msg = "Forgive me, I forgot to bring my spectacles.";
msg.toUpperCase();
trace(msg);    // Выводит: "Forgive me, I forgot to bring my
               // spectacles." Значение msg не изменилось при вызове toUpperCase()
```

Функция toLowerCase()

Функция *toLowerCase( )* переводит символы строки из верхнего в нижний регистр, например:

```
// Присвоить normal значение "this sentence has mixed caps!"
normal = "ThiS SenTencE Has MixED CaPs!".toLowerCase();
```

Чтобы сравнить две строки без учета регистра, переведите их обе в одинаковый регистр. Например:

```
if (userEntry.toLowerCase() == password.toLowerCase()) {
  // Предоставить доступ
}
```

Обе функции – и *toUpperCase( )* и *toLowerCase( )* – позволяют записать слово с прописной буквы:

```
var word = "fLASH";
var firstLetter = word.charAt(0).toUpperCase();
var remainderOfWord = word.substring(1).toLowerCase();
initialCapWord = firstLetter + remainderOfWord;
trace(initialCapWord);  // Выводит: Flash
```

В примере 4.4 показано, как с помощью функций строк и преобразования регистра создать анимационный текст в текстовом поле. Здесь ко всем текстовым полям добавляется новый метод *caseAni( )*. Прикрепите сценарий к кадру 1 в новом фильме и посмотрите, что произойдет. Учтите, что в этом коде используется несколько более сложных приемов, с которыми мы еще не сталкивались. Вернитесь к его изучению после чтения глав 9 и 12.

Пример 4.4. Анимация с помощью регистра символов

```
// Добавить новый метод caseAni() во встроенный класс TextField.
TextField.prototype.caseAni = function (msg) {
  // Сохранить ссылку на текущее текстовое поле в tf. Нам нужно будет
  // обратиться к нему из встроенной функции moveUppercaseLetter().
  var tf = this;
  tf.upperIndex = 0;  // Текущее местоположение заглавной буквы.
  tf.msg = msg;       // Анимируемое сообщение.

  // Начать анимацию путем вызова функции
  // moveUppercaseLetter() каждые 100 миллисекунд.
  var id = setInterval(moveUppercaseLetter, 100);

  // Определить функцию moveUppercaseLetter().
  // Она производит все манипуляции со строкой.
  function moveUppercaseLetter () {
    // Разрезать строку msg на три части и выделить среднюю верхним регистром.
    var part1 = tf.msg.slice(0, tf.upperIndex);
    var part2 = tf.msg.charAt(tf.upperIndex);
    var part2 = part2.toUpperCase();
    var part3 = tf.msg.slice(tf.upperIndex+1, tf.msg.length);
    tf.msg  = part1 + part2 + part3;

    // Вывести форматированную строку msg.
    tf.text = tf.msg;

    // Вернуть строку в исходное состояние.
    tf.msg  = tf.msg.toLowerCase();
```

```
      // Переместить выделение на одну букву вправо.
      tf.upperIndex++;

      // Если заглавная буква находится в конце строки, переместить ее снова в начало.
      if (tf.upperIndex > (tf.msg.length - 1)) {
        tf.upperIndex = 0;
      }
    }
  }

  // Опробуем с экземпляром текстового поля!
  this.createTextField("msgOutput_txt", 1, 0, 0, 300, 30);
  // Использовать моноширинный шрифт...
  msgOutput_txt.setNewTextFormat(new TextFormat("_typewriter", 16));
  msgOutput_txt.caseAni("my what fat cheeks you have");
```

Функции кода символов

Ранее в этой главе, в разделе «Escape-последовательности в стиле Unicode» мы узнали, как вставлять в строку символы в виде escape-последовательностей. ActionScript содержит также две встроенные функции для работы с кодами символов строк: *fromCharCode( )* и *charCodeAt( )*.

Функция fromCharCode()

Любой символ или последовательность символов могут быть созданы с помощью функции *fromCharCode( )*. В отличие от других строковых функций, *fromCharCode( )* вызывается не со строковым литералом или идентификатором, содержащим строку, а как метод особого объекта *String* в следующем виде:

```
String.fromCharCode(code_point1, code_point2, ...)
```

Вызов *fromCharCode( )* всегда начинается со *String.fromCharCode*. Затем в качестве аргументов передаются одна или более кодовых позиций создаваемых символов. В отличие от escape-последовательностей в стиле Unicode, которые должны быть заданы в шестнадцатеричном виде, кодовые позиции в *fromCharCode( )* можно задавать и в десятичном, и в шестнадцатеричном виде. Если вы незнакомы с шестнадцатеричными числами, то вам, может быть, удобнее применять *fromCharCode( )* вместо escape-последовательностей в стиле Unicode. Вот несколько примеров:

```
// Присвоить lastName значение "moock"
lastName = String.fromCharCode(109, 111, 111, 99, 107);
// Для сравнения сделаем то же с помощью escape-последовательностей в стиле Unicode
lastName = "\u006D\u006F\u006F\u0063\u006B";
// Создать символ copyright
copyNotice = String.fromCharCode(169) + " 2003";
// Создать знак "евро"
price = "5.99 " + String.fromCharCode(0x20AC);
```

Изучая код Flash 4, вы можете столкнуться с устаревшими функциями создания символов *chr( )* и *mbchr( )*. Во Flash 5 и более поздних версиях следует применять функцию *fromCharCode( )*.

Функция charCodeAt()

Для определения кодовой позиции любого символа строки применяется функция *charCodeAt( )*, которая имеет следующий формат:

```
string.charCodeAt(index)
```

где *string* – это любое литеральное строковое значение или идентификатор, содержащий строку, а *index* представляет позицию исследуемого символа. Функция *charCodeAt( )* возвращает десятичное целое, соответствующее кодовой позиции в Unicode символа с указанным индексом. Для значений, меньших 128, *charCodeAt( )* по существу преобразует символ в эквивалентный ASCII-код. Например:

```
var msg = "A is the first letter of the Latin alphabet.";
trace(msg.charCodeAt(0));  // Выводит: 65 (кодовая позиция "A")
trace(msg.charCodeAt(1));  // Выводит: 32 (кодовая позиция пробела)
```

Обычно *charCodeAt( )* применяется для работы со строками, содержащими символы, которые нельзя непосредственно ввести с клавиатуры. Так, следующий код проверяет, не является ли символ знаком копирайта:

```
msg = String.fromCharCode(169) + " 2000";
if (msg.charCodeAt(0) == 169) {
  trace("The first character of msg is a copyright symbol.");
}
```

Изучая код Flash 4, вы можете столкнуться с устаревшими функциями кодовых позиций *ord( )* и *mbord( )*. Во Flash 5 и более поздних версиях следует использовать функцию *charCodeAt( )*.

Выполнение кода в строке с помощью eval

В ActionScript функция *eval( )* преобразует строку в идентификатор (то есть имя переменной). Но чтобы до конца понять функцию ActionScript *eval( )*, надо разобраться, как работает аналогичная функция JavaScript *eval( )*. В JavaScript *eval( )* представляет собой встроенную функцию верхнего уровня, которая преобразует любую строку в блок кода и выполняет этот блок кода. Синтаксис JavaScript *eval( )* такой:

```
eval(string)
```

Когда *eval( )* выполняется в JavaScript, интерпретатор преобразует *string* в код, выполняет этот код и возвращает результирующее значение (если оно генерируется). Рассмотрим следующие примеры JavaScript:

```
eval("parseInt('1.5')");  // Вызывает функцию parseInt(), которая возвращает 1
eval("var x = 5");        // Создает новую переменную с именем x
                          // и присваивает ей значение 5
```

Если раньше вы никогда не встречались с *eval( )*, то можете подумать: «Зачем мне нужна строка, содержащая код? Почему просто не написать код?» Потому что *eval( )* позволяет динамически создавать код, когда это требуется.

Например, с помощью *eval( )* можно создавать функции, назначаемые в качестве обработчиков событий для групп аналогичных объектов.

Функция ActionScript *eval( )* поддерживает малую часть возможностей родственной ей функции в JavaScript: она работает только тогда, когда ее аргумент является идентификатором. Поэтому функция ActionScript *eval( )* может только извлекать данные, связанные с указанным идентификатором, например:

```
var num = 1;
var person1 = "Eugene";
trace (eval("person" + num));  // Выводит: "Eugene"
```

Но даже в этом урезанном виде *eval( )* весьма полезна. Вот как мы динамически создаем в цикле клипы. Мы помещаем клипы в массив и обращаемся к ним, вызывая фунцию *eval( )*:

```
for (var i = 0; i < 10; i++) {
  duplicateMovieClip("ballParent", "ball" + i, i);
  balls[i] = eval("ball" + i);
}
```

Учтите, однако, что почти всегда вместо *eval( )* можно воспользоваться оператором доступа к массиву, создающим динамические ссылки на клипы. Например:

```
duplicateMovieClip("ballParent", "ball" + i , i);
balls[ballCount] = _root ["ball" + i];
```

Во Flash 4 *eval( )* часто применялась для моделирования массивов путем динамического генерирования переменных, поскольку поддержка массивов появилась только во Flash 5. При модификации старого кода следует заменять такое использование *eval( )* массивами.

Булев тип

Булевы данные используются для представления логических состояний истинности и ложности. Поэтому есть только два допустимых значения *булева* типа данных: `true` и `false`. Заметьте, что слова `true` и `false` не заключаются в кавычки, поскольку булевы данные *не* являются строковыми. Ключевые слова `true` и `false` являются зарезервированными значениями элементарных данных и не могут выступать в качестве имен переменных или идентификаторов.

Булевы значения служат для создания логики выполнения кода. Например, можно присвоить значение `true` переменной, которая хранит состояние огневой мощи космического корабля:

```
shipHasDoubleShots = true;
```

Сравнивая `shipHasDoubleShots` с булевым литералом `true`, мы можем решить, каков объем повреждений, нанесенных выстрелом, достигшим цели:

```
if (shipHasDoubleShots == true) {
  // Обстрел удвоенной мощности.
  // Выполняется при истинности сравнения.
} else {
  // Обстрел одинарной мощности.
  // Выполняется при ложности сравнения.
}
```

Когда удвоенная огневая мощь израсходована, можно установить переменную в `false`:

```
shipHasDoubleShots = false;
```

В результате чего выражение *shipHasDoubleShots == true* получит значение `false`, и при поражении цели выстрелом будет выполняться сценарий с разрушениями, соответствующими одинарной мощности.

Все операторы сравнения выражают свои результаты в булевых величинах. Когда мы спрашиваем: «Совпадает ли ввод пользователя с истинным паролем?», ответ дается в виде булева значения:

```
// userGuess == password возвращает true или false
if (userGuess == password) {
  gotoAndStop("secretContent");
}
```

А если мы спрашиваем: «Повернут ли этот клип более чем на 90°?», то ответ снова будет иметь булев тип:

```
// theClip_mc._rotation > 90 возвращает true или false
if (theClip_mc._rotation > 90) {
// Уменьшить видимость клипа, если он повернут более чем на 90 градусов
  theClip_mc._alpha = 50;
}
```

Многие внутренние свойства и методы ActionScript описывают среду фильмов Flash в булевых терминах. Так, на вопрос: «Нажата ли клавиша пробела?» интерпретатор ответит булевым `true` (да) или `false` (нет):

```
// Функция Key.isDown() возвращает true или false
if (Key.isDown(Key.SPACE)) {
  // Пробел нажат, поэтому космический корабль открывает огонь
}
```

В главе 5 мы узнаем, как с помощью булевых операторов формулировать сложные логические выражения.

Создание предзагрузчика с помощью булевых выражений

Рассмотрим прикладной пример с булевыми выражениями. Допустим, у нас есть документ с 500 кадрами и богатым содержанием. Вступление нашего фильма – это кадр 20 с меткой `intro`. Мы поместим в кадр 2 главной временной диаграммы фильма следующий код:

```
if (this._framesloaded >= this._totalframes) {
  this.gotoAndPlay("intro");
} else {
  this.gotoAndPlay(1);
}
```

При просмотре фильма воспроизводящая головка минует кадр 1 и входит в кадр 2. Интерпретатор ActionScript доходит до условной инструкции и вычисляет булево выражение *this._framesloaded >= this._totalframes*. Пока фильм еще загружается, `_this._framesloaded` меньше количества кадров в фильме (`this._totalframes`). Если `this._framesloaded` меньше или равно `this._totalframes`, то выражение *this._framesloaded >= this._totalframes* имеет значение `false`. Поэтому инструкция *this.gotoAndPlay("intro")* пропускается, а вместо нее выполняется *this.gotoAndPlay(1)*. Инструкция *this.gotoAndPlay(1)* направляет воспроизводящую головку обратно в кадр 1 и проигрывает фильм. Когда воспроизводящая головка входит в кадр 2, наш код выполняется *снова*. Головка осуществляет это циклическое движение, пока значением выражения *this._framesloaded >= this._totalframes* не станет `true` (то есть пока не будут загружены все кадры). В этот момент будет выполнена инструкция *this.gotoAndPlay("intro")*, которая отправляет головку воспроизведения на метку `intro`. Оттуда можно спокойно начать наш фильм, потому что все кадры загружены.

Ура! Вы создали предзагрузчик на основе булева выражения. Это уже немало. Значительно больше об условных инструкциях и управлении фильмами с помощью булевых выражений мы узнаем в главе 7.

Тип undefined

Большинство изученных до сих пор типов данных использовались для хранения и обработки информации. Тип данных *undefined* имеет более узкое применение: с его помощью проверяют, существует ли переменная и присвоено ли ей значение. Тип данных *undefined* имеет только одно допустимое значение – элементарное значение `undefined`.

Когда мы впервые определяем переменную, ей по умолчанию присваивается значение `undefined`:

```
var velocity;
```

Для интерпретатора предыдущая инструкция означает:

```
var velocity = undefined;
```

Для того чтобы проверить, есть ли у переменной значение, можно сравнить переменную с `undefined`, например:

```
if (myVariable != undefined) {
  // myVariable имеет значение, поэтому продолжить выполнение...
}
```

Удалять переменную или свойство объекта следует посредством оператора *delete*, а не присваивать значение undefined переменной или свойству. Присваивание undefined не удалит свойство, а лишь изменит его значение. Удаление свойства уничтожает его полностью.

Обратите внимание, что значение undefined, выступающее в качестве строки, преобразуется в пустую строку. Например, в следующем коде мы создаем две переменные, firstName и lastName, конкатенируем их и записываем результат в переменную fullName. У переменной firstName нет значения (ее значение undefined), поэтому она преобразуется в "".

```
var firstName;
var lastName = "Moock";
var fullName = firstName + lastName;  // Присваивает fullName значение "Moock"
```

В JavaScript значение undefined в строковом контексте преобразуется в строку «undefined», поэтому предшествующий код в JavaScript присвоил бы переменной fullName значение «undefinedMoock». ActionScript преобразует undefined в "" в целях обратной совместимости. Однако во Flash MX и последующих версиях при выводе значения undefined в окно Output функцией *trace( )* появляется слово «undefined», благодаря чему undefined легче обнаружить при отладке.

Тип *undefined* не был реализован в ActionScript для Flash 4, поэтому многие программы Flash 4 проверяли, содержит ли переменная полезное значение, сравнивая ее с пустой строкой. Часто встречался такой код:

```
if (myVar eq "") {
  // Пока ничего не делать: myVar не определена
}
```

Если бы Flash преобразовывал undefined в строковом контексте во что-либо отличное от "", то такой старый код не работал бы в проигрывателе Flash 5 и более поздних.

Обратите внимание, что ActionScript возвращает undefined как для несуществующих переменных, так и для переменных, которые объявлены, но не имеют значения. Это тоже отход от JavaScript, где ссылки на несуществующие переменные вызывают ошибку.

Тип null

Тип *null* почти идентичен типу *undefined*. Как и *undefined*, тип данных *null* применяется для представления отсутствия данных и имеет только одно допустимое значение, элементарное значение null. Однако значение null не присваивается интерпретатором автоматически, это делаем намеренно мы сами.

Значение null присваивается переменной, элементу массива или свойству объекта, чтобы показать, что указанный контейнер данных не содержит допустимое число, строку, булево значение, массив или объект. Например,

можно присвоить начальное значение null свойству объекта, чтобы показать, что оно существует, но еще не имеет полезного значения.

Чтобы удалить переменную или свойство объекта, следует воспользоваться оператором *delete*, а не присваивать значение null переменной или свойству.

Обратите внимание, что при сравнении null может быть равным только самому себе и undefined:

```
null == undefined;   // true
null == null;        // true
```

Чтобы гарантировать, что переменная или свойство содержит null (а не undefined), проверьте их тип с помощью оператора *typeof*:

```
var x = null;
if (typeof x == "null") {
  trace("x is null");
}
```

Во Flash Player 6 есть другая возможность – оператор строгого равенства (===):

```
if (x === null) {
  trace("x is null");
}
```

Хотя null и undefined сходны между собой, их роль в программах ActionScript различна. При помощи значения null программисты указывают на то, что переменная или свойство оставлены неопределенными намеренно. Добавим, что значение undefined присваивается интерпретатором автоматически, когда нет другого значения. Рассмотрим приложение для фотоальбома, которое последовательно показывает ряд графических изображений. Приложение инициализируется переменными title (название) и description (описание). Если мы решим пренебречь описанием некоторого альбома, то установим description равным null.

```
var title = "My Holiday Photos";
var description = null;
```

Последующий код инициализации может тогда определить, что значение description опущено умышленно.

```
// Если есть допустимое описание...
if (description !== null) {
  // ...создать текстовое поле описания.
}
```

Допустим, что вместо null мы используем undefined:

```
var title = "My Holiday Photos";
var description = undefined;
```

Тогда код инициализации не может определить, было ли значение description установлено интерпретатором автоматически или мы сделали это намеренно. Разница тонкая, но важная в больших программах. Мы расширим

прежний код инициализации, чтобы он выводил сообщение об ошибке, когда не задано описание.

```
// Если есть допустимое описание...
if (description === undefined) {
  trace("WARNING: No description value was specified!");
} else if (description !=== null) {
  // ... создать текстовое поле описания.
}
```

Различая `undefined` и `null` в нашем коде инициализации, мы можем обеспечить точную передачу данных для отладки и обнаружить возможный недосмотр программиста. Другой пример практического применения `null` можно найти в описании параметров конструктора класса *TextFormat* в «Справочнике по языку».

Вперед!

Мы уже изучили значительно больше, чем вам может показаться. Мы построили прочный фундамент, который позволит разобраться в более сложных темах, ожидающих нас в следующих главах. Вы пока не постигли всех тонкостей, но не беспокойтесь. Когда понадобится, вы вспомните все, чему учились. Сохраняйте уверенность во время чтения следующей главы, в которой вы узнаете о том, как сливать вместе и преобразовывать данные. Затем мы займемся некоторыми более полноценными практическими примерами.

5

Операторы

Оператор (*operator*) представляет собой символ или ключевое слово и обрабатывает, объединяет или трансформирует данные. Начинающим программистам покажутся очень знакомыми некоторые математические операторы, такие как + (сложение) и – (вычитание). В остальных случаях им придется изучить особый синтаксис программирования. Например, для перемножения двух чисел в ActionScript используется символ * (оператор умножения) вместо ×, знакомого всем по средней школе, например 5, умноженное на 6:

```
5 * 6;
```

Общие характеристики операторов

У каждого оператора своя специальная задача, но у них есть ряд общих характеристик. Прежде чем говорить об операторах по отдельности, посмотрим, как они ведут себя в целом.

Операторы и выражения

Операторы выполняют некоторые действия над передаваемыми им значениями данных (*операндами*). Например, в операции 5 * 6 числа 5 и 6 являются *операндами* оператора умножения (*). Операнды могут быть любыми выражениями, например:

```
total = player1score + bonusScore;              // Операнды являются переменными
answer = (x + y) - (Math.PI * radius * radius);  // Операнды являются
                                                 // сложными выражениями
```

Обратите внимание, что во втором примере левый и правый операнды оператора «-» представляют собой выражения, которые сами содержат другие операции. При помощи составных выражений можно создавать более сложные, например:

```
answer = ((x + y) - (Math.PI * radius * radius)) / 2          // Разделить все на 2
```

Если выражения становятся очень большими, то можно сохранять промежуточные результаты в переменных. Это и удобнее, и яснее. Старайтесь давать переменным содержательные имена, например:

```
var radius = 10;
var height = 25;
var circleArea = (Math.PI * radius * radius);
var cylinderVolume = circleArea * height;
```

Количество операндов

Операторы иногда делятся на категории по количеству принимаемых операндов. Некоторые операторы ActionScript принимают один операнд, некоторые два, а один – даже три:

```
-x                                        // Один операнд
x * y                                     // Два операнда
(x == y) ? "true result" : "false result"  // Три операнда
```

Операторы с одним операндом называются *унарными*; операторы, принимающие два операнда, называются *бинарными*, а операторы с тремя операндами – *тернарными*. Для наших задач будем группировать операторы в зависимости от производимых ими действий, а не по количеству принимаемых операндов.

Приоритеты операторов

Приоритеты операторов определяют порядок выполнения операций в выражении с несколькими операторами. Например, если в выражении есть умножение и сложение, то первым выполняется умножение:

```
4 + 5 * 6  // Возвращает 34, потому что 4 + 30 = 34
```

Выражение `4 + 5 * 6` вычисляется как `4 + (5 * 6)`, потому что оператор `*` имеет более высокий приоритет, чем оператор `+`. Аналогично, когда в одном и том же выражении встречаются операторы «меньше» (`<`) и конкатенации (`+`), сначала выполняется конкатенация. Например, следующее предложение выводит `false` в окно Output (что вызовет удивление, если вы не знаете, какие приоритеты у операторов `<` и `+`):

```
trace("result: " + "a" < "b");
```

Сначала «result:» конкатенируется с «a», создавая новую строку «result: a». Затем эта новая строка сравнивается с «b», что дает `false`. Для того чтобы операции выполнялись в требуемом порядке, следует расставить круглые скобки:

```
trace("result: " + ("a" < "b"));  // Выводит: result: true
```

При возникновении сомнений или необходимости задать иной порядок операций используйте круглые скобки, которые имеют наивысший приоритет:

```
(4 + 5) * 6  // Дает 54, потому что 9 * 6 = 54
```

Даже если в скобках нет строгой необходимости, они могут облегчить чтение сложного выражения. Выражение:

```
// x больше, чем y или y равно z
x > y || y == z
```

может оказаться трудным для понимания без обращения к таблице приоритетов. Значительно легче прочесть его, если добавить скобки:

```
(x > y) || (y == z)  // Куда лучше!
```

В табл. 5.1 приведены приоритеты всех операторов. В первую очередь выполняются операторы с наивысшим приоритетом (вверху таблицы). Операторы с одинаковым приоритетом выполняются в порядке их появления в выражении, обычно слева направо, если только не установлена ассоциативность справа налево (см. колонку «Ассоциативность» в табл. 5.1). Подробная информация представлена в следующем разделе «Ассоциативность операторов».

Таблица 5.1. Ассоциативность и приоритет операторов ActionScript

Оператор	Приоритет	Ассоциативность	Описание
.	15	слева направо	доступ к свойству объекта
[]	15	слева направо	доступ к элементу массива
()	15	слева направо	круглые скобки
function()	15	слева направо	вызов функции
x++	14	слева направо	постфиксное инкрементирование
x--	14	слева направо	постфиксное декрементирование
++x	14	справа налево	префиксное инкрементирование
--x	14	справа налево	префиксное декрементирование
-	14	справа налево	унарное отрицание
~	14	справа налево	поразрядное отрицание
!	14	справа налево	логическое отрицание
new	14	справа налево	создание объекта/массива
delete	14	справа налево	удаление объекта/свойства/элемента массива
typeof	14	справа налево	определение типа данных
void	14	справа налево	возврат неопределенного значения
*	13	слева направо	умножение
/	13	слева направо	деление
%	13	слева направо	деление по модулю
+	12	слева направо	сложение или конкатенация строк
-	12	слева направо	вычитание
<<	11	слева направо	поразрядный сдвиг влево

Таблица 5.1 (продолжение)

Оператор	Приоритет	Ассоциативность	Описание
`>>`	11	слева направо	поразрядный сдвиг вправо со знаком
`>>>`	11	слева направо	поразрядный сдвиг вправо без знака
`<`	10	слева направо	меньше
`<=`	10	слева направо	меньше или равно
`>`	10	слева направо	больше
`>=`	10	слева направо	больше или равно
`instanceof`	10	слева направо	проверка класса объекта
`==`	9	слева направо	равенство
`!=`	9	слева направо	неравенство
`===`	9	слева направо	строгое равенство
`!==`	9	слева направо	строгое неравенство
`&`	8	слева направо	поразрядное *AND* (*И*)
`^`	7	слева направо	поразрядное *XOR* (исключающее *ИЛИ*)
`|`	6	слева направо	поразрядное *OR* (*ИЛИ*)
`&&`	5	слева направо	логическое *AND* (*И*)
`||`	4	слева направо	логическое *OR* (*ИЛИ*)
`?:`	3	справа налево	условный оператор
`=`	2	справа налево	присваивание
`+=`	2	справа налево	сложение с присваиванием
`-=`	2	справа налево	вычитание с присваиванием
`*=`	2	справа налево	умножение с присваиванием
`/=`	2	справа налево	деление с присваиванием
`%=`	2	справа налево	деление по модулю с присваиванием
`<<=`	2	справа налево	поразрядный сдвиг влево с присваиванием
`>>=`	2	справа налево	поразрядный сдвиг вправо с присваиванием
`>>>=`	2	справа налево	поразрядный сдвиг вправо (без знака) с присваиванием
`&=`	2	справа налево	поразрядное *AND* (*И*) с присваиванием
`^=`	2	справа налево	поразрядное *XOR* с присваиванием
`|=`	2	справа налево	поразрядное *OR* (*ИЛИ*) с присваиванием
`,`	1	слева направо	запятая

Ассоциативность операторов

Как мы только что узнали, приоритет операторов указывает на порядок их действия: имеющие более высокий приоритет выполняются раньше тех, чей приоритет ниже. Но что происходит, если несколько операторов имеют одинаковый приоритет? В этом случае применяются правила *ассоциативности операторов*, указывающие, в каком направлении должна осуществляться операция. Операторы могут быть ассоциативными слева (выполняются слева направо) либо ассоциативными справа (выполняются справа налево). Рассмотрим, например, такое выражение:

```
a = b * c / d
```

Операторы `*` и `/` ассоциативны слева, поэтому первой выполняется операция `*` над значениями слева `(b * c)`. Этот пример эквивалентен следующему:

```
a = (b * c) / d
```

Напротив, оператор `=` (присваивания) ассоциативен справа, поэтому выражение:

```
a = b = c = d
```

означает «присвоить значение `d` переменной `c`, затем присвоить значение `c` переменной `b` и, наконец, присвоить значение `b` переменной `a`», что равносильно следующему:

```
a = (b = (c = d))
```

Ассоциативность операторов определяется достаточно интуитивно, но если вы получили неожиданное значение сложного выражения, сверьтесь с табл. 5.1 или добавьте лишние скобки. В данной главе мы обратим внимание на случаи, в которых ассоциативность часто становится источником ошибок.

Типы данных и операторы

Некоторые операторы могут принимать в качестве операндов данные разных типов. В зависимости от типа данных операнда действие оператора может быть различным. Например, оператор `+` выполняет сложение, если применяется к числовым операндам, и конкатенацию, если к строковым. Если операнды имеют разные или неверные типы, ActionScript осуществляет преобразование типов согласно правилам, описанным в главе 3, что может оказать серьезное влияние на ваш код.

Оставшаяся часть главы содержит справочные описания каждого оператора ActionScript. Программистам-новичкам, возможно, придется пропустить темы, о которых подробно еще не говорилось.

Оператор присваивания

Мы уже неоднократно использовали оператор присваивания. Он может помещать значение в переменную, элемент массива или свойство объекта. Операции присваивания имеют вид:

```
identifier = expression
```

Здесь *identifier* является переменной, элементом массива или свойством объекта, куда требуется поместить значение. *expression* представляет значение (то есть данные, которые надо сохранить). Например:

```
x = 4;                  // Присваивает значение 4 переменной x
x = y;                  // Присваивает значение переменной y переменной x
name = "dj duong";      // Присваивает строку переменной name
products[3] = "Flash"   // Присваивает строку 4-му элементу products
                        // (индексы массивов отсчитываются с нуля)
// Присваивает число свойству area объекта square
square.area = square.width * 2;
```

Можно одновременно осуществить несколько присваиваний, например:

```
x = y = 4;  // Присвоить значение 4 переменным x и y
```

Помните, что операции присваивания ассоциативны справа налево, поэтому сначала 4 присваивается переменной y, а затем значение переменной y (которая теперь равна 4) присваивается переменной x.

В левой части операции присваивания не должно быть других операций. Например, в программировании недопустимо выражение, столь часто встречающееся в математике:

```
x + y = 10;  // ОШИБКА!
```

Как мы увидим позднее, проверка равенства двух выражений выполняется с помощью операторов == и !=. Например, следующий код проверяет равенство x + y и 10:

```
var x = 1;
var y = 8;
trace(x + y == 10);           // Выводит: false
```

И это не так, как в математике, где выражение $x + y = 10$ утверждает, что $x + y$ равно 10, а не выполняет проверку на равенство, как в нашем случае.

Сочетание операций с присваиванием

Операции присваивания часто применяются для назначения переменной нового значения, частично основанного на ее прежнем значении. Например:

```
counter = counter + 10;              // Прибавить 10 к текущему значению counter
xPosition = xPosition + xVelocity;   // Прибавить xVelocity к xPosition
score = score / 2;                   // Разделить score на два
```

Не путайте знак равенства, используемый для присваивания значения переменной, с алгебраическим знаком равенства. Отличие разъясняется в разделе «Присваивание значений переменным» главы 2.

ActionScript поддерживает сокращенную запись присваивания, называемую *составным присваиванием* (*compound assignment*), которая объединяет такие операторы, как +, - и /, с оператором присваивания, образуя единую

операцию «вычислить во время присваивания». Чтобы объединить сложение с присваиванием, используется +=. Деление объединяется с присваиванием при помощи /=. Предыдущие примеры, следовательно, можно более кратко записать с помощью операторов составного присваивания:

```
counter += 10;
xPosition += xVelocity;
score /= 2;
```

Список составных операторов присваивания имеется в табл. 5.1.

Арифметические операторы

Арифметические операторы выполняют математические действия над числовыми операндами. При использовании с арифметическими операторами нечисловых операндов ActionScript пытается преобразовать чуждые данные в числа. Например, `false - true` получает значение –1, поскольку `false` преобразуется в числовое значение 0, а `true` преобразуется в 1. Аналогично выражение `"3" * "5"` дает в результате число 15, потому что строки «3» и «5» перед выполнением умножения преобразуются в числа 3 и 5. Оператор + представляет собой особый случай: если в нем есть хотя бы один строковый операнд, то выполняется конкатенация строк, а не математическое сложение.

Если попытка преобразовать нечисловой операнд в число оказывается неудачной, то операнду присваивается особое числовое значение `NaN`. В результате операция в целом дает значение `NaN`. Подробную информацию о преобразованиях в число можно найти в табл. 3.1.

Сложение

Оператор сложения возвращает сумму двух своих операндов:

```
operand1 + operand2
```

Для того чтобы возвращаемый результат имел математический смысл, операндами оператора + должны быть выражения, возвращающие числовые значения, например:

```
234 + 5          // возвращает 239
(2 * 3 * 4) + 5  // возвращает 29
```

Оператор сложения отличается от прочих арифметических операторов тем, что если один или оба его операнда представляют собой строки, то он выполняет конкатенацию строк. См. раздел «Конкатенация строк» в главе 4.

Инкрементирование

Удобной разновидностью сложения является оператор *инкрементирования*, который принимает один операнд и добавляет к его текущему значению единицу. У инкрементирования есть два общих формата, называемые *префиксным инкрементированием* и *постфиксным инкрементированием*:

```
++operand    // Префиксное инкрементирование
operand++    // Постфиксное инкрементирование
```

В обоих форматах инкрементирование добавляет 1 к переменной, элементу массива или свойству объекта:

```
var x = 1;
x = x + 1;  // x теперь 2 (подробный синтаксис)
++x;        // Прибавить 1 с помощью префиксного оператора: x теперь 3
x++;        // Прибавить 1 с помощью постфиксного оператора: x теперь 4
```

При использовании отдельно от других операций разницы между префиксным и постфиксным инкрементированием нет, хотя чаще применяется постфиксное инкрементирование.

Однако в больших выражениях префиксное и постфиксное инкрементирование ведут себя по-разному: префиксное инкрементирование добавляет 1 к *operand* и возвращает *operand* + 1; постфиксное инкрементирование добавляет 1 к *operand*, но возвращает значение самого *operand*, а не *operand* + 1:

```
var x = 1;
// Префиксное инкрементирование: сначала увеличивается x, поэтому y
// устанавливается в 2
var y = ++x;
var x = 1;
// Постфиксное инкрементирование: y устанавливается в 1, затем x увеличивается до 2
var y = x++;
```

Мы снова обратимся к операторам инкрементирования в главе 8.

Вычитание

Оператор *вычитания* вычитает второй операнд из первого. Его общий вид таков:

```
operand1 - operand2
```

Операнды могут быть любыми допустимыми выражениями. Если какой-нибудь из операндов имеет нечисловой тип и преобразование в действительное число оказывается невозможным, операция дает в результате NaN:

```
234 - 5  // Возвращает 229
5 - 234  // Возвращает -229
```

Определение абсолютной (то есть положительной) разности между двумя числами осуществляется с помощью метода *Math.abs( )* (см. *Math. abs( )* в «Справочнике по языку»).

Декрементирование

Оператор *декрементирования* аналогичен оператору инкрементирования, но не прибавляет единицу к текущему значению операнда, а вычитает из него единицу. Подобно инкрементированию, оператор декрементирования

имеет два общих формата, называемые префиксным *декрементированием* и *постфиксным декрементированием*:

```
--operand  // префиксное декрементирование
operand--  // постфиксное декрементирование
```

В обоих форматах он применяется для вычитания 1 из переменной, элемента массива или свойства объекта. Префиксное декрементирование вычитает 1 из *operand* и возвращает значение *operand* - 1; постфиксное декрементирование вычитает 1 из *operand*, но возвращает значение самого *operand*, а не *operand* - 1. Как и в операторах инкрементирования, форма декрементирования имеет значение только в составе более крупного выражения:

```
var x = 10;
var y;
x = x - 1;     // x = 9
--x;           // x = 8 (префиксное декрементирование)
x--;           // x = 7 (постфиксное декрементирование)
y = --x;       // y = 6, x = 6 (префиксное декрементирование)
y = x--;       // y = 6 по-прежнему, а x = 5 (постфиксное декрементирование)
```

Умножение

Оператор *умножения* перемножает два числовых операнда и возвращает результат (то есть *произведение*). Общий вид операции умножения:

```
operand1 * operand2
```

Операнды могут быть любыми допустимыми выражениями. Символ * употребляется вместо символа × (умножить) из обычной математики. Если какой-либо из операндов имеет нечисловой тип и преобразование в число оказывается невозможным, результатом операции оказывается NaN:

```
6 * 5  // Возвращает 30
```

Деление

Оператор *деления* делит первый операнд (*делимое*) на второй операнд (*делитель*) и возвращает результат (*частное*). Общий вид операции деления:

```
operand1 / operand2
```

Операнды должны быть допустимыми числовыми выражениями. Символ / употребляется вместо символа ÷ из обычной математики. Если какой-либо из операндов имеет нечисловой тип и преобразование в число оказывается невозможным, результатом операции становится NaN. Если получается дробный результат, то частное представлено числом с плавающей точкой, даже если оба операнда целые:

```
20 / 5  // Возвращает 4
5 / 4   // Возвращает 1.25; в других языках можно получить 1, а не 1.25
```

Заметьте, что в других языках, например в Lingo для Macromedia Director, операция умножения возвращает целое число, если хотя бы один из операндов не является числом с плавающей точкой.

Если делитель (знаменатель) равен нулю, то результат деления будет таким:

- `NaN`, если числитель (делимое) равен нулю или не является числом
- `Infinity` (бесконечность), если числитель – положительное число
- `-Infinity`, если числитель – отрицательное число

Вот ряд примеров:

```
trace (0/0);   // NaN
trace ("a"/0); // NaN
trace (1/0);   // Infinity
trace (-1/0);  // (Infinity
```

Если делитель может оказаться нулем, проверьте его значение, прежде чем выполнять деление:

```
if (numItems != 0) {
  trace ("Среднее значение равно" + total / numItems);
} else {
  trace ("Нет чисел для вычисления среднего");
}
```

Заметьте, что в некоторых языках попытка деления на ноль вызывает ошибку.

Деление по модулю

Оператор деления *по модулю* возвращает остаток, получаемый при делении первого операнда на второй. Общий вид операции деления по модулю:

```
operand1 % operand2
```

Например, `14 % 4` возвращает 2, потому что 4 делит 14 нацело три раза, а в остатке получается 2.

Операнды деления по модулю могут быть любыми допустимыми числовыми выражениями, в том числе целыми числами и (в отличие от C и C++) числами с плавающей точкой. Например, `5 % 4` равно 1, а `5 % 4.5` равно 0,5. Если какой-либо из операндов имеет *нечисловой* тип и преобразование в число оказывается невозможным, операция дает результат `NaN`.

Если число четное, то его деление по модулю на 2 дает ноль. Чтобы проверить четность числа, можно использовать прием, показанный в примере 5.1.

Пример 5.1. Проверка четности чисел с помощью деления по модулю

```
var x = 3;
if (x%2 == 0) {
  trace("x - четное ");
} else {
  trace("x - нечетное");
}
```

Унарное отрицание

Оператор *унарного отрицания* принимает только один операнд. Он меняет знак операнда на противоположный (то есть положительное число становит-

ся отрицательным, а отрицательное – положительным). Унарное отрицание имеет общий вид:

```
-operand
```

Операнд может быть любым допустимым выражением. Вот как проверяется, что положение по горизонтали некоторого объекта не больше положительной границы и не меньше отрицательной границы:

```
if (xPos > xBoundary || xPos < -xBoundary){
  // Мы зашли слишком далеко
}
```

Унарный оператор отрицания – это просто знак «минус» (-). Он эквивалентен оператору вычитания, в котором роль первого операнда играет 0, например:

```
0 -operand
```

Операторы равенства и неравенства

Оператор *равенства* (==) позволяет проверить, не одинаковые ли значения имеют два выражения. Проверка равенства имеет общий вид:

```
operand1 == operand2
```

где *operand1* и *operand2* могут быть любыми допустимыми выражениями. Оператор равенства может сравнивать операнды любого типа. Если *operand1* и *operand2* равны, то выражение возвращает булево значение `true`; если они различны, оно возвращает `false`. Например:

```
// Записать значение 2 в x.
var x = 2;
// x равно 1?
if (x == 1) {
  // false; ничего не выводится
  trace("x is equal to 1);
}
// x равно 2?
if (x == 2) {
  // true; выводится сообщение
  trace("x is equal to 2);
}
```

Оператор равенства записывается с помощью двух знаков «равно» (==) подряд. Он определяет, равны ли два выражения. Не следует путать его с оператором присваивания (=), состоящим из одного знака равенства и используемым для присваивания переменной нового значения. Flash MX поддерживает также оператор строгого равенства, записываемый тремя знаками «равно» (===). См. ниже раздел «Операторы строгого равенства и неравенства».

Рассмотрим такой пример:

```
if (x = 5) {
```

```
  trace ("x равен 5")
}
```

В этом примере *не* проверяется, равна ли переменная x пяти. Вместо этого переменной x *присваивается* значение 5. Затем переменная x обрабатывается как булева, что дает результат true (положительные числа преобразуются в булево значение true). Вот как интерпретатор воспринимает наш пример:

```
x = 5;
if (x) {
  trace("x is equal to 5")
}
```

Правильная конструкция должна быть следующей:

```
// Использовать == вместо =
if (x == 5) {
  trace ("x равен 5")
}
```

Равенство для элементарных типов данных

Для элементарных типов данных результат проверки равенства обычно достаточно интуитивен. В табл. 5.2 перечислены правила, которым подчиняется равенство для каждого элементарного типа данных.

Таблица 5.2. Равенство элементарных типов данных

Тип	Условия равенства (оба операнда должны иметь указанный тип)

Числовой	Если *operand1* такое же число, как *operand2*, то результат true. Если оба операнда +Infinity или оба -Infinity, то результат true. Если оба операнда -0 или +0, то результат true. Для всех остальных комбинаций, в том числе если *один* или *оба* операнда NaN, результат false:
	`1 == 4 // false`
	`4 == 4 // true`
	`NaN == NaN // false`
	`+Infinity = -Infinity // false`

Строковый*	Выполняет сравнение строк с учетом регистра. Если *operand1* и *operand2* содержат строки одинаковой длины, состоящие из одинаковой последовательности символов, результат true; в противном случае результат false:
	`"Flash" == "Flash" // true`
	`"O'Reilly" == "O Reilly" // false`
	`"Moock" == "moock" // false ("m" и "M" - разные символы)`

Булев	Если оба операнда true или оба операнда false, результат true; в противном случае результат false:
	`true == true // true`
	`false == false // true`
	`true == false // false`

Тип	Условия равенства (оба операнда должны иметь указанный тип)
undefined	Если оба операнда undefined или один операнд undefined, а другой null, результат true; в противном случае результат false
null	Если оба операнда null или один операнд undefined, а другой null, результат true; в противном случае результат false
Сложные типы данных	См. следующий раздел «Равенство сложных типов данных»

* Оператор равенства Flash 4 eq считается устаревшим. Во Flash 5 и Flash MX следует применять оператор ==.

Равенство сложных типов данных

Переменные, содержащие составные данные (объекты, массивы, функции или клипы), хранят *ссылки* на данные, а не сами данные (см. раздел «Копирование, сравнение и передача данных» в главе 3), поэтому две переменные могут ссылаться на один и тот же объект. Такие два операнда считаются равными, только когда они ссылаются на одни и те же составные данные, но не тогда, когда два операнда ссылаются на два разных объекта с одинаковым содержимым. Даже если два операнда могут быть преобразованы в одно и то же элементарное значение, они не обязательно считаются равными.

Последующие примеры показывают, как ActionScript сравнивает ссылки, указывающие на составные данные, а не сами данные. В первом примере операнды (nameList1 и nameList2) ссылаются на массивы, которые содержат одинаковые элементы, но фактически это два разных массива. Поэтому ссылки разные, и сравнение дает результат false:

```
nameList1 = ["Linkovich", "Harris", "Sadler"];
nameList2 = ["Linkovich", "Harris", "Sadler"];
// Эти два массива не считаются равными, поэтому
// инструкция trace() не будет выполнена.
if (nameList1 == nameList2) {
  trace("Они равны");
}
```

В следующем примере cities и canadianCities ссылаются на один и тот же массив, поэтому они равны:

```
canadianCities = ["Toronto","Montreal","Vancouver"];
cities = canadianCities;
// Эти два массива считаются равными, поэтому инструкция trace() будет выполнена.
if (cities == canadianCities) {
  trace("Они равны");
}
```

В этом примере myFirstBall и mySecondBall имеют один и тот же конструктор (то есть оба объекта порождены одним и тем же классом), но они существуют как отдельные (не равные) экземпляры:

```
function Ball () {
  // ...здесь находится код инициализации...
```

```
}
myFirstBall = new Ball();
mySecondBall = new Ball();
myFirstBall == mySecondBall  // false
```

Как показано в главе 3, проверка равенства для сложных типов производится по ссылке, а не по значению.

Чтобы создать копию содержимого массива, не копируя ссылку на него, можно воспользоваться методом *Array.slice( )*. В следующем примере элементы массива `dishes` копируются в `kitchenItems`:

```
dishes = [ "cup", "plate", "spoon" ];
kitchenItems = dishes.slice(0, dishes.length);
trace(kitchenItems == dishes);  // Выводит: false
```

Теперь каждый из массивов `kitchenItems` и `dishes` содержит собственный набор элементов, которые могут быть изменены независимо друг от друга.

Равенство и преобразование типов данных

Мы видели результаты проверки равенства, когда оба операнда имеют одинаковый тип данных. Теперь посмотрим, что происходит, когда сравниваются операнды разных типов, например строка и число:

```
"asdf" == 13;
```

Если типы операндов разные, интерпретатор осуществляет перед сравнением преобразование типов. Вот правила, которым следует интерпретатор:

1. Если оба операнда имеют одинаковый тип, сравнить их и возвратить результат. (Если `null` сравнивается с `undefined`, возвращается `true`.)
2. Если один операнд представляет собой число, а другой – строку, преобразовать строку в число и вернуться к шагу 1.
3. Если один операнд булев, преобразовать булево значение в число (`true` = 1, `false` = 0) и вернуться к шагу 1.
4. Если один операнд представляет собой объект, вызвать метод *valueOf( )* объекта, чтобы преобразовать его в элементарный тип. Возвратить `false`, если это невозможно. В остальных случаях вернуться к шагу 1.
5. Возвратить `false`, если на предыдущих шагах не получен допустимый результат.

Обратите внимание, что если один операнд является объектом, а другой – булевым значением, то булево значение преобразуется в число и сравнивается со значением объекта элементарного типа. Это означает, что `someObject == true` обычно имеет значение `false`, даже если существует `someObject`, потому что `someObject` для сравнения преобразуется в число или строку, а `true` преобразуется в число 1. Чтобы заставить рассматривать `someObject` при сравнении как булево значение, применяется функция *Boolean( )*, например:

```
Boolean(someObject) == true    // Возвращает true, если someObject
                               // существует, иначе false
```

Преобразования, вызываемые операциями равенства, отдают предпочтение числовому типу. Если вам не ясны результаты только что описанных различных типов преобразований, обратитесь к разделу «Преобразование типа данных» главы 3. Обратите внимание, что преобразования типа, осуществляемые во время сравнения, не изменяют значения или тип данных исходного объекта. Результаты временного преобразования отбрасываются, как только завершено вычисление выражения.

Оператор неравенства

Оператор «*не равно*» (или оператор *неравенства*) возвращает значение, противоположное булеву результату оператора равенства. Часто «если `x` *не* равен `y`, сделать это» читается легче, чем «если `x` равен `y`, не делать ничего, в противном случае сделать это», как показано далее в примере 5.2. Проверка неравенства имеет общий вид:

```
operand1 != operand2
```

Например:

```
var a = 5;
var b = 6;
var c = 6;
a != b  // true
b != c  // false
```

Оператор неравенства следует тем же самым правилам преобразования типа, что и оператор равенства, и всегда возвращает противоположный результат (и если в качестве операнда выступает `NaN`):

```
NaN != 7    // true
NaN != NaN  // true!
```

В некоторых языках, в том числе ActionScript для Flash 4, в качестве оператора неравенства используется оператор `<>`. Во Flash 4 сравнение строк выполнялось посредством устаревшего оператора `ne`. Во Flash 5 и Flash MX вместо `<>` и `ne` для сравнений следует применять `!=`, независимо от типов данных операндов. См. также оператор *логического отрицания* (`!`), рассматриваемый далее в этой главе.

Частые случаи применения операций равенства

Мы будем часто использовать операции равенства для создания булевых выражений в условных конструкциях или для присваивания булева значения переменной. В примере 5.2 показаны обе ситуации и продемонстрированы операторы `!=` и `==` в действии.

Пример 5.2. Применение операторов равенства и неравенства

```
version = getVersion();      // Получить версию плеера
// Проверить, есть ли в version строка "WIN". Если да, установить
// isWin равной true, иначе установить ее равной false.
```

```
isWin = (version.indexOf("WIN") != -1);
// Если isWin - истина...
if (isWin == true) {
  // ...выполнить специфичные для Windows действия.
  trace("В Windows используйте IE5 или более поздний.");
}
```

Опытные программисты быстро укажут, что можно сократить *if(isWin== true)* до *if(isWin)*, потому что isWin содержит булево значение, которое и требуется инструкции *if*. Это совершенно верно, но мы лишимся хорошего примера использования оператора равенства. Кроме того, новичкам в программировании обычно понятнее более пространная форма. Оба подхода совершенно приемлемы. Более подробно эту тему мы рассмотрим в главе 7.

Операторы строгого равенства и неравенства

Как мы только что увидели, стандартный оператор равенства (==) может проверить, одинаковы ли значения двух выражений. Оператор *строгого неравенства*, записываемый как три последовательных знака равенства (===), сначала проверяет совпадение типов данных двух выражений и только потом – равенство их значений.

Например, следующая стандартная проверка равенства дает результат true, поскольку интерпретатор преобразует строку «13» в число 13 перед тем, как сравнивать значения:

```
trace("13" == 13);  // Выводит: true
```

Напротив, следующая проверка строгого равенства дает false, поскольку операнды имеют разные типы данных:

```
trace("13" === 13);  // Выводит: false
```

Общий вид строгой проверки равенства:

```
operand1 === operand2
```

где *operand1* и *operand2* могут быть любыми допустимыми выражениями. Оператор строгого равенства определяет, равны ли его операнды, по следующим правилам:

- Если *operand1* и *operand2* имеют разные типы данных, они не равны.
- Если *operand1* и *operand2* имеют значения undefined, они равны.
- Если *operand1* и *operand2* имеют значения null, они равны.
- Если *operand1* и/или *operand2* содержат особое числовое значение NaN, они не равны.
- Если *operand1* и *operand2* представляют собой числа и имеют одинаковые числовые значения, они равны.
- Если *operand1* и *operand2* представляют собой строки, состоящие из одних и тех же символов в одинаковом порядке, они равны.

- Если *operand1* и *operand2* оба имеют значение `true` или оба `false`, они равны.
- Если *operand1* и *operand2* хранят ссылку на один и тот же объект, они равны.
- Во всех остальных случаях *operand1* и *operand2* не равны.

Главная цель строгого равенства состоит в том, чтобы помешать интерпретатору выполнять преобразование типов перед сравнением двух операндов. Например, следующая условная инструкция проверяет, является ли переменная `numLives` числом и равно ли нулю ее значение:

```
if (numLives === 0) {
  trace("Game over");
}
```

Это более краткая запись следующего кода:

```
if ((typeof numLives == "number") && (numLives == 0)) {
  trace("Game over");
}
```

Строгая проверка неравенства имеет вид:

```
operand1 !== operand2
```

Оператор строгого неравенства (`!==`) возвращает булево значение, противоположное значению оператора строгого равенства.

Появившийся во Flash MX оператор строгого равенства записывается с помощью трех знаков равенства (`===`). Не следует путать его ни с обычным оператором равенства (`==`), ни с оператором присваивания (`=`). Аналогично строгое неравенство, введенное во Flash MX, записывается в виде восклицательного знака и двух знаков равенства (`!==`), тогда как обычный оператор неравенства записывается в виде восклицательного знака и одного знака равенства (`!=`).

Операторы сравнения

Операторы *сравнения* (называемые также операторами *отношения*) применяются для того, чтобы определять, которое из двух значений больше или меньше другого. Как и операторы равенства и неравенства, операторы сравнения возвращают булевы значения `true` или `false`, указывающие, является ли отношение, описываемое в сравнении, верным (`true`) или неверным (`false`).

Операторы сравнения предназначены для сравнения строк и чисел. При сравнении данных других типов происходит автоматическое преобразование типов, о котором говорится далее в этой главе. Если два операнда сравнения представляют собой числа, то сравнение производится математически: $5 < 10$ дает `true`, $-3 < -6$ дает `false` и т. д. Если операнды оператора сравнения представляют собой строки, то сравнение производится в соответствии с кодовыми позициями символов, как описывается в разделе «Порядок символов и сравнение по алфавиту» главы 4.

Интерпретатор пытается преобразовать указанные в операции сравнения данные, не являющиеся строками или числами, в данные *строкового* или *числового* типа. Действие преобразования типа в операциях сравнения мы рассмотрим после обсуждения самих операторов сравнения.

Оператор «меньше»

Оператор *меньше* имеет следующий общий вид:

```
operand1 < operand2
```

Когда операнды представляют собой числа, оператор «меньше» возвращает булево значение `true`, если *operand1* математически меньше, чем *operand2*:

```
5 < 6        // true
5 < 5        // false; числа равны, но 5 не меньше, чем 5
-3 < -6      // false; -3 больше -6
-6 < -3      // true;  -6 меньше -3
```

Когда операнды представляют собой строки, оператор «меньше» возвращает булево значение `true`, если *operand1* предшествует *operand2* в наборе символов Unicode; иначе он возвращает `false`:

```
"a" < "z"         // true; "a" на нижнем регистре предшествует "z" на нижнем регистре
"A" < "a"         // true; буквы верхнего регистра предшествуют буквам
                  // нижнего регистра
"Z" < "a"         // true; буквы верхнего регистра предшествуют буквам
                  // нижнего регистра
"hello" < "hi"    // true; "e" предшествует "i"
```

Оператор «больше»

Оператор *больше* имеет следующий общий вид:

```
operand1 > operand2
```

Когда операнды представляют собой числа, оператор «больше» возвращает булево значение `true`, если *operand1* математически больше, чем *operand2*:

```
5 > 6            // false
5 > 5            // false; равны, но 5 не больше, чем 5
-3 > -6          // true; -3 больше, чем -6.
-6 > -3          // false; -6 не больше, чем -3.
```

Когда операнды являются строками, оператор «больше» возвращает булево значение `true`, если *operand1* следует за *operand2* в наборе символов Unicode; иначе он возвращает `false`:

```
"a" > "z"        // false; "a" на нижнем регистре предшествует "z" на
                 // нижнем регистре
"A" > "a"        // false; буквы верхнего регистра не следуют после
                 // букв нижнего регистра
"Z" > "a"        // false; буквы верхнего регистра не следуют после
                 // букв нижнего регистра
"hello" > "hi"   // false; "e" меньше "i"
```

Оператор «меньше или равно»

Оператор *меньше или равно* имеет следующий общий вид:

```
operand1 <= operand2
```

Когда операнды являются числами, оператор «меньше или равно» возвращает булево значение `true`, если *operand1* математически меньше или равен *operand2*:

```
5 <= 6          // true
5 <= 5          // true; заметьте отличие от 5 < 5
-3 <= -6        // false
-6 <= -3        // true
```

Когда операнды являются строками, оператор «меньше» возвращает булево значение `true`, если *operand1* предшествует *operand2* в наборе символов Unicode или операнды идентичны (согласно правилам, описанным в разделе «Сравнение строк» главы 4); иначе он возвращает `false`:

```
"a" <= "z"       // true; "a" на нижнем регистре предшествует "z" на нижнем регистре
"A" <= "a"       // true; не равны, но "A" предшествует "a"
"Z" <= "a"       // true; буквы верхнего регистра предшествуют буквам
                 // нижнего регистра
"hello" <= "hi"  // true; "e" предшествует "i"
```

Обратите внимание, что в операторе <= знак равенства должен следовать за знаком «меньше». Запись оператора =< неправильна.

Оператор «больше или равно»

Оператор *больше или равно* имеет следующий общий вид:

```
operand1 >= operand2
```

Когда операнды являются числами, оператор «больше или равно» возвращает булево значение `true`, если *operand1* математически больше или равен *operand2*:

```
5 >= 6           // false
5 >= 5           // true; заметьте различие с 5 > 5
-3 >= -6         // true
-6 >= -3         // false
```

Когда операнды являются строками, этот оператор возвращает булево значение `true`, если *operand1* следует за *operand2* в наборе символов Unicode или операнды идентичны (согласно правилам, описанным в разделе «Сравнение строк» главы 4); иначе он возвращает `false`:

```
"a" >= "z"       // false; "a" на нижнем регистре предшествует "z"
                 // на нижнем регистре
"A" >= "a"       // false; "A" предшествует "a", и они не равны
"Z" >= "a"       // false; буквы верхнего регистра предшествуют буквам
                 // нижнего регистра
"hello" >= "hi"  // false: "e" предшествует "i"
```

Обратите внимание, что в операторе >= знак «больше» предшествует знаку равенства. Запись => неправильна.

Операции сравнения и преобразование типов данных

Чаще всего операторы сравнения применяются для сравнения чисел. Поэтому преобразования типов, вызываемые операторами сравнения, отдают предпочтение числам (так же, как операторы равенства и неравенства, описанные выше). Обратите внимание на отличие от оператора +, отдающего при преобразовании предпочтение строковому типу. Если два операнда в каком-либо операторе сравнения принадлежат разным типам или ни один из операндов не является строкой или числом, интерпретатор пытается выполнить преобразование типа согласно следующей процедуре:

1. Если оба операнда – числа, то операнды сравниваются математически и возвращается результат. Если один или оба операнда принимают значение NaN, то результат сравнения равен false, кроме случая оператора !=.
2. Если оба операнда – строки, то операнды сравниваются в лексикографическом порядке в соответствии с кодовыми позициями символов в Unicode и возвращается результат. Для символов латинского алфавита сравнение выполняется с учетом регистра.
3. Если один операнд представляет собой число, а другой – строку, то строка преобразуется в число, после чего выполняется возврат на шаг 1.
4. Если один из операндов – булево значение, null или undefined, то выполняется преобразование операнда в число и возврат на шаг 1.
5. Если один из операндов является объектом, то вызывается его метод *valueOf( )* для преобразования объекта в значение элементарного типа, после чего выполняется возврат на шаг 1. Если вызов метода *valueOf( )* оказывается неудачным или не возвращает значение элементарного типа, то результат сравнения принимает значение false.
6. Во всех других случаях возвращается false.

Обратите внимание, что преобразования типа, выполняемые во время сравнения, не изменяют значение или тип данных исходного объекта. Результаты временного преобразования отбрасываются, как только завершено вычисление выражения.

Вот простой пример сравнения двух булевых значений:

```
false < true      // true: 0 меньше 1
```

Операторы сравнения всегда преобразуют составные типы данных в строки или в числа для проведения сравнения. В следующем примере, поскольку someObj и someOtherObj являются членами класса *Object*, их строковые значения, «[object Object]», одинаковы:

```
someObj = new Object();
someOtherObj = new Object();
someObj <= someOtherObj;    // true!
```

Напротив, выражение `someObj == someOtherObj` дает значение `false`, поскольку равенство объектов проверяется по ссылкам, и они не преобразуются к элементарным типам во время проверки.

В следующем примере, несмотря на то что кодовая позиция символа «A» равна 65, его преобразование в число дает `NaN`, а это значит, что выражение в целом имеет значение `false`. Для получения кодовой позиции строки следует применять функцию *charCodeAt( )*:

```
"A" <= 9999                // false
"A".charCodeAt(0) < 9999   // true
```

Строковые операторы Flash 4

При изучении кода Flash 4 вам могут встретиться устаревшие операторы для работы со строками – конкатенации (`&`), равенства (`eq`), неравенства (`ne`) и сравнения (`lt`, `gt`, `le`, `ge`). Во Flash 5 и Flash MX для всех сравнений, включая сравнения строк, следует применять операторы `==`, `!=`, `<`, `>`, `<=` и `>=`, если только вы не экспортируете файл в формат Flash 4 *.swf*; в этом случае необходимо использовать старые операторы (`eq`, `ne`, `lt`, `gt`, `le`, `ge`). Точно так же конкатенировать строки следует при помощи оператора `+`, а не устаревшего `add` или `&`, который во Flash 5 и последующих версиях является поразрядным оператором *AND*.

Дополнительные сведения и список эквивалентных строковых операций Flash 4 и Flash 5 можно найти в разделе «Строковые операторы Flash 4 и Flash 5» приложения C.

Логические операторы

В главе 4 было показано, как принимать логические решения с использованием условных инструкций и булевых значений. Решения основывались на одиночных факторах: если фильм не загружен, не начинать его воспроизведение; если у космического корабля оружие удвоенной мощности, увеличить разрушения, причиняемые его выстрелами, и т. д. Не всегда логика программ так проста. Часто требуется рассматривать множественные факторы в логике *ветвления* (то есть принятия решений).

Предположим, что мы создаем персональный Flash-сайт, требующий регистрации пользователей. Если пользователь регистрируется как гость, предлагаемое ему содержимое может быть ограничено. При авторизации на сайте зарегистрированного пользователя устанавливается переменная, которая на протяжении всего фильма указывает, что для данного пользователя открыт неограниченный доступ к просмотру содержимого:

```
var userStatus = "registered";
```

Для того чтобы определить, разрешен ли доступ в привилегированную область, используем такую булеву проверку:

```
if (userStatus == "registered") {
```

```
  // okay, впустить...
}
```

Допустим, что требуется показать сайт предполагаемым инвесторам, не заставляя их регистрироваться и не показывая материал, предназначенный для зарегистрированных пользователей. Можно создать новую категорию пользователей, «specialGuest», которая позволит инвесторам увидеть больше, чем гостям, но меньше, чем разрешено зарегистрированным пользователям. Когда мы хотим указать, что некто является особым гостем, мы присваиваем переменной `userStatus` значение «specialGuest».

Как проводить булеву проверку теперь, когда есть две категории пользователей? Можно сделать это грубым способом, дублируя участки кода:

```
if (userStatus == "registered") {
  // Выполнить код для легальных пользователей...
}
if (userStatus == "specialGuest") {
  // Выполнить тот же самый код для легальных пользователей...
}
```

Очевидно, что сопровождать такой код будет весьма затруднительно, поскольку модифицировать одно и то же придется дважды.

Лучше было бы обойтись одной булевой проверкой, сформулированной так: «Если пользователь зарегистрирован *или* является особым гостем, то предоставить доступ». Строить такие составные логические конструкции позволяют булевы *логические операторы*. В данном случае оператор логического *ИЛИ* (||) позволяет в одном выражении проверить состояние нескольких булевых операций. Чтобы сделать порядок действий более понятным, я включил дополнительные скобки:

```
if (userStatus == "registered") || (userStatus == "specialGuest") {
  // Выполнить код для легальных пользователей...
}
```

Красиво? Иногда программирование очень элегантно. Такие четкие вертикальные линии... такие гибкие скобки...

Логические выражения и значения данных

Применение логических операторов для вычисления булева результата аналогично применению арифметических операторов для получения числового результата. Так, следующие четыре операнда и три оператора используются для вычисления единственного результата:

```
userStatus == "registered" || userStatus == "specialGuest"
```

Если пользователь не зарегистрирован, а является особым гостем, то предыдущее выражение сводится к следующему:

```
false || true
```

что, в свою очередь, сводится к булеву результату `true`.

Вот аналогичная арифметическая операция, которая сводится к одному числу:

```
(1 + 2) * (3 + 4)  // Четыре значения (1, 2, 3, 4), становящиеся...
3 * 7              // двумя значениями (3, 7), становящимися...
21                 // одним значением (21)
```

Запомните, что сложное выражение, сводящееся к булеву значению, может находиться в любом месте, где требуется одно булево значение (например, в проверочном выражении условной инструкции).

Логическое ИЛИ

Оператор логическое *ИЛИ* чаще всего применяется для выполнения каких-либо действий, когда удовлетворяется хотя бы одно из условий. Например: «Если я голоден *или* мне хочется пить, я пойду на кухню». Значок логического *ИЛИ* составлен из двух символов вертикальной черты (||). Символ | обычно можно получить с помощью клавиш Shift и Backslash (\) с изображением прерывистой вертикальной черты, которая на большинстве западных клавиатур находится в правой верхней области. Логическое *ИЛИ* имеет следующий общий вид:

```
operand1 || operand2
```

Когда оба операнда представляют собой булевы выражения, логическое *ИЛИ* возвращает `true`, если один из операндов равен `true`, и возвращает `false`, только если *оба* операнда равны `false`. Подытожим:

```
true  || false   // true, потому что первый операнд true
false || true    // true, потому что второй операнд true
true  || true    // true (достаточно истинности любого из операндов)
false || false   // false, потому что оба операнда false
```

Это упрощенный случай, когда оба операнда булевы, но операнды могут быть любыми допустимыми выражениями, в том числе небулевыми. В последнем случае возвращаемое операцией логического *ИЛИ* значение не обязательно является булевым (`true` или `false`). Формально первый шаг в вычислении логического *ИЛИ* состоит в преобразовании *operand1* к булеву типу. Если результатом преобразования является `true`, логическое *ИЛИ* возвращает разрешенное значение *operand1*, в противном случае логическое *ИЛИ* возвращает разрешенное значение *operand2*. Проиллюстрируем это несколькими примерами:

```
null || "hi there!"  // Operand1 не преобразуется в true, поэтому
                     // возвращается значение operand2: "hi there!"
true || false        // Operand1 есть true, поэтому возвращается
                     // значение operand1: true
"hey" || "dude"      // Operand1 - непустая строка, которая
                     // преобразуется в false и операция возвращает
                     // значение operand2: "dude"
false || 5 + 5       // Operand1 не преобразуется в true, поэтому
                     // возвращается значение operand2 (а именно 10)
```

На практике небулевы значения, возвращаемые выражением с логическим *ИЛИ*, используются редко. Вместо этого результат обычно применяется в условной инструкции, где его используют для принятия булева решения. Рассмотрим пример 5.3.

Пример 5.3. Операция логического ИЛИ как проверочное выражение в условной инструкции

```
var x = 10;
var y = 15;
if (x || y) {
  trace("Один из x и y отличен от нуля");
}
```

В строке 3 мы видим операцию логического *ИЛИ* (*x* || *y*) там, где ожидается булева величина в качестве проверочного выражения инструкции *if*. Первый шаг в определении значения *x* || *y* заключается в преобразовании 10 (значения первого операнда, `x`) в булеву величину. Как показывает табл. 3.3, любое конечное число, отличное от нуля, преобразуется в булево значение `true`. Когда первый операнд преобразуется в `true`, логическое *ИЛИ* возвращает значение первого операнда, в данном случае 10. Поэтому для интерпретатора наша инструкция *if* выглядит так:

```
if (10) {
  trace("Один из x и y отличен от нуля");
}
```

Но 10 – это число, а не булева величина. Что же происходит? Инструкция *if преобразует* значение, возвращаемое операцией логического *ИЛИ*, в булево. В данном случае 10 преобразуется в булево значение `true` (опять-таки, согласно правилам табл. 3.3), и интерпретатор видит наш код так:

```
if (true) {
  trace("Один из x и y отличен от нуля");
}
```

Вот и все. Проверочное выражение равно `true`, поэтому выполняется инструкция *trace( )* в фигурных скобках.

Обратите внимание, что если первый операнд в операции логического *ИЛИ* разрешается (resolves) в `true`, то нет необходимости и потому неэффективно вычислять второй операнд. ActionScript вычисляет второй операнд, только если первый операнд разрешается в `false`. Это обстоятельство полезно в тех случаях, когда требуется, чтобы второй операнд разрешался только тогда, когда первый операнд разрешается в `false`. В следующем примере мы проверяем, не вышло ли число за допустимые границы. Если число слишком маленькое, то во второй проверке не надо выяснять, не слишком ли оно велико:

```
if (xPos < 0 || xPos > 100) {
  trace ("xPos не попадает в диапазон от 0 до 100 включительно.");
}
```

Обратите внимание, что переменная xPos должна участвовать в каждом сравнении. Следующий код демонстрирует распространенную ошибку с двумя проверками xPos:

```
// Пропущено xPos в сравнении с числом 100
if (xPos < 0 || > 100) {
  trace ("xPos не попадает в диапазон от 0 до 100 включительно.");
}
```

Логическое И

Как и оператор логического *ИЛИ*, оператор *логического И* задает условие выполнения какого-то блока кода, на этот раз только если удовлетворены оба условия. Оператор логического *И* имеет следующий общий вид:

```
operand1 && operand2
```

Оба операнда, *operand1* и *operand2*, могут быть любыми допустимыми выражениями. В простейшем случае, когда оба операнда являются булевыми выражениями, логическое *И* возвращает false, если хотя бы один из операндов false, и возвращает true, только если *оба* операнда true. Вот как это выглядит:

```
true  && false   // false, потому что второй операнд false
false && true    // false, потому что первый операнд false
true  && true    // true потому что оба операнда true
false && false   // false потому что оба операнда false (достаточно одного)
```

Рассмотрим применение оператора логического *И* на двух реальных примерах. В примере 5.4 функция *trace( )* вызывается только тогда, когда обе переменные больше 50.

Пример 5.4. Операция логического И как проверочное выражение в условной инструкции

```
x = 100;
y = 51;
if (x>50 && y>50) {
  trace("Каждый из x и y больше 50");
}
```

Поскольку оба выражения, *x>50* и *y>50*, имеют значение true, выполняется инструкция *trace( )*.

В примере 5.5 мы возвращаемся к нашему веб-сайту, на котором доступ предоставлялся зарегистрированным пользователям и особым гостям. Но на этот раз мы впустим пользователей, только если дата отлична от 1 января. Мы объединим выражения с помощью операторов логического *ИЛИ* (для проверки двух разрешенных значений userStatus) и логического *И* (для проверки данных). Обратите внимание на круглые скобки, гарантирующие, что предложение с логическим *ИЛИ* вычислит userStatus до выполнения логического *И* в проверке даты. Текущая дата определяется с помощью класса *Date*, в котором метод *getMonth( )* возвращает 0 для января.

Пример 5.5. Составные логические выражения

```
userStatus = "registered";
now = new Date();         // Создать новый объект Date
day = now.getDate();      // Возвращает целое от 1 до 31
month = now.getMonth();   // Возвращает целое от 0 до 11
// Обратите внимание на отступы и скобки... инструкции if могут
// для улучшения читаемости разбиваться на несколько строк
if ( ((userStatus == "registered") || (userStatus == "specialGuest"))
    && (month + day) > 1) {
  // Впустить пользователя...
}
```

Формальное поведение оператора *И* аналогично действию оператора *ИЛИ*. Сначала *operand1* преобразуется в булево значение. Если результатом преобразования является `false`, возвращается значение *operand1*. Если результат этого преобразования равен `true`, то возвращается значение *operand2*.

Если первый операнд в операции логического *ИЛИ* разрешается в `false`, то нет необходимости и потому неэффективно вычислять второй операнд. ActionScript вычисляет второй операнд, только если первый операнд разрешается в `true`. Это обстоятельство полезно в тех случаях, когда требуется, чтобы второй операнд разрешался только тогда, когда первый операнд разрешается в `true`. В следующем примере мы выполняем деление, только если делитель отличен от нуля:

```
if (numItems != 0 && total / numItems > 3) {
  trace ("Средняя сумма операций превысила 3 доллара");
}
```

Логическое отрицание

Оператор *логического отрицания* (`!`) возвращает булево значение, противоположное своему единственному операнду. Он имеет следующий общий вид:

```
!operand
```

где *operand* может быть любым допустимым выражением. Если *operand* равен `true`, логическое отрицание возвращает `false`. Если *operand* равен `false`, логическое отрицание возвращает `true`. Если *operand* не является булевой величиной, его значение преобразуется в булево и возвращается значение, противоположное ему.

Как и оператор «не равно» (`!=`), оператор логического отрицания удобен для проверки именно того, что операнд *не является чем-либо*. Вернемся к примеру 5.5, в котором мы хотели выполнить действие, только если дата отлична от 1 января. На первый взгляд следующая попытка может показаться корректной:

```
month != 0 && day != 1
```

Однако мы обнаруживаем, что *month != 0* имеет значение `true` всегда, когда месяц отличен от января (пока все хорошо), но *day != 1* имеет значение `false` первого числа каждого месяца. Поэтому выражение *month != 0 && day != 1*

отвергает пользователей первого числа любого месяца, а не только 1 января. Это не то, что нам требовалось.

Значительно легче проверить, что текущий день – *именно* 1 января:

```
month == 0 && day == 1
```

Это выражение дает `true` только в первый день января. Теперь нам требуется лишь применить оператор логического отрицания, чтобы определить, что сегодня *не* 1 января:

```
!(month==0 && day==1)
```

Очевидно, назначение этого кода значительно понятнее, чем в проверке $month+day > 1$, реализованной в примере 5.5.

Другое типичное применение оператора логического отрицания состоит в *переключении* переменной с `true` на `false` и наоборот. Допустим, что имеется одна кнопка для включения и выключения звука. Код может быть примерно таким:

```
soundState = !soundState  // Изменить текущее состояние звука
if (soundState) {
  // Если звук включен, обеспечить его слышимость
} else {
  // Если звук выключен, установить громкость в 0, чтобы заглушить его
}
```

Обратите внимание, что знак `!` есть также в операторе (`!=`). В качестве программной *лексемы* (то есть знака) символ `!` обычно означает *не*, или *противоположный*. Он не имеет отношения к символу (`!`), обозначающему факториал в обычных математических формулах.

Оператор группирования

Круглые скобки `()` находят применение не только в вызовах функций, но и служат для объединения конструкций кода в группы (что делается для преодоления приоритетов различных операторов). Они также улучшают читаемость кода, даже если они не нужны для изменения порядка выполнения операций по умолчанию. Круглые скобки требуются также в некоторых инструкциях, в частности в инструкции *if*. Круглые скобки имеют следующий общий вид:

```
(expression)
```

Заключенное в скобки выражение *expression* вычисляется, и возвращается его результат. Вот несколько примеров:

```
if (x == y) {                  // Необходимый синтаксис в инструкции if
  trace("x and y are equal");  // Требуются в вызове функции
}
(5 + 6) * 7                    // Навязывают нестандартный порядок операций
(x >= 100) || (y <= 50)        // Не требуются, но добавлены для читаемости
x >= 100 || y <= 50            // Без скобок... читать немного труднее
```

Оператор «запятая»

Редко применяемый оператор *запятая* (`,`) позволяет вычислять два выражения там, где ожидается одно выражение. Он имеет следующий общий вид:

```
operand1, operand2
```

Оба операнда должны быть допустимыми выражениями. При выполнении оператора вычисляется *operand1*, затем вычисляется *operand2* и возвращается полученное значение *operand2*. На практике значение, возвращаемое этим оператором, обычно игнорируется.

Оператор «запятая» применяется в основном в циклах for для инициализации нескольких переменных, как показано в примере 5.6. Обратите внимание, что в первой строке примера два выражения, `i=0` и `j=10`, представляют собой операнды оператора «запятая». (Эти выражения осуществляют инициализацию переменных `i` и `j` значениями 0 и 10.) Выражения `i++` и `j--` представляют собой операнды второго оператора запятой. (Эти выражения вызывают инкрементирование `i` и декрементирование `j` при каждом выполнении цикла.) Оператор «запятая» используется для того, чтобы втиснуть несколько выражений в рамки синтаксиса цикла *for*.

Пример 5.6. Применение оператора «запятая» в цикле for

```
for (var i=0, j=10; i!=j; i++, j--) {
  trace ("i: " + i + " j: " + j);
}
// получаем в окне Output:
 i: 0 j: 10
 i: 1 j: 9
 i: 2 j: 8
 i: 3 j: 7
 i: 4 j: 6
```

Оператор «запятая» не следует путать с другими применениями запятой, например для разделения аргументов, передаваемых функции, или элементов в массиве.

Оператор void

Оператор *void* предназначен для вычисления выражения без возврата результата. Его синтаксис:

```
void expression
```

где *expression* представляет собой выражение, подлежащее вычислению. В JavaScript *void* применяется для вычисления выражения JavaScript в гипертекстовой ссылке без вывода результата в адресную панель броузера. Такое применение *void* в ActionScript встречается редко, если вообще встречается, и этот оператор включен в состав языка для совместимости с ECMA-262.

Прочие операторы

Применение остальных операторов подробнее рассматривается в других главах, а здесь они представлены лишь для краткой справки.

Оператор typeof

Оператор *typeof* предназначен для определения типа данных выражения или идентификатора. Он принимает один операнд:

```
typeof operand;
```

где *operand* может быть любым допустимым выражением. Операция *typeof* возвращает строку, обозначающую тип данных вычисляемого *operand*. Дополнительные сведения представлены в главе 3.

Оператор new

Оператор *new* создает новый составной элемент данных – в виде массива или объекта. Объект может быть членом встроенного класса или класса, определенного пользователем. Синтаксис *new*:

```
new constructor
```

где *constructor* должен быть функцией, определяющей свойства вновь создаваемого объекта. См. главы 11, 12 и разделы «Конструктор» в «Справочнике по языку».

Оператор delete

Оператор *delete* применяется для удаления из сценария объекта, свойства объекта, элемента массива или переменных. Синтаксис *delete*:

```
delete identifier
```

Если *identifier* не является контейнером данных (переменной, свойством или элементом), операция *delete* оказывается неудачной и возвращает значение false; в противном случае она возвращает true, что указывает на успешное выполнение. Оператор *delete* часто используется для удаления обработчика события, например:

```
// Добавить обработчик события:
_root.onEnterFrame = function () {
  trace("A frame passed");
}

// Удалить обработчик события:
delete _root.onEnterFrame;
```

Подробнее об операторе *delete* рассказано в главах 10, 11 и 12.

Оператор элемента массива/свойства объекта

Как мы увидим в главах 11 и 12, оператор [ ] используется для извлечения и присваивания значения элемента массива или свойства объекта. При обращении к массиву он имеет следующий вид:

```
array[element]
```

где *array* – это имя массива или литерал массива, а *element* – это выражение, значением которого является неотрицательное целое число, представляющее собой индекс элемента массива, к которому происходит обращение (индексация начинается с нуля).

При обращении к объекту оператор [ ] имеет следующий вид:

```
object[property]
```

где *object* представляет собой имя объекта или литерал объекта, а *property* – это любое выражение, значением которого является строка, представляющая имя свойства объекта, к которому происходит обращение.

Элементу или свойству, расположенному в левой части оператора присваивания (=), присваивается новое значение, представленное в правой части:

```
var colors = new Array();  // Создать новый массив
colors[0] = "orange";      // Установить его первый элемент
colors[1] = "green";       // Установить его второй элемент
var ball = new Object();   // Создать новый объект
var myProp = "xVelocity";  // Сохранить строку в переменной
ball["radius"] = 150;      // Установить свойство radius
ball[myProp] = 10;         // Установить свойство xVelocity через myProp
```

В других случаях выражение возвращает значение указанного элемента или свойства:

```
diameter = ball["radius"] * 2;  // Присваивает diameter значение 300
trace(colors[0]);               // Выводит "orange"
```

Оператор «точка»

Оператор *точка* – это основное средство обращения к свойствам объектов и вложенным клипам. Функционально оператор «точка» имеет точно такое же назначение, как оператор [ ] – он позволяет задавать и извлекать значения свойств объектов. Но эти два оператора имеют синтаксические различия, делающие их уникальными. Общий синтаксис оператора «точка» таков:

```
object.property
```

где *object* должен быть именем объекта или литералом объекта, а *property* – это идентификатор, представляющий свойство *object*. Обратите внимание, что *property* ***не*** может быть произвольным выражением или строковым литералом, это должно быть имя свойства. Поскольку элементы массива нумеруются, а не именуются, с помощью оператора «точка» нельзя получить доступ к элементам массива.

В качестве левого операнда оператора присваивания оператор «точка» участвует в присваивании свойству нового значения:

```
var ball = new Object();
ball.radius = 150;
ball.xVelocity = 10;
```

В любом другом месте оператор «точка» возвращает значение указанного свойства объекта:

```
diameter = ball.radius;
newX = ball.xPosition + ball.xVelocity;
```

Оператор «точка» применяется также для вызова методов объектов (функций, хранимых в свойствах), например:

```
intro_mc.gotoAndPlay("beginMovie");
```

Подробнее об операторе «точка» рассказано в главах 12 и 13.

Условный оператор

Условный оператор является синтаксической конструкцией, позволяющей в сжатом виде представить простую условную инструкцию. Этот оператор принимает три операнда и имеет следующий формат:

```
condition ? result_if_true : result_if_false;
```

При выполнении условной операции вычисляется первый операнд (*condition*). Если *condition* имеет значение `true` или может быть преобразован в `true`, возвращается значение второго операнда (*result_if_true*). В противном случае возвращается значение третьего операнда (*result_if_false*). Подробнее об условном операторе рассказано в главе 7.

Оператор instanceof

Оператор *instanceof*, добавленный во Flash MX, проверяет, принадлежит ли объект заданному классу (или одному из его подклассов). Формат этого оператора следующий:

```
object instanceof classConstructor
```

Здесь *object* – это любой объект, а *classConstructor* представляет собой ссылку на какую-либо функцию конструктора класса. Операция возвращает `true`, если *object* является экземпляром *classConstructor* или одного из его подклассов; в противном случае возвращается `false`. Например, следующий код создает объект *Date*, а затем проверяет, является ли этот объект экземпляром класса *Date*:

```
// Создать экземпляр встроенного класса.
var now = new Date();
trace(now instanceof Date);  // Выводит: true
```

Следующий код создает класс *Shape* с подклассом *Circle*, а затем проверяет, является ли объект класса *Circle* экземпляром *Shape*:

```
// Создать классы Shape и Circle (код тела конструктора опущен)
function Shape () {
  // ...
}
function Circle () {
  // ...
}
// Установить наследование: Circle является подклассом Shape
Circle.prototype = new Shape();

// Создать объект Circle
var theCircle = new Circle();
// Проверить класс объекта Circle с помощью instanceof
trace(myCircle instanceof Circle);  // true
trace(myCircle instanceof Shape);   // true
```

Оператор *instanceof* можно применять для того, чтобы условно выполнять действия над объектом в зависимости от его класса. Например, в следующем коде с помощью *instanceof* удаляются все текстовые поля и клипы из текущего клипа (`this`):

```
for (var p in this) {
  if (this[p] instanceof MovieClip) {
    this[p].removeMovieClip();
  } else if (this[p] instanceof TextField) {
    this[p].removeTextField();
  }
}
```

Обратите внимание, что оператор *instanceof* просто проверяет, входит ли *classConstructor* в цепочку прототипов объекта *object*, как описано в разделе «Всемогущая цепочка прототипов» главы 12.

«Оператор» super

Во Flash MX введен инструмент *super*, который напоминает оператор и решает две задачи объектно-ориентированного программирования:

- Может вызывать конструктор надкласса объекта.
- Может вызывать реализованный в надклассе перегруженный метод.

В том и другом случае *super* можно использовать только внутри конструктора или метода класса.

Синтаксис вызова конструктора надкласса:

```
super(arg1, arg2,...argn);
```

Здесь *arg1,...argn* представляет собой список аргументов, передаваемых функции-конструктору надкласса. Внутри конструктора надкласса ключевое слово `this` указывает на объект, класс или метод которого вызвал конструктор. Такое применение оператора *super* функционально эквивалентно следующей инструкции:

```
this.constructor.prototype.constructor.apply(this, arguments);
```

Синтаксис вызова метода из надкласса таков:

```
super.methodName(arg1, arg2,...argn);
```

Здесь *methodName* – какой-либо метод надкласса, а *arg1,...argn* – список аргументов, передаваемых этому методу. Внутри метода надкласса ключевое слово `this` указывает на объект, класс или метод которого вызвал данный метод. Такое применение оператора *super* функционально эквивалентно следующей довольно громоздкой инструкции:

```
this.constructor.prototype.__proto__.constructor.apply(this, arguments);
```

Практические примеры применения *super* можно найти в разделах «Создание класса» и «Доступ к перегруженным методам» главы 12.

Обратите внимание, что формально *super* не является оператором. Скорее это идентификатор, доступный только во время выполнения функции, примерно как объект `arguments`. В ActionScript *super* не есть реализация оператора *super* из ECMAScript v4. Тем не менее для удобства изложения в ActionScript Dictionary Macromedia и данной книге *super* помещается в одну категорию с операторами.

Оператор вызова функции

Изучая функцию *trace( )* и функции обработки строк, мы видели, что для вызова функции применяется оператор вызова функции `()`. Вызов функции имеет следующий общий вид:

```
function_name(argument_list)
```

Первый операнд, *function_name*, – это имя некоторой функции, и он *должен* быть идентификатором, а не выражением. Если функция не существует, интерпретатор не сгенерирует ошибку, а просто не выполнит вызов без всяких сообщений. Параметр *argument_list* представляет собой последовательность из нуля или более аргументов, передаваемых функции, которые разделяются запятыми. Значение, возвращаемое операцией вызова функции, предоставляется самой функцией.

Оператор вызова функции может вызывать любую встроенную или определенную пользователем функцию:

```
trace("an internal function"); // Встроенная функция, один аргумент
theClip_mc.play();             // Метод клипа, без аргументов
myRectangle.area(6, 9);        // Определенный пользователем метод, два аргумента
init();                       // Определенная пользователем функция, без аргументов
```

Поразрядные операторы

ActionScript поддерживает группу так называемых *поразрядных* (*bitwise*, *побитовых*) операторов, которые действуют над двоичными цифрами (битами) целых чисел. Если вы планируете разработку крупных систем, в которых большое значение для производительности имеют каждый байт памя-

ти, эффективность расчетов и оптимизация скорости передачи данных, почитайте электронную документацию о поразрядных операторах по адресу:

http://www.moock.org/asdg/technotes/bitwise

В иных случаях пользуйтесь булевыми логическими операторами, о которых рассказывалось ранее в этой главе. Они выполняют те же задачи, что и поразрядные операторы, но менее оптимизированным способом. Краткая справка по поразрядным операторам (<<, >>, >>>, ~, &, ^ и |) есть в табл. 5.1.

Посмотрите в описании метода *Color.getRGB( )* в «Справочнике по языку» пример, в котором поразрядные операторы применяются для извлечения красной, зеленой и синей составляющих из кода цвета RGB.

Вперед!

Как и во всех языках, в ActionScript есть грамматика и разные части речи. В нескольких последних главах вы познакомились с эквивалентами структуры предложения, существительных, прилагательных и союзов. Теперь перейдем к командам, которые, подобно глаголам, могут выполнять действия. Команды дополняют наши возможности создавать похожие на предложения инструкции.

6

Инструкции

В предыдущих главах мы посмотрели, как ActionScript хранит и обрабатывает данные, а в этой главе увидим, как с этими данными можно что-то делать. Мы будем давать Flash указания в виде *инструкций* (*statements*), или кодовых конструкций, предписывающих интерпретатору выполнение некоторой задачи.

Для того чтобы вызвать во Flash некоторое событие – прекратить звуковоспроизведение, воспроизвести фильм, многократно выполнить некоторый код, – используются инструкции. Фактически программу ActionScript можно определить просто как перечень инструкций, сообщающих интерпретатору, выполнения каких действий мы хотим от Flash. Вот, например, целая программа ActionScript, которая состоит из четырех инструкций и предписывает интерпретатору загрузить в броузер веб-страницу:

```
var protocol = "http";                                  // Инструкция 1
var domain = "www.moock.org";                           // Инструкция 2
var path = "webdesign/flash/";                          // Инструкция 3
getURL(protocol + "://" + domain + "/" + path);         // Инструкция 4
```

Создание сценария фильма с помощью ActionScript состоит просто в прикреплении инструкций к кадрам, клипам и кнопкам. Если не указано иное, Flash выполняет все инструкции в порядке их появления в сценарии (позднее мы поговорим о том, как и когда Flash решает, какой сценарий выполнять). В этой главе рассмотрена синтаксическая структура инструкций и перечислены их общие категории. В ней мы коснемся всех инструкций ActionScript, но некоторые наиболее важные из них будут более подробно исследованы в следующих главах.

Типы инструкций

Концептуально инструкции можно разбить на пять типов:

Инструкции, управляющие логикой выполнения программы

- циклы
- условные инструкции
- ifFrameLoaded

Инструкции, объявляющие переменные

- var
- set

Инструкции, объявляющие функции, а также вызывающие их и возвращающие значения

- function
- function call
- return

Инструкции для работы с объектами

- with
- for ... in

Инструкции, представляющие значения данных

- любое выражение (особенно выражение с побочным эффектом)

Эти неформальные категории помогут понять, каких действий мы можем потребовать от интерпретатора с помощью инструкций. На первый взгляд список может показаться довольно ограниченным. Однако мы увидим, что есть много разновидностей условных инструкций и циклов, а также тысячи вещей, которые можно делать с помощью вызовов функций. Прежде всего рассмотрим формат инструкций.

Синтаксис инструкций

Инструкции обычно состоят из *ключевых слов* и *выражений*. Мы уже видели инструкцию *var*, которая объявляет и может дополнительно инициализировать переменную. Вот пример общего синтаксиса инструкции *var*:

```
var numFrames;
```

Ключевое слово (в данном случае *var*) начинает инструкцию. За каждым ключевым словом следует синтаксис (скобки, параметры, операнды и т. д.), необходимый для завершения инструкции. В нашем примере он состоит просто из имени переменной, `numFrames`. Наконец, точка с запятой отмечает конец инструкции.

Каждая инструкция должна заканчиваться точкой с запятой. Это не означает, что точка с запятой присутствует в каждой строке сценария. Иногда в ней размещается несколько инструкций, каждая из которых может оканчиваться своим символом точки с запятой.

Иногда инструкция занимает несколько строк, и точка с запятой появляется только в ее конце. Точка с запятой не требуется после фигурных скобок, в которые иногда заключаются инструкции, за исключением особого случая литерала функции. Использование точки с запятой – хороший стиль, но он не является обязательным. Об употреблении точки с запятой также рассказано в главе 15.

Некоторые инструкции могут иметь несколько форматов. Например, можно использовать *var* с начальным присвоением значения или без него (в следующем примере 10 является простым выражением, значение которого присваивается переменной `x`):

```
var x;       // Простое объявление
var x = 10;  // Объявление с присвоением
```

В данной главе мы обсудим специфический синтаксис каждой инструкции.

Блоки инструкций

Некоторые инструкции фактически содержат другие инструкции, или *подынструкции* (*substatements*), в составе своего синтаксиса. Например, инструкция *if* имеет такой синтаксис:

```
if (expression) substatement;
```

substatement, которая выполняется, только если *expression* дает значение `true`, может быть одиночной инструкцией, например объявлением переменной:

```
if (x == 5) var numFrames = 2;
```

либо это может быть ряд инструкций, объединенных в *блок инструкций*:

```
if (x == 5) {
  var numFrames;
  numFrames = 10;
  play();
}
```

Как видите, блок инструкций представляет собой любое количество инструкций, расположенных на одной или более строках и заключенных в фигурные скобки. Вот три инструкции, разделенные символами точки с запятой, и все они находятся в одной строке:

```
{ statement1; statement2; statement3... }
```

Используя блок инструкций в качестве подынструкции нашей инструкции *if*, мы можем задать несколько инструкций, выполняемых условно (только когда истинно условие инструкции *if*). Это очень удобно.

Везде, где ActionScript ожидает одну инструкцию, можно поместить блок. На самом деле применение блока инструкций иногда даже обязательно. Например, инструкция *function* всегда должна содержать блок, даже если тело объявляемой функции содержит всего одну инструкцию:

```
function aheadTwoFrames() {
  gotoAndStop(_currentframe + 2);
}
```

С целью улучшения читаемости блоки инструкций обычно записываются в таком формате:

```
parent_syntax {
  substatement1;
  substatement2;
  substatement3;
}
```

где *parent_syntax* обозначает инструкцию, для которой мы определяем блок инструкций.

Обратите внимание, что открывающая фигурная скобка находится в конце первой строки после *parent_syntax*. Подынструкции блока инструкций помещаются каждая на своей строке с отступом в два пробела. Закрывающая фигурная скобка, также на отдельной строке, помещается на одном уровне с главной инструкцией. Подынструкции в блоках инструкций должны заканчиваться точкой с запятой, но точек с запятой нет в строках с фигурными скобками.

Наличие отступов – всего лишь соглашение, а не требование синтаксиса. В этой книге соблюдается стиль редактора Flash ActionScript (окно Actions). Допустим и распространен также следующий формат:

```
parent_syntax
{
  substatement1;
  substatement2;
  substatement3;
}
```

Инструкции ActionScript

Теперь, когда вы знаете, какой формат имеет типичная инструкция, бегло ознакомимся с возможностями инструкций ActionScript (табл. 6.1).

Таблица 6.1. Инструкции ActionScript

Инструкция	Синтаксис	Использование
break	`break;`	Прекращает выполнение цикла или инструкции *switch*
case	`case expression:` `  statements`	Задает инструкцию, выполняемую условно в инструкции *switch*
continue	`continue;`	Пропускает оставшиеся в цикле инструкции и начинает новую итерацию с начала цикла

Инструкция	Синтаксис	Использование
default	`default:` `  *statements*`	Задает инструкцию, выполняемую в инструкции *switch*, когда проверяемое выражение не совпадает ни с одним из случаев *case*

do-while	`do {`
	`  *statements*`
	`} while (*expression*);`
		Разновидность цикла *while*, которая обеспечивает выполнение хотя бы одной итерации цикла

пустая инструкция	`;`	Занимает место там, где ожидается инструкция, и используется с *evaluate* в режиме Normal

for	`for (*init*; *test*; *increment*) {`
	`  *statements*`
	`}`
		Многократно выполняет некоторый код (цикл *for*)

for-in	`for (*property* in *object*) {`
	`  *statements*`
	`}`
		Перечисляет свойства объекта

function	`function *name*(*parameters*) {`
	`  *statements*`
	`}`
		Объявляет функцию

if-else if-else	`if (*expression*) {`
	`  *statements*`
	`} else if (*expression*) {`
	`  *statements*`
	`} else {`
	`  *statements*`
	`}`
		Выполняет одну или более инструкций, исходя из условия или ряда условий

ifFrameLoaded	`ifFrameLoaded (*frame*) {`
	`  *statements*`
	`}`
		Выполняет заданный код, если загружен определенный кадр; во Flash 5 признана устаревшей

return	`return;` `return *expression*;`	Завершает выполнение функции и может возвратить ее значение
set	`set (*variable*, *value*);`	Присваивает значение динамически именуемой переменной

switch	`switch (*expression*) {`
	`  *statements*`
	`}`
		Выполняет одну или более инструкций, исходя из одного или нескольких условий (альтернатива if-else if-else)

var	`var *variableName*;` `var *variableName* = *expression*;`	Объявляет переменную и дополнительно может ее инициализировать

while	`while (*expression*) {`
	`  *statements*`
	`}`
		Многократно выполняет блок инструкций (цикл *while*)

with	`with (*objectName*) {`
	`  *statements*`
	`}`
		Выполняет блок инструкций в области видимости указанного объекта

Циклы и условные инструкции

Мы уже сталкивались несколько раз с циклами и условными инструкциями. Оба эти типа инструкций в совокупности обеспечивают основную долю сложной логики в программах. Циклы позволяют многократно выполнять блоки инструкций, а условные инструкции позволяют выполнять блоки инструкций только при определенных обстоятельствах. Полное описание см. в главах 7 и 8.

Инструкции-выражения

Любое выражение может быть независимо использовано в качестве допустимой инструкции, но если выражение не осуществляет никаких действий и с его результатом ничего не производится, то его применение довольно бессмысленно:

```
"hi there";  // Эта инструкция ничего особенного не делает
345 + 5;     // И эта тоже
```

Однако некоторые выражения имеют побочные эффекты, которые изменяют состояние системы путем присваивания значений переменным или свойствам или путем физического изменения окружения Flash. В следующем примере выражение изменяет значение `x`:

```
x = 345 + 5;  // Гораздо больше пользы
```

Вызов функции представляет собой наиболее мощный вид инструкции-выражения. Даже если функция не возвращает полезного значения, она может иметь весьма полезный побочный эффект. Например, *gotoAndStop( )* возвращает значение `undefined`, но имеет важный побочный эффект: перемещает воспроизводящую головку в другой кадр:

```
gotoAndStop(5);  // Значение _currentframe изменилось на 5.
```

Подробнее о вызовах функций мы узнаем в разделе «Выполнение функций» главы 9.

Инструкция var

Инструкция *var* объявляет новую переменную:

```
var x;
```

В инструкции *var* можно задать начальное значение новой переменной:

```
var x = 10;
```

Инструкция *var*, расположенная за пределами функции, создает переменную, областью видимости которой является временная диаграмма, содержащая инструкцию. Внутри функции инструкция *var* создает переменную, локальную для этой функции (то есть такую, которая удаляется по завершении выполнения функции). О переменных подробно рассказано в главе 2.

Инструкция set (set variable)

Большинство присваиваний переменным осуществляется выражениями присваивания в виде инструкции, например:

```
x = 30;
```

Однако для такого вида выражений требуется заранее знать имя переменной. Генерировать имя переменной динамически, во время присваивания, можно посредством инструкции *set*, имеющей следующий синтаксис:

```
set (variable, expression);
```

где *variable* представляет собой строковое выражение, дающее имя переменной, которая участвует в присваивании, а *expression* – это новое значение, присваиваемое переменной. Например:

```
var x;
set ("x", 10);      // x теперь 10
var firstName;
set ("first" + "Name", "jane");  // firstName теперь "jane"
```

В следующем, более хитром примере переменной y в инструкции *set* ничего не присваивается. Вместо этого извлекается значение y ("x"), и это значение выступает в качестве имени переменной, которой и производится присваивание:

```
// Обратите особое внимание на этот пример...
var y = "x";
var x;
set(y, 15);  // x теперь 15, y по-прежнему "x".
```

Во Flash 4 инструкция *set* называлась *Set Variable* и часто применялась для динамического присваивания переменным, имена которых последовательно генерировались программно. Это позволяло программисту моделировать массивы, встроенной поддержки которых во Flash 4 ActionScript не было. Так как массивы включены во Flash 5, *set* теперь применяется редко. Дополнительные сведения о моделировании массивов с помощью *set* можно найти в главе 2. Обычно можно динамически обращаться к переменным с помощью оператора доступа к свойству объекта [ ], например:

```
// Присвоить значение переменной firstName в клипе form_mc.
form_mc["first" + "Name"] = "jane";

// Присвоить значение переменной firstName в текущем клипе.
this["first" + "Name"] = "jane";
```

Однако *set* требуется при динамическом обращении к локальной переменной в функции (потому что локальные переменные не являются свойствами какого-либо видимого объекта ActionScript).

Инструкция function

Подобно тому как инструкция *var* используется для *объявления* (то есть создания) переменных, инструкция *function* применяется для объявления функций. Вот ее синтаксис:

```
function funcName (param1, param2, param3,...paramn) {
  statements
}
```

Ключевое слово `function` начинает инструкцию; *funcName* – это имя объявляемой функции; *param1 ... paramn* определяют параметры, необходимые для выполнения функции (параметры разделяются запятыми); *statements* – список из одной или более инструкций, которые будут выполнены при вызове функции.

Инструкция *function* объявляет функцию, которая может вызываться в дальнейшем, но не выполняет ее. Для выполнения функции ее имя используется в инструкции *вызова функции*, которую мы сейчас обсудим.

Инструкции вызова функции

Инструкция вызова применяется для выполнения встроенной или определенной пользователем функции и содержит имя этой функции и входные данные, необходимые для нее. Термины *call*, *run* и *invoke* часто употребляются для обозначения того, что имя функции использовано для ее вызова. Общий синтаксис инструкции вызова функции:

```
funcName (arg1, arg2, ... argn);
```

где *funcName* представляет собой имя функции, которую надо выполнить, а *arg1 ... argn* – это список *аргументов* (то есть входных значений), которые функция предполагает получить при своем выполнении.

Инструкция вызова функции является исключительно мощным и фундаментальным инструментом: это основной способ управления фильмами Flash. Когда нам требуется каким-то образом воздействовать на фильм, мы часто вызываем функцию. Вот несколько примеров:

```
play();                        // Воспроизводит текущий фильм
gotoAndStop(5);                // Направляет воспроизводящую головку в кадр 5
startDrag("crosshair", true);  // Вызывает перемещение экземпляра
                               // "crosshair" за курсором мыши
```

Вызовы функций используются также с объектами для вызова методов:

```
circle.getArea();
today.getDate();
```

Поскольку экземпляры клипов фильмов представляют собой объекты, в сценариях часто применяются вызовы методов:

```
ball_mc.play()
intro_vc.gotoAndStop("end");
```

Подробнее о вызовах функций мы поговорим в главе 9, а в главе 12 мы узнаем о том, как функции могут стать методами объекта.

Инструкция return

Вызывая функцию, можно передавать ей одно или более значений (*параметров*, или *аргументов*), участвующих в выполнении. Аналогично функция может передавать обратно *возвращаемое значение* (значение, получаемое в результате ее выполнения и пересылаемое обратно вызвавшему). В теле функции инструкция *return* применяется для завершения работы функции и при необходимости возвращения результата. Инструкция *return* имеет один из следующих форматов:

```
return;
return expression;
```

Необязательное выражение *expression*, если оно есть, становится значением, возвращаемым функцией. Инструкция *return* без возвращаемого значения завершает выполнение функции и возвращает значение undefined. Все инструкции *return* осуществляют выход из текущей функции. Заметьте, что инструкция *return* не обязательно должна присутствовать в функции: функция без *return* просто заканчивает работу после выполнения последней инструкции в теле функции и неявно возвращает значение undefined. Подробнее о создании, вызове и завершении работы функций рассказано в главе 9.

Инструкция with

Инструкция *with* предоставляет сокращенный способ обращения к свойствам объекта без необходимости каждый раз указывать имя объекта. Общий вид инструкции *with*:

```
with (object) {
  statements
}
```

Если обращение к свойству происходит в блоке инструкции *with*, проверяется наличие этого свойства у *object*. Иными словами, *with* временно помещает *object* в конец цепочки областей видимости для выполняющегося в данный момент кода. Если у *object* существует указанное свойство, то оно и используется для разрешения ссылки на свойство. Если у *object* нет такого свойства, его поиск выполняется в остальной части текущей цепочки областей видимости, как описано в главе 2.

Следующий пример демонстрирует разницу между выполнением инструкции внутри *with* и за пределами *with*:

```
PI = 10;                    // Установить переменную временной диаграммы, PI
with (Math) {               // Выполнить предложения в контексте Math
  trace("pi is: " + PI);    // Выводит: 3.1459.(PI - свойство Math)
}
trace("PI is: " + PI);      // Выводит: 10 (обращения к Math больше нет)
```

Инструкция *with* может применяться не только для организации удобного доступа к свойствам объекта, но и для вызова методов объекта:

```
x = 10;
y = 11;
with (Math) {
  larger = max(x, y);
}
trace(larger);  // Выводит: 11
```

Нельзя определять новое свойство объекта, являющегося целью (target) инструкции *with*. Например, в предыдущем примере переменная `larger` не существует в объекте *Math*, поэтому присвоение свойству создает новую переменную на временной диаграмме, содержащей инструкцию *with* (не в *Math*!). Следующий код демонстрирует неверную попытку установить переменную в `theClip_mc`:

```
with (theClip_mc) {
  var x = 10;  // x устанавливается в текущей временной диаграмме, а не в theClip_mc
}
```

Можно, однако, с полным правом использовать *with* для воздействия на экземпляры клипов другими способами. Эта инструкция предоставляет удобный способ работы с глубоко вложенными структурами экземпляров. Например, следующий код:

```
this.player_mc.eyes_mc.blinkRate = 30;
this.player_mc.eyes_mc.gotoAndStop("eyesOpen");
```

можно заменить таким:

```
with (this.player_mc.eyes_mc) {
  blinkRate = 30;              // Переустанавливает существующую  переменную
                               // в экземпляре eyes_mc
  gotoAndStop("eyesOpen");     // Функция применяется к экземпляру eyes_mc
}
```

Но *with* не является единственным способом обеспечения такого удобства. Можно просто присвоить наш экземпляр переменной и использовать эту переменную в ссылках:

```
var eyes = this.player_mc.eyes_mc;
eyes.blinkRate = 30;
eyes.gotoAndStop("eyesOpen");
```

Многие разработчики считают, что с переменными проще работать, чем с инструкцией *with*, и что они облегчают чтение кода, но оба подхода имеют право на жизнь. Подробнее о работе с клипами как с объектами мы узнаем из главы 13.

Обратите внимание, что внутри инструкции *with* значение ключевого слова `this` не меняется: оно сохраняет то значение, которое было у него за пределами инструкции *with*. Например:

```
// Поместить следующий код на _level0
// На _level0 создать объект ball со свойством radius
_level0.ball = { radius: 20 };
// Проверить значение 'this' вне инструкции with
trace(this);  // Выводит: _level0
// Start our with statement
with (_level0.ball) {
  // Проверить значение 'this' внутри инструкции with
  trace(this);   // По-прежнему выводит: _level0 (не ball!)
  // Проверить радиус объекта ball
  trace(radius); // Выводит: 20
}
```

Инструкция ifFrameLoaded

Устаревшая инструкция *ifFrameLoaded* выполняет вложенные в нее инструкции, только если загружен указанный кадр. Синтаксис данной инструкции следующий:

```
ifFrameLoaded (frame_expression) {
  statements
}
```

где *frame_expression* – номер кадра или строка, содержащая метку кадра. Если кадр, заданный выражением *expression*, загружен в Player, то выполняются инструкции *statements*. Если нет, то блок инструкций пропускается.

Поскольку в инструкции *ifFrameLoaded* нет части *else*, она была признана устаревшей во Flash 4. В ней и в более поздних версиях следует использовать свойства _totalframes и _framesloaded с инструкциями *if-else*, чтобы создать более гибкий предзагрузчик (preloader). Например:

```
if (_totalframes > 0 && _framesloaded == _totalframes) {
  gotoAndPlay("beginMovie");
} else {
  gotoAndPlay(_currentframe - 1);
}
```

Пустая инструкция

Для полноты картины следует отметить, что допустимо использовать инструкцию, которая не имеет содержания и состоит только из точки с запятой:

```
;
```

Пустая инструкция имеет мало практической ценности и может выступать в качестве заместителя там, где обычно ожидается инструкция. Она не обязательна, если требуется лишь добавить в код пустые строки. ActionScript игнорирует пустые строки, которые позволяют улучшить читаемость кода.

В режиме Normal одиночная точка с запятой появляется, когда в блок кода добавляется действие *evaluate*. При этом можно добавить произвольную инструкцию, введя ее в поле Expression панели Parameters.

Сравнение инструкций с действиями

В среде разработки Flash ActionScript вы не найдете слова «statement» (инструкция). Даже в документации Macromedia термины «Action» (действие, процедура) и «statement» употребляются равноправно. Там говорится, что *действия* – это инструкции, указывающие фильму, что он должен сделать во время воспроизведения.

Употребление термина «Action» как синонима для «statement» скрадывает различие между несколькими разными инструментами ActionScript. Чтобы увидеть это, откройте окно Actions (клавиша F9) и посмотрите папку Actions. В списке действий этой папки (и ее вложенных папок) вы найдете инструкции, которые мы видели ранее в табл. 6.1. Вперемежку с инструкциями вы найдете много функций – *gotoAndPlay( )*, *getURL( )*, *startDrag( )* и т. д. Хотя перечисленные в Actions функции могут применяться в инструкциях, формально они не являются уникальными типами инструкций – это просто встроенные функции. Инструкции, некоторые встроенные функции и обработчики событий названы Macromedia действиями (Actions). В этой книге мы не употребляем общий термин «Action». Каждое действие (Action) мы описываем с термином, соответствующим его формальной роли в языке: *инструкция*, *функция* или *обработчик событий*. Если вы поймете ActionScript с такой традиционной точки зрения, вам будет легче перенести свои навыки в другие языки и понимать общие дискуссии по программированию, не относящиеся непосредственно к ActionScript.

Вперед!

Мы много узнали об инструкциях, но два важных типа инструкций – условные инструкции и циклы – еще ждут своего описания. Мы разберемся с этими типами инструкций в двух следующих главах.

7

Условные инструкции

Большинство рассмотренных примеров кода были *линейными*: все инструкции выполнялись поочередно, начиная с первой и заканчивая последней. Линейный код всегда делает одно и то же. Напротив, *условная инструкция* выполняет действие только в том случае, если удовлетворено указанное условие. В линейном коде интерпретатор может выполнить последовательно инструкции A и B, а затем – инструкцию C. А условные инструкции позволяют потребовать от интерпретатора сначала выполнить инструкцию A, а затем – *либо* инструкцию B, *либо* инструкцию C, в зависимости от условия X.

Условные инструкции позволяют создавать и контролировать ситуации, имеющие более одного возможного исхода. Допустим, например, что мы хотим создать сайт, защищенный паролем. При попытке пользователя зарегистрироваться пароль оказывается либо правильным, и пользователю разрешается зайти на сайт, либо неверным, и пользователь видит сообщение об ошибке. Для этих двух исходов в нашем сценарии фильма требуются совершенно различные блоки кода. Один блок должен послать воспроизводящую головку Flash в кадр, содержащий приветственный экран сайта, а другой – в кадр с сообщением об ошибке. Но только один из этих блоков должен быть выполнен во время регистрации пользователя. Условная инструкция позволяет выполнить нужный блок и пропустить ненужный.

Как интерпретатор узнает, какой блок кода должен быть выполнен? Когда мы определяем условную инструкцию, то указываем условие, которое должно быть удовлетворено, чтобы выполнялся *первый* блок кода. Если условие не выполнено, может быть выполнен альтернативный блок (который, в свою очередь, может содержать свое условие). В сущности, мы устанавливаем в программе логику блок-схемы, которую можно записать на псевдокоде следующим образом:

```
if (первое условие выполнено) {
  // Выполнить этот код, если справедливо первое условие
} else if (второе условие выполнено) {
  // Выполнить этот код, если справедливо второе условие
} ...otherwise {
```

```
  // Выполнить этот код, если не выполнено ни одно из предыдущих условий
}
```

Конечно, каждое условие надо описать на языке, понятном интерпретатору. Это не проблема – все дело в синтаксисе, который мы сейчас изучим. Смысл в том, что все условные инструкции разрешают или запрещают выполнение блока кода исходя из заданного условия. Теперь посмотрим на практике, как работают условные инструкции.

Инструкция if

Инструкция *if* – самая обычная условная инструкция. С ее помощью в коде создается двухлучевое разветвление, как развилка на дороге. Инструкция *if* содержит одну или более подынструкций, которые выполняются, только если удовлетворено заданное условие. Синтаксис инструкции *if* следующий:

```
if (condition) {
  substatements
}
```

Инструкция *if* начинается, как и можно ожидать, с ключевого слова `if`. Условие *condition*, которое должно быть удовлетворено, чтобы выполнились инструкции *substatements*, заключается в круглые скобки. *substatements* – это одна или более инструкций ActionScript. Каждая подынструкция должна занимать отдельную строку и завершаться точкой с запятой. Инструкция *if* в целом оканчивается закрывающей фигурной скобкой (}) без замыкающей точки с запятой.

Условие *condition* инструкции *if* может быть любым допустимым выражением. Когда выполняется инструкция *if*, интерпретатор проверяет значение этого выражения (называемого *проверочным выражением*). Если оно равно `true`, то выполняются *substatements*. В противном случае *substatements* не выполняются. Ниже в качестве проверочных выступают простые булевы выражения:

```
if (true) {
  trace("Условие выполнено!");  // Эта инструкция будет выполнена
}
if (false) {
  trace("Условие выполнено!");  // Эта инструкция не выполняется
}
```

Конечно, нет практического смысла использовать булевы литералы в качестве проверочных выражений, поскольку их значения никогда не меняются. Их место у нас займут сложные выражения, *возвращающие* булевы значения. Например, выражения, в которые входит операция сравнения, возвращают булево значение и отлично подходят в качестве проверочных:

```
var pointerX = _root._xmouse;  // Положение мыши по горизонтали
// Если pointerX > 300 дает true...
if (pointerX > 300) {
  // ...выполняется этот блок
```

```
  trace("Мышь за отметкой 300 пикселов ");
}
```

Теперь о самом главном: проверочное выражение условной инструкции не обязательно должно давать в результате булево значение – годится любое выражение. В качестве проверочного выражения условной инструкции может выступать число или строка:

```
if ("hi") {
  trace("Условие выполнено!");
}
if (4) {
  trace("Условие выполнено!");
}
```

Выражения «hi» и 4 не являются булевыми, а код работает. Почему? Все дело в чудесах преобразований типов, показанных в табл. 3.3. Когда проверочное выражение условной инструкции не является булевым, интерпретатор преобразует выражение в булево. Например, интерпретатор преобразует «hi» в `false`, потому что все нечисловые строки преобразуются в `false`, если встречаются в булевом контексте. Поэтому условие не выполнено, и первая инструкция *trace( )* тоже не выполняется. Аналогично интерпретатор преобразует число 4 в `true` (любое ненулевое число преобразуется в `true`), поэтому вторая инструкция *trace( )* выполняется.

Все наши прежние усилия с преобразованием типов данных окупились! Вот несколько основных практических примеров. Попытайтесь определить для каждой инструкции, будет ли она выполнена:

```
x = 3;
if (x) {
  trace("x не равен нулю");
}
```

В следующем примере используется оператор *ИЛИ*, описанный в главе 5:

```
lastName = "";
firstName = "";
if (firstName != "" || lastName != "") {
  trace("Welcome " + firstName + " " + lastName);
}
```

Наконец, проверим, существует ли экземпляр клипа (в булевом контексте клипы преобразуются в `true`):

```
if (theClip_mc) {
  theClip_mc._x = 0;  // Если theClip_mc существует, поместить его
                      // на левый край сцены
}
```

Инструкция else

С помощью одиночной инструкции *if* можно вызвать выполнение одиночного блока кода. Добавив блок *else*, можно выбрать, который из двух блоков

кода будет исполнен. Синтаксически инструкция *else* представляет собой расширение инструкции *if*:

```
if (condition) {
  substatements1
} else {
  substatements2
}
```

где *condition* может быть любым допустимым выражением, *substatements1* будут выполнены, если *condition* имеет значение `true`, а *substatements2* будут выполнены, если *condition* имеет значение `false`. Иными словами, инструкция *else* прекрасно подходит для отражения взаимоисключающего решения: один блок кода будет выполнен, а другой – нет.

Проиллюстрируем сказанное примером:

```
var lastName = "Grossman";
var gender = "male";
if (gender == "male") {
  trace("Good morning, Mr. " + lastName + ".");
} else {
  trace("Good morning, Ms. " + lastName + ".");
}
```

Инструкция *else* часто действует как резервный план для инструкции *if*. Вспомните наш пример веб-сайта, защищенного паролем. Если пароль верный, мы позволяем пользователю зайти на сайт, в противном случае выводим сообщение об ошибке. Вот код, реализующий проверку пароля (допустим, что `userName` и `password` содержат введенные пользователем данные, а `validUser` и `correctPassword` являются правильными значениями этих переменных для регистрации):

```
if (userName == validUser && password == correctPassword) {
  this.gotoAndPlay("intro");
} else {
  this.gotoAndStop("loginError");
}
```

Условный оператор

Простые условные инструкции из двух частей удобно записывать с помощью условного оператора (`?:`). Условный оператор состоит из трех частей, в которых можно узнать аналоги компонентов инструкций *if-else*:

```
condition ? expression1 : expression2;
```

В условной операции, когда *condition* имеет значение `true`, вычисляется и возвращается *expression1*. В противном случае вычисляется и возвращается *expression2*. Условный оператор представляет собой более короткую запись следующей условной инструкции:

```
if (condition) {
  expression1
```

```
} else {
  expression2
}
```

Как и условная инструкция, условный оператор может применяться для управления логикой выполнения программы:

```
// Если команда "go", воспроизвести фильм, иначе остановиться
command == "go" ? play() : stop();
```

Условный оператор предоставляет также краткий способ присвоения значений переменным:

```
var guess = "c";
var answer = "d";
var response = (guess == answer) ? "Right" : "Wrong";
```

Это эквивалентно следующему коду:

```
var guess = "c";
var answer = "d";
if (guess == answer) {
  response = "Right";
} else {
  response = "Wrong";
}
```

Инструкция else if

Используя *if* и *else*, можно избирательно выполнять один из двух блоков кода. С помощью *if* и *else if* можно избирательно выполнять один (или даже не выполнять ни одного) из *неограниченного* количества блоков кода. Как и *else*, *else if* является синтаксическим расширением инструкции *if*:

```
if (condition1) {
  substatements1
} else if (condition2) {
  substatements2
} else if (condition3) {
  substatements3
} else {
  substatements4  // Если ни одно из условий не подошло
}
```

где *condition1*, *condition2* и *condition3* должны быть допустимыми выражениями, а *substatements1* будут выполнены, если *condition1* имеет значение `true`. Если *condition1* имеет значение `false`, то *substatements2* будут выполнены только в случае, если *condition2* имеет значение `true`. В противном случае вычисляется *condition3*, и т. д. для всех имеющихся инструкций *else if*. Если ни одна из проверочных инструкций не имеет значения `true`, будет выполнен блок инструкций завершающей инструкции *else*. Например, можно написать программу проверки регистрации с «умными» сообщениями об ошибках:

```
if (userName != validUser) {
  message = "Пользователь не найден. Попробуйте снова.";
  this.gotoAndStop("loginError");
} else if (password != correctPassword) {
  message = "Пароль неверен. Попробуйте снова.";
  this.gotoAndStop("loginError");
} else {
  this.gotoAndPlay("intro");
}
```

Обратите внимание, что инструкция *else if* является просто объединением *else* с вложенной инструкцией *if*. Хотя следующие два участка кода эквивалентны, первый из них значительно легче читать:

```
// Обычный синтаксис "else if"
if (x > y) {
  trace("x больше y");
} else if (x < y) {
  trace("x меньше y");
} else {
  trace("x равен y");
}
// Расширенная цепочка if/else
if (x > y) {
  trace("x больше y");
} else {
  if (x < y) {
    trace("x меньше y");
  } else {
    trace("x равен y");
  }
}
```

Инструкция switch

Инструкция *switch*, появившаяся во Flash Player 6, позволяет выполнить один (в некоторых случаях несколько) из ряда возможных блоков кода, исходя из значения одного проверочного выражения. Иногда инструкцию *switch* легче написать и прочесть, чем инструкцию *if*. Например, в следующей инструкции *switch* мы приветствуем пользователя сообщением, соержание которого зависит от значения проверочного выражения `gender`:

```
var surname = "Porter";
var gender = "male";
switch (gender) {
  case "femaleMarried":
    trace("Hello Mrs. " + surname);
    break;
  case "femaleGeneric":
    trace("Hello Ms. " + surname);
    break;
```

```
    case "male":
      trace("Hello Mr. " + surname);
      break;
    default:
      trace("Hello " + surname);
  }
```

Наша инструкция *switch* пытается найти совпадение значения `gender` с одним из выражений *case*: «femaleMarried», «femaleGeneric» или «male». Поскольку `gender` соответствует выражению «male», выполняется подынструкция *trace ("Hello Mr. " + surname);*. Если проверочное выражение не соответствует ни одному случаю, то выполняется инструкция по умолчанию *trace ("Hello " + surname);*.

Общий формат инструкции *switch* таков:

```
switch (testExpression) {
  case expression1:
    substatements1
    break;
  case expression2:
    substatements2
    break;
  default:
    substatements3
    break;
}
```

Здесь *testExpression* – значение, которое интерпретатор попытается сопоставить с каждым из заданных выражений *case*, сверху донизу. Выражения *case* записываются с помощью метки *case* и двоеточия. Если *testExpression* совпадает с выражением *case*, то выполняются все инструкции, непосредственно следующие за этой меткой *case*, в том числе во всех последующих блоках *case*! Избежать выполнения последующих блоков можно с помощью инструкции *break*, помещаемой в конец каждого блока. Если ни одно выражение *case* не совпадает с *testExpression*, выполняются инструкции, следующие за меткой *default*.

Хотя метка *default* обычно указывается последней, допускается использование ее в любом месте инструкции *switch*. Более того, метка *default* не обязательно должна присутствовать в инструкции *switch*. Если *default* не задана и *testExpression* не совпадает с выражениями *case*, выполнение продолжается с кода, следующего за блоком инструкций *switch* (то есть весь код внутри *switch* пропускается). Однако если отсутствуют и метки *case*, и метка *default*, то инструкция *switch* генерирует ошибку (в отличие от JavaScript, где допускаются пустые инструкции *switch*).

Для того чтобы передать в *switch* не одно значение, а диапазон, применяется следующий прием:

```
switch (true) {
  case (x < 10):
```

```
      trace("x меньше 10.");
      break;
    case (x >= 10 && x <= 20):
      trace("x между 10 и 20.");
      break;
    case (x > 20):
      trace("x больше 20.");
      break;
  }
```

Обратите внимание, что если в этом примере оказываются истинными несколько выражений *case*, то выполняется код, следующий за первым истинным выражением.

Для проверки сразу нескольких случаев применяется следующий прием «проваливания» (обратите внимание, что инструкция *break* намеренно опущена в *case 1* и в *case 3*):

```
x = 3;

switch (x) {
  case 1:
  case 3:
  case 5:
    trace("x равно 1, 3 или 5");
    break;
  default:
    trace("x не равно 1, 3 или 5");
}
```

В инструкции *switch* при сравнении *testExpression* с *case* неявно используется оператор строгого равенства (===), а не обычного равенства (==). Их различия объясняются в главе 5. Выражения, следующие за ключевыми словами switch и case, могут принадлежать любому типу данных и необязательно должны быть литералами. Поскольку инструкция *switch* выполняется сверху вниз, в начале следует перечислить те выражения *case*, совпадение которых с *testExpression* наиболее вероятно (очевидно, это относится только к тем случаям, когда порядок инструкций *case* не влияет на результат инструкции *switch*).

Хотя инструкция *switch* появилась во Flash Player 6, код, в котором она применяется, можно экспортировать из среды разработки Flash MX в фильмы форматов Flash Player 5 и Flash Player 4. Однако при экспорте в форматы Flash 4 и Flash 5 для сравнении выражений *switch* применяется оператор обычного равенства (==), а не строгого равенства (===). В любом случае *switch* не дает выигрыша в скорости по сравнению с аналогичной конструкцией *if-else if-else*.

Инструкции case и default служат конкретными примерами более общей структуры, называемой *меткой инструкции* (*statement label*). Метки инструкций входят в состав ECMAScript v3, но в ActionScript поддерживает только case и default.

Эмуляция инструкции switch во Flash 5

Инструкция *switch* не поддерживается в среде разработки Flash 5, но ее можно эмулировать с помощью последовательности инструкций *if-else if-else*, например:

```
var surname = "Porter";
var gender = "male";
if (gender == "femaleMarried") {
  trace("Hello Mrs. " + surname);
} else if (gender == "femaleGeneric") {
  trace("Hello Ms. " + surname);
} else if (gender == "male") {
  trace("Hello Mr. " + surname);
} else {
  trace("Hello " + surname);
}
```

Компактный синтаксис условных инструкций

В этой книге мы используем блоки инструкций (инструкции, заключенные в фигурные скобки) во всех условных инструкциях, даже если в блоке всего одна инструкция:

```
if (x == y) {
  trace("x равно y");
}
```

Для краткости ActionScript разрешает опускать фигурные скобки, когда в условной инструкции лишь одна подынструкция. Единственную подынструкцию можно вполне законно поместить прямо после инструкции *if* или *else if* без фигурных скобок, например:

```
if (x == y) trace ("x равно y");
```

Или даже на следующей строке:

```
if (x == y)
  trace ("x равно y");
```

Некоторые программисты считают такой стиль неудобным для чтения и более подверженным ошибкам, но несомненно, что при этом экономится место в исходном коде.

Вперед!

Условные инструкции составляют крайне важную часть ActionScript. Они дают точный контроль над выполнением кода. В следующей главе мы рассмотрим другой важный тип инструкций, управляющих выполнением, – *циклы*.

8

Инструкции цикла

В главе 7 мы узнали, что *условная инструкция* вызывает однократное выполнение блока инструкций, если его проверочное выражение имеет значение `true`. В *цикле* же блок инструкций выполняется многократно, пока проверочное выражение сохраняет значение `true`.

Циклы бывают различных типов: *while*, *do-while*, *for* и *for-in*. Первые три обладают сходным действием, различаясь синтаксисом. Последний тип цикла, *for-in*, является особым видом цикла, ориентированным на работу с объектами. Мы начнем исследование циклов с инструкции *while* – простейшего для понимания вида цикла.

Цикл while

По структуре инструкция *while* во многом похожа на инструкцию *if*: главная инструкция содержит блок подынструкций, выполняемых только тогда, когда заданное условие имеет значение `true`:

```
while (condition) {
  substatements
}
```

Если *condition* имеет значение `true`, то выполняются подынструкции *substatements*. Но, в отличие от *if*, по завершении последней из подынструкций выполнение повторяется с начала инструкции *while* (то есть интерпретатор «зацикливается» на начале инструкции *while*). Второй проход через инструкцию *while* осуществляется точно так же, как первый: вычисляется условие и, если оно по-прежнему оказывается равным `true`, снова выполняются *substatements*. Этот процесс продолжается, пока *condition* не станет равным `false`, и тогда выполнение продолжится с инструкций, следующих в сценарии за *while*.

Вот пример очень простого цикла:

```
var i = 3;
while (i < 5) {
  trace("x меньше 5");
}
```

Этот пример совершенно верен с точки зрения синтаксиса цикла *while*, но, надо полагать, ошибочен. Чтобы узнать причину, проследим за тем, как интерпретатор выполняет данный пример.

Интерпретатор начинает с инструкции, предшествующей *while*, а именно *var i = 3*, которая присваивает переменной `i` значение 3. Поскольку переменная `i` используется в проверочном выражении цикла, этот шаг часто называется *инициализацией* цикла. Затем интерпретатор начинает выполнять инструкцию *while*, вычисляя проверочное выражение: *i < 5*. Поскольку `i` равно 3, а 3 меньше 5, значением проверочного выражения является `true`, и интерпретатор выполняет инструкцию *trace( )* в цикле.

Когда это сделано, наступает момент перезапуска цикла. Интерпретатор снова проверяет значение проверочного выражения. Значение переменной `i` не изменилось, поэтому проверочное выражение по-прежнему `true`, и интерпретатор снова выполняет инструкцию *trace( )*. Закончив выполнение тела цикла, надо повторить цикл. Что дальше? Переменная `i` все еще не изменилась, поэтому проверочное выражение по-прежнему равно `true`, и интерпретатор должен выполнять предложение *trace( )* снова, и снова, и снова – до бесконечности. Поскольку проверочное выражение всегда возвращает `true`, невозможно выйти из цикла – Flash навсегда попал в *бесконечный цикл*. Как мы увидим позже, в ActionScript бесконечный цикл вызывает ошибку.

Наш цикл бесконечен, поскольку в нем отсутствует *обновляющая инструкция*, которая изменяет значение переменной, включенной в проверочное выражение. Обновляющая инструкция обычно приводит к тому, что проверочное выражение в конечном итоге становится равным `false`, благодаря чему цикл завершается. Подправим наш бесконечный цикл, добавив к нему обновляющую инструкцию:

```
var i = 3;
while (i < 5) {
  trace("x меньше 5");
  i++;
}
```

Инструкция *i++;* находится в конце тела цикла. Проходя через цикл, интерпретатор, как и прежде, выполняет инструкцию *trace( )*, но выполняет также инструкцию *i++;*, которая прибавляет 1 к переменной `i`. При каждом проходе цикла значение `i` возрастает. После второго прохода значение `i` равно 5, поэтому проверочная инструкция *i < 5* получает значение `false`. Благодаря этому цикл благополучно завершается после двух итераций.

Наша обновляющая инструкция выполняет в цикле базовую функцию – ведет счетчик. Переменная `i` (называемая *счетчиком*) пробегает предсказуемую последовательность чисел, что очень подходит для методических задач, таких как создание дубликатов клипов или доступ к элементам массива. Вот как создаются пять копий клипа `square` без использования цикла:

```
// Дать каждому клипу последовательное имя и поместить на отдельный уровень
duplicateMovieClip("square", "square1", 1);
duplicateMovieClip("square", "square2", 2);
```

```
duplicateMovieClip("square", "square3", 3);
duplicateMovieClip("square", "square4", 4);
duplicateMovieClip("square", "square5", 5);
```

А вот как та же самая операция выполняется в цикле:

```
var i = 1;
while (i <= 5) {
  duplicateMovieClip("square", "square" + i, i);
  i++;
}
```

Представьте себе разницу, если надо изготовить 100 копий `square`!

Циклы чрезвычайно полезны для работы с данными, особенно с теми, которые хранятся в массивах. В примере 8.1 показан цикл, который выводит все элементы массива в окно Output. Обратите внимание, что первый элемент имеет номер 0, а не 1.

Пример 8.1. Вывод массива в цикле

```
var people = ["John", "Joyce", "Margaret", "Michael"];  // Создание массива
var i = 0;
while (i < people.length) {
  trace("people element " + i + " is " + people[i]);
  i++;
}
```

В окне Output получаем следующий результат:

```
people element 0 is John
people element 1 is Joyce
people element 2 is Margaret
people element 3 is Michael
```

Обратите внимание, что переменная `i` участвует в проверочном выражении и одновременно выступает в качестве индекса массива, что типично для циклов. Ниже мы снова используем `i` как аргумент функции *charAt( )*:

```
var city = "Toronto";
trace("Буквы в переменной 'city':");
var i = 0;
while (i < city.length) {
  trace(city.charAt(i));
  i++;
}
```

В окне Output мы увидим следующее:

```
Буквы в переменной 'city':
T
o
r
o
n
t
o
```

Наконец, применим цикл не для разделения данных, а для построения предложения из последовательности слов, хранящихся в массиве:

```
var words = ["Toronto", "is", "not", "the", "capital", "of", "Canada"];
var sentence;
var i = 0;
while (i < words.length) {
  sentence += words[i];       // Добавить к предложению текущее слово.
  // Если это не последнее слово...
  if (i < words.length - 1) {
    sentence += " ";          // ...вставить пробел.
  } else {
    sentence += ".";          // ...либо завершить точкой.
  }
  i++;
}
trace(sentence);              // Выводит: "Toronto is not the capital of Canada."
```

Почти все циклы содержат какого-либо рода счетчик (иногда называемый *итератором*, или *переменной индексирования*). Счетчики позволяют последовательно перебирать данные. Этот прием особенно удобен, когда максимальная граница счетчика определена с помощью свойства `length` массива или строки, которые должны быть обработаны, как в предыдущем примере.

Возможно создание цикла, завершение которого не зависит от счетчика. Когда проверочное выражение цикла станет равно `false`, цикл будет завершен. Вот, например, как можно просмотреть стек уровней документа проигрывателя Flash для нахождения первого свободного уровня:

```
var i = 0;
while (typeof eval("_level" + i) == "movieclip") {
  i++;
}
trace("Первый свободный уровень " + i);
// Теперь можно загрузить фильм на свободный уровень
loadMovieNum("myMovie.swf", i);
```

Подробнее об уровнях документов можно прочитать в разделе «Порядок помещения фильмов и экземпляров в стек» главы 13.

Терминология циклов

В предыдущем разделе мы познакомились с несколькими новыми терминами. Рассмотрим их более формально, чтобы лучше понимать работу с циклами:

Инициализация

Инструкция или выражение, используемые в проверочном выражении цикла.

Проверочное выражение

Выражение, при истинности которого выполняются подынструкции в теле цикла. Часто называется *условием* (*condition*) или *проверкой* (*test*), а иногда *управлением* (*control*).

Модификация (*update*)

Инструкции, которые модифицируют переменные, используемые в проверочном выражении перед последующей проверкой. Типичная модифицирующая инструкция инкрементирует или декрементирует счетчик цикла.

Итерация

Одно полное выполнение проверочного выражения и инструкций в теле цикла. Иногда ее также называют одним *циклом* (*loop*), или одним *проходом* (*pass*).

Вложенный цикл (*nested loop*)

Цикл, включающий в себя другой цикл, благодаря чему можно совершать обход неких двумерных данных. Например, можно циклически пройти по каждой строке для каждой колонки таблицы. Внешний цикл, или цикл верхнего уровня, будет перемещаться по колонкам, а внутренний цикл будет перемещаться по строкам в каждой колонке.

Итератор, или индексная переменная

Переменная, значение которой уменьшается или увеличивается при каждом проходе цикла; обычно применяется для подсчета или последовательного перебора данных. Итераторы циклов часто называют *счетчиками*. Обычно итераторы именуют `i`, `j` и `k` или иногда `x`, `y` и `z`. В последовательности вложенных циклов `i` обычно является итератором верхнего уровня, `j` – итератором первого вложенного цикла, `k` – итератором второго вложенного цикла и т. д. Но для ясности этой переменной можно дать любое понравившееся имя. Например, можно использовать в качестве имени переменной `charNum`, чтобы помнить, что это текущий символ строки.

Тело цикла

Блок инструкций, выполняемых, когда удовлетворено условие цикла. Тело может не выполняться вообще или выполняться тысячи раз.

Заголовок цикла (*loop header*) *или управление циклом* (*loop control*)

Часть цикла, содержащая ключевое слово инструкции цикла (*while*, *for*, *do-while* или *for-in*) и условие цикла. Управление циклом зависит от типа цикла. В цикле *for* управление включает в себя инициализацию, проверку и модификацию; в цикле *while* управление содержит только проверочное выражение.

Бесконечный цикл

Цикл, который выполняется бесконечно, так как проверочное выражение никогда не дает значение `false`. Бесконечные циклы вызывают в ActionScript ошибку, о чем рассказывается ниже в этой главе в разделе «Максимальное количество итераций».

Цикл do–while

Как мы уже видели, инструкция *while* позволяет интерпретатору многократно выполнять блок кода, пока заданное условие сохраняет значение `true`. Структура цикла *while* такова, что его тело будет целиком пропущено, если условие окажется невыполненным при первой проверке. Предложение *do-while* гарантирует, что тело цикла будет выполнено хотя бы один раз. Тело цикла *do-while всегда* выполняется при первом проходе цикла. Синтаксис инструкции *do-while* в некотором роде обратен синтаксису инструкции *while*:

```
do {
  substatements
} while (condition);
```

Начинается цикл с ключевого слова `do`, затем следуют находящиеся в теле подынструкции *substatements*. При первом проходе интерпретатором цикла *do-while* подынструкции *substatements* выполняются раньше какой-либо проверки условия *condition*. В конце блока *substatements*, если *condition* имеет значение `true`, цикл начинается заново, и *substatements* выполняются снова. Цикл выполняется многократно, пока *condition* не станет равным `false`, на чем выполнение инструкции *do-while* заканчивается. Обратите внимание, что за скобками, содержащими условие *condition*, необходимо поставить точку с запятой.

Очевидно, что цикл *do-while* удобен, если надо выполнить задачу хотя бы один раз, а возможно и еще несколько раз. В примере 8.2 мы копируем ряд клипов мерцающей звезды из клипа с именем `starParent` и случайным образом размещаем их на рабочем столе. В нашей галактике всегда будет хоть одна звезда, даже если `numStars` равно 0.

Пример 8.2. Применение цикла do-while

```
var numStars = 5;
var i = 1;
do {
  // Создаем копию клипа starParent
  duplicateMovieClip(starParent, "star" + i, i);
  // Помещаем полученный клип в случайную позицию
  _root["star" + i]._x = Math.floor(Math.random() * 551);
  _root["star" + i]._y = Math.floor(Math.random() * 401);
} while (i++ < numStars);
```

А вы заметили, что мы тайком изменили переменную `i` в проверочном выражении? Вспомните из главы 5, что постфиксный оператор инкрементирования возвращает значение своего операнда, а также прибавляет к этому операнду единицу. Оператор инкрементирования очень удобен (и часто применяется) при работе с циклами.

Цикл for

Цикл *for* по сути синонимичен циклу *while*, но записывается более компактно. Проще всего заметить, что заголовок цикла может, помимо проверочного выражения, содержать инструкции инициализации и модификации.

Вот синтаксис цикла *for*:

```
for (initialization; condition; update) {
  substatements
}
```

Цикл *for* аккуратно помещает в заголовок цикла ключевые компоненты цикла, разделенные точкой с запятой. Перед первой итерацией цикла *for* выполняется инструкция `initialization` (один и только один раз). Обычно она устанавливает начальное значение одной или нескольких переменных итератора. Как и в других циклах, если `condition` имеет значение `true`, выполняются подынструкции `substatements`. В противном случае цикл завершается. Несмотря на то что инструкция `update` находится в заголовке цикла, она выполняется в *конце* каждой итерации цикла, перед очередной проверкой `condition`, позволяющей узнать, должен ли продолжаться цикл. Вот типичный цикл *for*, который просто считает от 1 до 10:

```
for (var i = 1; i <= 10; i++) {
  trace("Теперь обрабатывается число " + i);
}
```

Понять, как работает цикл *for*, проще, если посмотреть на его эквивалент, составленный с помощью синтаксиса цикла *while*:

```
var i = 1;
while (i <= 10) {
  trace("Теперь обрабатывается число " + i);
  i++;
}
```

Привыкнув к синтаксису *for*, вы обнаружите, что он экономит место и позволяет легко интерпретировать тело и управление цикла.

Множественные итераторы в циклах for

Если требуется управлять в цикле более чем одним фактором, можно ввести более одной переменной-итератора. Цикл *while* с несколькими итераторами может выглядеть так:

```
var i = 1;
var j = 10;
while (i <= 10) {
  trace("Going up " + i);
  trace("Going down " + j);
  i++;
  j--;
}
```

Того же результата можно достичь в инструкции *for* с помощью оператора «запятая»:

```
for (var i = 1, j = 10; i <= 10; i++, j--) {
  trace("Going up " + i);
  trace("Going down " + j);
}
```

Обратите внимание, что ключевое слово *var* указано только один раз – оно применяется ко всем заданным переменным.

Цикл for-in

Инструкция *for-in* представляет собой специальный цикл, предназначенный для перечисления свойств объекта. Новички в программировании могут пока пропустить этот раздел и вернуться к нему после чтения главы 12.

В отличие от других инструкций цикла, которые повторяют группу инструкций, пока заданное проверочное выражение не даст значение `false`, цикл *for-in* осуществляет по одной итерации для каждого свойства заданного объекта. Инструкции *for-in* не требуют явной обновляющей инструкции, поскольку количество итераций цикла определяется количеством свойств рассматриваемого объекта. Синтаксис цикла *for-in* выглядит так:

```
for (var thisProp in object) {
  substatements;  // Подынструкции, которые каким-то образом используют thisProp
}
```

Подынструкции *substatements* выполняются по одному разу для каждого свойства *object*; *object* представляет собой ссылку на любой действительный объект; *thisProp* – имя переменной (не следует жестко прописывать здесь имя существующего свойства объекта). На каждой итерации цикла переменная *thisProp* автоматически получает строку, которая содержит имя свойства объекта, перечисляемого в данный момент. Это значение строки можно использовать во время каждой итерации, чтобы получить доступ к каждому свойству и обработать его. В простейшем примере цикл *for-in* перечисляет свойства объекта. Ниже мы создаем объект, а затем перечисляем его свойства с помощью цикла *for-in*:

```
var ball = new Object();
ball.radius = 12;
ball.color = "red";
ball.style = "beach";
for (var prop in ball) {
  trace("ball has the property " + prop);
}
```

Поскольку `prop` хранит имена свойств `ball` в виде строк, для извлечения этих свойств можно использовать `prop` с оператором `[]`, например:

```
for (var prop in ball) {
  trace("ball." + prop + " is " + ball[prop]);
}
```

Извлечение значений свойств с помощью цикла *for-in* предоставляет также отличную возможность обнаружения клипов, присутствующих на временной диаграмме. Применение цикла *for-in* в качестве детектора клипов показано в примере 3.1.

Обратите внимание, что свойства объекта, просматриваемые в цикле *for-in*, нумеруются в непредсказуемом порядке. Кроме того, инструкции *for-in* не всегда перечисляют все свойства объекта. Если объект определен пользователем, перечисляются все свойства, включая унаследованные. Но инструкция *for-in* пропускает некоторые свойства встроенных объектов. Например, циклом *for-in* не перечисляются методы встроенных объектов. Собираясь применять инструкцию *for-in* для работы со свойствами встроенного объекта, сначала создайте проверочный цикл для определения доступных свойств объекта, например:

```
thisFormat = new TextFormat();
for (var prop in thisFormat) {
  trace("Property found: " + prop);
}
```

Если исследуемый `object` является массивом, то значением `thisProp` оказывается строка, содержащая целое число – индекс перечисляемого элемента.

Чтобы не позволить свойству объекта, определенному пользователем, появиться в цикле *for-in*, можно обратиться к недокументированной функции *ASSetPropFlags( )*:

```
// Создать новый объект ball со свойствами radius и color.
var ball = new Object();
ball.radius = 10;
ball.color = "red";

// Скрыть ball.radius от циклов for-in.
ASSetPropFlags(ball, ["radius"], 1);

// Выводит: color (radius скрыто)
for (var prop in ball) {
  trace(prop);
}
```

Этот прием официально не документирован Macromedia и может быть в будущем заменен более очевидной функцией. Данная возможность есть во Flash Player 5 и Flash Player 6. Дополнительные сведения можно найти по адресу:

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?ASSetPropFlags

Во Flash Player 5 текстовые поля ввода без значений по умолчанию не перечисляются в цикле *for-in*. По этой причине код проверки формы, который должен обнаружить пустые поля, будет работать неправильно, если не объявить эти текстовые поля в явном виде как переменные на временной диаграмме, где они находятся. Во Flash Player 6 все текстовые поля представляются объектами и нумеруются в цикле *for-in*, даже когда они пусты.

Инструкцию *for-in* можно также использовать для извлечения элементов массива, и тогда она принимает вид:

```
for (var thisElem in array) {
  substatements; // Обычно инструкции как-то используют thisElem
}
```

В следующем примере перечисляются элементы массива:

```
var myArr = [123, 234, 345, 456];
for (var elem in myArr) {
  trace(myArr[elem]);
}
```

Досрочное завершение цикла

В простом цикле проверочное выражение является единственным фактором, определяющим прекращение выполнения цикла. Когда проверочное выражение простого цикла дает значение `false`, цикл завершается. Однако если цикл становится более сложным, может потребоваться остановить его выполнение с помощью инструкции *break* независимо от значения проверочного выражения. Родственная инструкция *continue* завершает текущую итерацию цикла и возобновляет выполнение с новой итерации.

Инструкция break

Инструкция *break* прекращает выполнение текущего цикла или инструкции *switch* (см. главу 7). Синтаксис скромен:

```
break
```

Единственное требование состоит в том, что инструкция *break* должна находиться в теле цикла или в инструкции *switch*.

Инструкция *break* предоставляет способ остановить процесс, продолжать который больше нет необходимости. Представим себе цикл *for-in* в программе проверки формы, которая обходит переменные текстовых полей ввода на временной диаграмме. Обнаружив пустое поле ввода, мы предупреждаем пользователя о том, что он неправильно заполнил форму. Процесс можно прекратить, выполнив инструкцию *break*. Соответствующий код показан в примере 8.3. Обратите внимание, что в примере предполагается существование клипа с именем `form_mc`, который содержит ряд объявленных переменных ввода с именами `input01_txt`, `input02_txt` и т. д.

Пример 8.3. Простая программа проверки полей формы

```
for (var prop in form_mc) {
  // Если это свойство является одним из текстовых полей ввода
  if (prop.indexOf("input") != -1) {
    // Если в поле формы пусто, прервать операцию
    if (form[prop].text == "") {
      displayMessage = "Пожалуйста, заполните форму полностью.";
```

```
        break;
      }
      // Все подынструкции после команды break пропускаются,
      // когда выполняется break
    }
  }
  // Отсюда возобновляется выполнение после цикла, окончание которого вызвано
  // командой break или присваиванием проверочному выражению значения false
```

Инструкция continue

Инструкция *continue* аналогична инструкции *break*, поскольку прерывает текущую итерацию цикла, но в отличие от *break* возобновляет выполнение цикла с очередной естественной итерации. Синтаксис инструкции *continue* прост:

```
continue
```

Во всех типах циклов инструкция *continue* прерывает текущую итерацию тела цикла, но при возобновлении цикла могут быть некоторые различия в зависимости от типа инструкции цикла. В циклах *while* и *do-while* проверочное выражение проверяется перед возобновлением цикла. Но в цикле *for* перед проверкой условия выполняется модификация цикла. А в цикле *for-in* новая итерация начинается с очередного свойства рассматриваемого объекта (или элемента массива), если таковое существует.

С помощью инструкции *continue* можно сделать выполнение тела цикла необязательным при определенных обстоятельствах. Так, в следующем коде мы перемещаем все непрозрачные экземпляры клипов к левому краю рабочего стола и пропускаем тело цикла для прозрачных экземпляров:

```
for (var prop in _root) {
  if (typeof _root[prop] == "movieclip") {
    if (_root[prop]._alpha < 100) {
      continue;
    }
    _root[prop]._x = 0;
  }
}
```

Максимальное количество итераций

Ранее в этой главе отмечалось, что в ActionScript не разрешается бесконечное выполнение циклов. Во Flash Player 5 и более поздних версиях продолжительность выполнения циклов ограничена 15 секундами. Количество итераций, которые могут быть совершены за это время, зависит от содержимого цикла и от быстродействия компьютера. По соображениям надежности не следует создавать циклы, выполнение которых требует более нескольких секунд (для процессора это целая эра!). Большинство циклов должно завершаться за считанные миллисекунды. Если цикл выполняется дольше (например, из-за того, что он обрабатывает сотни строк при инициализации игры с перестановкой слов), имеет смысл переписать код с применением цикла

временной диаграммы, как описано в следующем разделе «Циклы временной диаграммы и событий клипа». Циклы временной диаграммы позволяют обновлять экран в ходе выполнения сценария и избегать возможного вывода сообщения об ошибке, показанного на рис. 8.1.

Рис. 8.1. *Этот цикл никуда не годится! Закрывай!*

Если цикл выполняется во Flash Player 5 или более поздней версии дольше 15 секунд, выводится окно, предупреждающее пользователя о том, что сценарий в фильме задерживает воспроизведение фильма. Пользователю предлагается выбор: ждать завершения сценария или прекратить его выполнение.

Flash Player 4 еще более строг: он допускает только 200 000 итераций, после чего все сценарии выключаются без предупреждения.

Следует проявлять особую осторожность: 15-секундное предупреждение, которое видят пользователи, не сообщает о том, что остановка одного сценария на самом деле вызовет прекращение выполнения всех сценариев в фильме! Если пользователь выберет «Yes», чтобы прекратить выполнение цикла, все сценарии фильма выключатся.

Циклы временной диаграммы и событий клипа

Все циклы, которые мы до сих пор видели, заставляют интерпретатор повторно выполнять блоки кода. Такими же будут большинство ваших циклов. Но иногда желательно также создать *цикл временной диаграммы* (*timeline loop*), зациклив воспроизводящую головку Flash на временной диаграмме. Для этого прикрепите ряд инструкций к любому кадру. К следующему кадру прикрепите функцию *gotoAndPlay( )*, адресом которой является предыдущий кадр. При воспроизведении фильма головка будет циклически перемещаться из одного кадра в другой, вызывая многократное выполнение кода первого кадра.

Простой цикл временной диаграммы можно создать, выполнив следующие шаги:

1. Создайте новый фильм Flash.
2. К кадру 1 прикрепите следующую инструкцию:

```
trace("Привет! Добро пожаловать в кадр 1");
```

3. К кадру 2 добавьте пустой ключевой кадр и прикрепите следующие инструкции:

```
trace("Это кадр 2");
this.gotoAndPlay(1);
```

4. Выберите Control → Test Movie.

При тестировании этого фильма мы видим бесконечный поток текста:

```
Привет! Добро пожаловать в кадр 1
Это кадр 2
Привет! Добро пожаловать в кадр 1
Это кадр 2
```

Циклы временной диаграммы могут делать две вещи, недоступные обычным циклам:

- выполнять блок кода бесконечное количество раз, не вызывая при этом ошибок;
- выполнять блок кода, требующий обновления сцены между итерациями.

Эта вторая особенность циклов временной диаграммы требует некоторых разъяснений. При выполнении сценария любого кадра сцена фильма зрительно не обновляется до окончания сценария. Это означает, что обычные инструкции цикла не могут применяться для выполнения повторяющихся зрительных и звуковых заданий, потому что результаты заданий не выводятся между итерациями цикла. Например, перемещение клипа требует обновления сцены, поэтому с помощью обычного цикла нельзя осуществить программную анимацию клипа.

Можно предположить, что следующий код зрительно перемещает клип `ball_mc` в горизонтальном направлении по сцене:

```
for (var i = 0; i < 50; i++) {
  ball_mc._x += 10;
}
```

Теоретически в этом цикле реализован правильный подход: он многократно изменяет положение шарика с малыми приращениями, что должно создавать впечатление движения. Однако фактически шарик не сдвигается при каждом изменении позиции `_x`, потому что сцена в промежутке между итерациями цикла не обновляется. Вместо этого мы видим, что шарик внезапно прыгает на 500 пикселов вправо – по 10 пикселов на каждую из 50 итераций – после завершения сценария.

Обновлять экран после каждого выполнения инструкции `ball_mc._x += 10;` можно посредством такого цикла временной диаграммы:

```
// КОД В КАДРЕ 1
ball_mc._x += 10;
// КОД В КАДРЕ 2
this.gotoAndPlay(1);
```

Поскольку Flash обновляет экран между каждыми двумя кадрами, шарик будет двигаться. Но цикл временной диаграммы полностью монополизирует временную диаграмму, на которой находится. Во время его выполнения

нельзя анимировать какие-либо другие объекты на временной диаграмме. Более правильный подход состоит в помещении нашего цикла временной диаграммы в пустой клип из двух кадров. Мы сможем использовать обновление сцены между итерациями цикла, не замораживая временную диаграмму, которая может потребоваться для другой анимации.

Создание цикла временной диаграммы пустого клипа

Далее показано, как создать цикл временной диаграммы пустого клипа:

1. Создать новый фильм Flash.
2. Создать символ с именем *ball*, который содержит круг.
3. На главной сцене переименовать слой *Layer 1* в *ball*.
4. В слое *ball* разместить экземпляр символа *ball*.
5. Дать экземпляру *ball* имя `ball_mc`.
6. Выбрать Insert → New Symbol, чтобы создать пустой символ.
7. Дать символу имя *process*.
8. К кадру 1 символа *process* прикрепить следующий код:

```
_root.ball._x += 10;
```

9. К кадру 2 символа *process* добавить пустой ключевой кадр и привязать код:

```
this.gotoAndPlay(1);
```

10. Вернуться в главную временную диаграмму фильма и создать слой с именем *scripts*.
11. Поместить экземпляр символа *process* в слой *scripts*.
12. Назвать экземпляр `processMoveBall_mc`.
13. Выбрать Control → Test Movie.

Теперь экземпляр `processMoveBall_mc` будет перемещать `ball_mc`, не мешая воспроизведению главной временной диаграммы, на которой располагается `ball_mc`.

Обратите внимание, что шаг 12 не является обязательным, но он дает нам больший контроль над циклом. Дав нашему экземпляру *process* имя, мы сможем останавливать и запускать наш цикл, запуская и останавливая воспроизведение экземпляра:

```
processMoveBall_mc.play();
processMoveBall_mc.stop();
```

Заметьте, что `processMoveBall_mc` и `ball_mc` должны существовать на главной временной диаграмме в течение предполагаемого времени выполнения цикла. Для лучшей переносимости можно было создать в символе *process* относительную ссылку на клип `ball_mc`:

```
_parent.ball_mc._x += 10;
```

Циклы временной диаграммы нельзя зацикливать на одном кадре. Это значит, что если поместить функцию *gotoAndPlay(5)* в кадр 5 фильма, она будет проигнорирована. Проигрыватель понимает, что воспроизводящая головка уже находится в кадре 5, и просто ничего не делает.

Файлы *.fla* примеров цикла временной диаграммы можно найти в сетевом хранилище кода (Code Depot).

Циклы событий клипов onEnterFrame()

Циклы временной диаграммы эффективны, но не всегда элегантны. С помощью обработчика событий клипа можно достичь тех же результатов, что и с помощью цикла временной диаграммы, но с большей гибкостью (попробуйте разобрать следующий пример или прочитайте об обработчиках событий клипов в главе 10).

Будучи помещенным в клип, обработчик события *onEnterFrame( )* вызовет выполнение блока кода с частотой смены кадров в фильме. Можно поместить обработчик события *onEnterFrame( )* в пустой клип с одним кадром для многократного выполнения блока кода, позволяя сцене обновляться между каждыми двумя повторами (так, как это делает цикл временной диаграммы). В качестве испытания попробуйте выполнить следующие шаги:

1. Создайте экземпляр `ball_mc`, выполнив шаги 1–7 из предыдущего раздела.
2. На главной временной диаграмме создайте слой с именем *scripts*.
3. К кадру 1 слоя *scripts* прикрепите код:

```
// Этот код требует Flash Player 6 или последующую версию
this.createEmptyMovieClip("processMoveBall_mc", 1);
processMoveBall_mc.onEnterFrame = function () {
  _root.ball_mc._x += 10;
}
```

4. Выберите Control → Test Movie.

Экземпляр `ball_mc` должен двигаться по сцене.

Циклы событий *onEnterFrame( )* освобождают от необходимости вкладывать код внутрь клипа и не требуют кадров на временной диаграмме. Действие такого цикла происходит в одном обработчике события. Для того чтобы прекратить цикл *onEnterFrame( )*, достаточно удалить клип, в котором он находится, например:

```
processMoveBall_mc.removeMovieClip();
```

Либо удалить обработчик *onEnterFrame( )*:

```
delete processMoveBall_mc.onEnterFrame;
```

Если приложение простое, можно попробовать вообще отказаться от пустого клипа для цикла *onEnterFrame( )*. В некоторых случаях вполне допустимо прикрепить цикл событий непосредственно к клипу, с которым идет работа. В нашем примере с `ball_mc` можно не создавать отдельный пустой клип, если

определить обработчик *onEnterFrame( )* непосредственно в экземпляре ball_mc:

```
ball_mc.onEnterFrame = function () {
  this._x += 10;
}
```

Мы даже можем определить цикл события *onEnterFrame( )* на главной временной диаграмме, прикрепив к одному из ее кадров следующий код:

```
this.onEnterFrame = function() {
  this.ball_mc._x += 10
}
```

Обратите внимание, что мы избегаем ссылаться на главную временную диаграмму как на переменную _root, значение которой может измениться, если весь фильм загружается в клип.

Во Flash Player 6 каждый клип (включая главную временную диаграмму) может иметь только один обработчик событий *onEnterFrame( )*. В большинстве приложений один, главный цикл приложения выполняется один раз для каждого кадра, обновляя весь фильм. Например:

```
// КОД НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
function mainLoop () {
  // Выполнить здесь обновления для всего приложения
}
this.onEnterFrame = mainLoop;
```

В более редких случаях несколько клипов обновляют разные аспекты приложения, используя собственные обработчики *onEnterFrame( )*.

Влияние частоты кадров на циклы временной диаграммы и циклы событий клипов onEnterFrame()

Поскольку каждая итерация циклов временной диаграммы и *onEnterFrame( )* приходится на один кадр, частота их выполнения связана со скоростью смены кадров фильма. Если объект перемещается по экрану с помощью цикла временной диаграммы или события клипа, увеличение частоты кадров может увеличить скорость анимации.

Когда мы программировали перемещение клипа ball_mc в наших первых примерах, то неявно задавали скорость шарика относительно частоты кадров. Наш код говорит, что в каждом новом кадре надо переместить шарик на десять пикселов вправо:

```
_ball_mc._x += 10;
```

Поэтому скорость шарика зависит от частоты кадров. Если фильм воспроизводится со скоростью 12 кадров в секунду, то клип ball_mc перемещается со скоростью 120 пикселов в секунду. Если фильм воспроизводится со скоростью 30 кадров в секунду, наш клип ball_mc перемещается со скоростью *300* пикселов в секунду!

При определении временных параметров анимаций, управляемых сценарием, хочется рассчитать расстояние, на которое перемещается объект в соответствии с частотой кадров. Это значит, что если фильм воспроизводится со скоростью 20 кадров в секунду, а мы хотим, чтобы объект двигался со скоростью 100 пикселов в секунду, то можно установить скорость объекта равной 5 пикселам на кадр (5 пикселов × 20 кадров в секунду = 100 пикселов в секунду). В таком подходе есть два серьезных недостатка:

- Поставив скорость объекта в зависимость от скорости воспроизведения, мы столкнемся с трудностями при изменении частоты кадров. Изменив частоту кадров, придется пересчитать скорость и внести соответствующие изменения в код.
- Проигрыватель Flash не обязательно воспроизводит фильм с той частотой кадров, которая была установлена при редактировании фильма. Если компьютер, воспроизводящий фильм, не в состоянии выводить кадры с запланированной частотой, он отображает их медленнее. Замедление зависит от загрузки системы: если выполняются другие программы или Flash выполняет задачу, требующую интенсивных вычислений, то частота кадров может на некоторое время снизиться, а затем восстановиться до нормальной величины.

Вы можете следить за временем и получить фактическую частоту кадров самостоятельно с помощью инструмента, расположенного по адресу:

http://www.moock.org/webdesign/flash/actionscript/fps-speedometer

В некоторых случаях воспроизведение анимации на незначительно отличающихся скоростях может быть допустимо. Но когда важна точность зрительного представления или скорость реакции в игре, значительно лучшим решением будет расчет перемещения объекта в зависимости от истекшего времени, а не от частоты кадров. Пример 8.4 демонстрирует грубый образец анимации, основанной на времени (то есть скорость воспроизведения `ball_mc` не зависит от частоты кадров). В этом новом фильме три кадра и два слоя: один слой с экземпляром `ball_mc`, а другой с нашими сценариями.

Пример 8.4. Расчет величины перемещения в зависимости от времени, а не от частоты кадров

```
// КОД В КАДРЕ 1
var distancePerSecond = 50;  // Перемещение в пикселах в секунду
var now = getTimer();        // Текущее время
var then = 0;                // Время вывода предыдущего кадра
var elapsed;                 // Число миллисекунд между выводом кадров
var numSeconds;              // elapsed в секундах
var moveAmount;              // Величина перемещения

// КОД В КАДРЕ 2
then = now;
now = getTimer();
elapsed = now - then;
numSeconds = elapsed / 1000;
```

```
moveAmount = distancePerSecond * numSeconds;
ball_mc._x += moveAmount;

// КОД В КАДРЕ 3
gotoAndPlay(2);
```

Заметьте, что перемещение с учетом времени может производиться рывками, если частота кадров резко изменится. Сгладить его можно путем усреднения времени между выводом нескольких кадров, а не всего лишь двух.

Альтернатива циклам временной диаграммы: setInterval()

Во Flash Player 6 с помощью *setInterval( )* можно выполнять некоторую функцию каждые *n* миллисекунд. Например, в следующем коде создается функция *moveBall( )*, которая перемещает клип `ball_mc` на 10 пикселов вправо, а затем использует *setInterval( )* для выполнения этой функции каждые 20 миллисекунд.

```
moveBall = function () {
  _root.ball_mc._x += 10;
}
ballMoverID = setInterval(moveBall, 20);
```

Чтобы прекратить периодическое выполнение функции, позже можно выполнить такой вызов:

```
clearInterval(ballMoverID);
```

Подобно циклу временной диаграммы или *onEnterFrame( )*, функция *setInterval( )* может выполнять блок кода до бесконечности и допускает обновление экрана между итерациями. Однако функция *setInterval( )* не привязана непосредственно к частоте кадров фильма (хотя зависит от нее). Подробности представлены в описании *setInterval( )* в «Справочнике по языку».

Вперед!

Мы забрались довольно далеко. Конец этой главы знаменует определенную веху: изучение инструкций ActionScript закончено. Это означает, что в нашем распоряжении есть переменные, данные, типы данных, выражения, операторы и инструкции. Эти элементы языка лежат в основе всех сценариев. Если вы прочли и поняли все до этого места, хотя бы в основном, можете официально заявить, что умеете «говорить» на ActionScript.

До конца части I мы будем работать над тем, чтобы наш разговор стал более красноречивым, а наши команды – более мощными. Мы рассмотрим более сложные темы, касающиеся переносимости кода, создания событий, инициирующих выполнение кода, организации сложных данных и программного управления клипами. Соответствующие технологии помогут нам создать более совершенные прикладные примеры.

9

Функции

У меня почти кружится голова – я собираюсь рассказать вам о функциях, а ведь это чрезвычайно мощная часть ActionScript. Функция представляет собой всего лишь фрагмент кода, который можно многократно использовать в программе. Функции не только придают сценариям чрезвычайную гибкость и делают их удобными, но и предоставляют возможность управлять элементами фильма Flash. Трудно представить себе программирование без функций: они облегчают выполнение стандартных операций – от сортировки слов до вычисления расстояния между двумя клипами. В этой главе мы познакомимся с функциями, а в главе 12 узнаем, как с их помощью создавать сложные, мощные программы.

Сначала сосредоточимся на функциях, с которыми проще всего разобраться, – пользовательских *функциях*, создаваемых нами в своих сценариях. Эти функции отличаются от *встроенных функций*, предоставляемых ActionScript.

В спецификации ECMA-262 пользовательские функции (custom functions) называются *программными функциями* (*program functions*), а встроенные функции (built-in functions) – *внутренними функциями* (*internal functions*). В обоих случаях мы употребляем принятые в обиходе названия вместо формальных терминов.

Научившись создавать собственные функции, мы освоим следующие базовые понятия:

Объявление (*определение*) *функции*

Создание функций, используемых в сценариях.

Вызов функций

Действие, запускающее выполнение функции. Иными словами, выполнение кода, содержащегося в функции.

Аргументы и параметры функций

Предоставление вызываемым функциям данных для обработки.

Завершение функций

Окончание выполнения функции с возможным возвратом результата.

Область видимости функции

Определение доступности и срока жизни функции, а также доступности переменных, содержащихся в теле функции.

Итак, приступим!

Создание функций

Чтобы создать простую *функцию*, требуется задать имя функции и блок инструкций, которые должны быть выполнены, например:

```
function funcName (param1, param2,...paramn) {
  statements
}
```

Ключевое слово `function` начинает *объявление* новой функции. Затем следует имя функции, *funcName*, по которому позднее мы будем *вызывать* нашу функцию. Указанное имя *funcName* должно быть допустимым идентификатором[1], но в целом может быть любым выбранным нами уникальным именем (не создавайте функции с такими же именами, как у встроенных функций). Затем ставится пара круглых скобок, `()`, в которых размещаются параметры (*param1*, *param2* и т. д.), разделяемые запятыми. О параметрах мы расскажем позже. Если у функции нет параметров, оставим скобки пустыми. Наконец, мы размещаем *тело функции* (то есть блок инструкций), который содержит код, выполняемый, когда наша функция *вызывается*.

Напишем очень простую функцию:

1. Создадим новый пустой фильм Flash.
2. К кадру 1 уровня 1 главной временной диаграммы прикрепим следующий код:

```
function sayHi () {
  trace("Hi there!");
}
```

Это было нетрудно: мы всего лишь создали простую функцию с именем *sayHi( )*, не требующую параметров. При ее запуске будет выполнена инструкция *trace( )*, находящаяся в теле функции. Не закрывайте созданный вами фильм: сейчас мы научимся запускать нашу функцию.

Запуск функций

Мы запускаем функцию, как волшебник произносит заклинание: произнося ее имя, за которым следует оператор вызова функции `()`, с которым мы познакомились в главе 5:

```
funcName(param1, param2, ...paramn)
```

[1] В главе 15 рассматриваются правила, регламентирующие выбор допустимых идентификаторов.

(Можно почти различить слабый запах тлеющего в воздухе корня мандрагоры.)

Скобки содержат значения параметров, требуемых функцией. Если функция не требует параметров (согласно ее объявлению), скобки остаются пустыми. Попробуем вызвать нашу функцию *sayHi( )*, у которой нет параметров.

Еще раз напомним объявление нашей функции *sayHi( )*:

```
function sayHi () {
  trace("Hi there!");
}
```

А вот вызов нашей функции *sayHi( )*:

```
sayHi();
```

В этом простом примере мы поместили вызов функции в тот же сценарий, где находится ее объявление, но, как мы потом увидим, возможны и другие варианты. При вызове функции *sayHi( )* выполняется тело этой функции, в результате чего в окне Output появляется строка «Hi there!».

Как видите, удобнее набрать на клавиатуре *sayHi( );*, чем полностью вводить инструкцию *trace( )*, а имя функции *sayHi* представляет собой более содержательное описание действий нашего кода. Продуманные имена функций облегчают чтение кода, – его строки становятся похожи на предложения естественного языка. В данной книге и в документации по программированию имена функций указываются вместе со скобками вызова, например «функция *sayHi( )*», а не «функция *sayHi*». Благодаря этому в описаниях легче обнаружить имя функции.

Прежде чем продолжить, обратим внимание на точку с запятой в конце вызова функции:

```
sayHi();
```

Точка с запятой добавляется, поскольку вызов функции составляет законченную инструкцию, а правила хорошего тона требуют, чтобы все инструкции заканчивались точкой с запятой.

Можете не поверить, но мы только что изучили основы создания и вызова функций. Это уже немало. Функции позволяют централизовать код и облегчают его сопровождение, особенно если одно и то же действие приходится многократно выполнять на протяжении программы. Мощь функций увеличивается, если использовать их с параметрами, что мы и обсудим в следующем разделе.

Передача информации в функции

В предыдущем разделе мы создали функцию, выполнявшую простую инструкцию *trace( )* – не самый убедительный представитель рода функций. Вот более интересная функция, которая перемещает экземпляр клипа с именем `ball` на небольшое расстояние. Чтобы создать ее, выполните следующие действия:

1. Создайте новый пустой фильм Flash.
2. Создайте новый символ с именем *ballSymbol*, который содержит круг.
3. Поместите экземпляр *ballSymbol* на уровень 1.
4. Дайте экземпляру *ballSymbol* имя `ball`.
5. К кадру 1 уровня 1 главной временной диаграммы прикрепим следующий код:

```
// Сохраним переносимую ссылку на главную временную диаграмму.
_global.mainTimeline = this;

function moveBall () {
  mainTimeline.ball._x += 10;
  mainTimeline.ball._y += 10;
}
```

Не рекомендуется обращаться из функций к внешним переменным временной диаграммы (таким, как `ball`) непосредственно, поэтому мы обращаемся к `ball` через глобальную ссылку приложения на главную временную диаграмму фильма. Определив функцию *moveBall( )*, мы можем перемещать с ее помощью клип `ball` по диагонали. Добавьте в код кадра 1 вызов этой функции:

```
moveBall();
```

Шарик перемещается по диагонали вправо вниз. (Заметьте, что исходное положение (0, 0) – в левом верхнем углу сцены. Если значение `_x` увеличивается, шарик перемещается вправо, но если увеличивается значение `_y`, то шарик перемещается *вниз*, а не вверх, – совсем не так, как в декартовых координатах.)

Наша функция *moveBall( )* удобна, но ей не хватает гибкости. Она работает только с одним клипом шарика (`ball`), перемещает `ball` только в одном направлении и всегда на одно и то же расстояние.

Правильно спроектированная функция должна работать при различных обстоятельствах. Можно обобщить функцию *moveBall( )* так, чтобы она перемещала *любой* клип на *любое* расстояние в *любом* направлении. Первый шаг в обобщении функции состоит в том, что определяются факторы, управляющие ее поведением. В нашей функции *moveBall( )* факторами являются имя перемещаемого клипа, расстояние, на которое он перемещается по горизонтали, и расстояние, на которое он перемещается по вертикали. Такие факторы называют *параметрами* функции: это информация, которую мы можем изменять при каждом вызове функции.

Создание функций с параметрами

Вспомните общий синтаксис объявления простой функции:

```
function funcName (param1,  param2, . . . paramn) {
  statements
}
```

Параметры функции – это переменные, которые можно использовать в функции, но их значения определяются во время ее выполнения. Мы задаем параметры в объявлении функции в виде списка идентификаторов, разделенных запятыми и заключенных в круглые скобки, как показано в предыдущем фрагменте кода.

Если определены параметры, можно обращаться к ним из тела функции, как к любым другим переменным. Вот пример функции с одним параметром:

```
function say (msg) {                  // Определить параметр msg
  trace("The message is " + msg);     // Использовать msg в инструкции trace()
}
```

В нашем определении функции указан параметр `msg`. Инструкция *trace( )* использует этот параметр, как любую другую переменную, выводя ее значение в окно Output.

Зададим параметры нашей функции, чтобы при каждом ее выполнении можно было устанавливать имя клипа, расстояние по горизонтали и расстояние по вертикали. Соответствующий модифицированный код показан в примере 9.1.

Пример 9.1. Обобщенная функция moveClip

```
function moveClip (theClip, xDist, yDist) {
  theClip._x += xDist;
  theClip._y += yDist;
}
```

Мы переименовали функцию *moveBall( )*, дав ей общее имя *moveClip( )*, которое отражает ее способность работать с любым клипом. Кроме того, мы определили три параметра: `theClip` (клип, который требуется переместить), `xDist` (величина перемещения по горизонтали) и `yDist` (величина перемещения по вертикали). Внутри тела функции мы используем эти параметры вместо «жестко закодированных» литералов в прежнем варианте *moveBall( )*, позволяя тем самым перемещать любой клип на любое расстояние по горизонтали и вертикали. Обратите внимание, что глобальная переменная `mainTimeline` нам больше не нужна, так как при вызове функции ей сообщается о том, какой клип надо переместить. В следующем разделе мы расскажем о вызове *moveClip( )* и задании значений ее параметров.

Вызов функций с параметрами

Создавая функцию, мы задаем имена параметров, которые, по сути, замещают значения данных. При вызове функции мы передаем значения, подставляемые вместо каждого из параметров.

Термины *параметры* и *аргументы* употребляются в обычной речи и в большей части документации взаимозаменяемым образом. Формально *аргументы* представляют собой значения, передаваемые при вызове функции, а *параметры* – это заместители в функции, которые «получают» аргументы.

Вспомните общий синтаксис вызова функции без параметров:

```
funcName()
```

Чтобы *передать* функции аргументы, при вызове функции мы заключаем список значений в круглые скобки:

```
funcName(arg1, arg2, arg3,...argn)
```

Значения, указываемые в качестве аргументов, могут быть любыми допустимыми выражениями ActionScript, в том числе составными выражениями. Например, ранее мы определили простую функцию *say( )*, которая ждет единственный параметр `msg`:

```
function say(msg) {
  trace("The message is " + msg);
}
```

Функцию *say( )* можно вызвать посредством такой инструкции:

```
say("This is my first argument...how touching");
```

или такой:

```
say(99);
```

или даже так:

```
now = new Date();
say(now)
```

Обратите внимание, что значения «This is my first argument... how touching», 99 и текущая дата принадлежат различным типам данных. ActionScript разрешает передавать в функцию любые данные любого типа, если только функция знает, что делать с переданным значением. (Языки типа C и Java требуют предварительно объявить тип данных для каждого параметра и выводят сообщение об ошибке пре передаче данных неверного типа.)

До передачи любого аргумента его значение полностью вычисляется. Поэтому можно вызывать функции со сложными выражениями. Очень часто в качестве аргументов функции выступают переменные или выражения, содержащие переменные, например:

```
var name = "Gula";
say("Welcome to my web site " + name);
```

Выражение "*Welcome to my web site* " + *name* вычисляется до передачи функции *say( )*, поэтому функция получает значение «Welcome to my web site Gula». Это означает, что можно генерировать аргументы функции программно: аргументы, указываемые при ее вызове, будут различаться в зависимости от обстоятельств на этапе выполнения, таких как имя пользователя или дата. Мощный ресурс.

Чтобы передать в функцию более одного аргумента, мы разделяем их запятыми. Вспомните, что функция *moveClip( )* из примера 9.1 принимает три параметра: `theClip`, `xDist`, `yDist`. Поэтому вызов *moveClip( )* может выглядеть так:

```
moveClip(ball, 5, -15);
```

Каждый из трех аргументов присваивается как значение соответствующему параметру, указанному в объявлении функции: `ball` присваивается параметру `theClip`, 5 – параметру `xDist`, а –15 – параметру `yDist`. Соответствие между параметрами и аргументами основывается исключительно на порядке их перечисления через запятую в списках: первый аргумент в вызове функции соответствует первому параметру в объявлении функции, второй аргумент – второму параметру и т. д. Содержимое аргументов, указанных в вызове функции, передается в тело функции, где обращение к ним происходит по именам параметров, заданным в объявлении функции. Это означает, что хотя при вызове функции *moveClip( )* в качестве аргумента выступал идентификатор `ball`, экземпляр клипа `ball` будет известен в теле функции по имени параметра `theClip`.

Аргументы, заданные при вызове функции, могут быть любыми допустимыми выражениями, в том числе сложными выражениями, литералами или идентификаторами. Параметры, указываемые в объявлении функции и используемые внутри ее тела, всегда являются идентификаторами (то есть именами переменных, локальных для этой функции).

Любой клип можно переместить на любое расстояние путем вызова общей функции *movieClip( )* с различными аргументами. Например, переместим экземпляр клипа с именем `square` на 2 пиксела вправо и на 100 пикселов вниз посредством следующего кода:

```
moveClip(square, 2, 100);
```

Обратите внимание, что для перемещения клипа по горизонтали, но не по вертикали, можно присвоить *yDist* значение 0:

```
moveClip(square, 2, 0);
```

Наоборот, можно переместить этот клип по вертикали, но не по горизонтали, присвоив *xDist* значение 0:

```
moveClip(square, 0, 100);
```

Поэтому можно не создавать три отдельные функции, а ограничиться одной функцией и перемещать клип по горизонтали, по вертикали или в обоих направлениях, передавая различные аргументы при вызове функции. Избегая создания лишних функций, мы делаем код компактным и облегчаем его сопровождение.

Чтобы передать функции значение только для одного параметра, даже если она ожидает несколько параметров, вместо пустых параметров надо указать `undefined`, например:

```
function showName (first, middle, last) {
  trace("The name is " + first + " " + middle + " " + last);
}
showName(undefined, undefined, "moock");
```

Конечно, такой подход хорош только тогда, когда функция допускает передачу пустых аргументов. Создавая функцию, которая должна получить значение для конкретного параметра, обеспечьте в ней проверку этого значения. Например, переработаем функцию *showName( )*, чтобы она требовала присвоения значения каждому из своих параметров:

```
function showName (first, middle, last) {
  // Если параметры недопустимы...
  if (first === undefined || middle === undefined || last === undefined) {
    // Вывести сообщение об ошибке и завершить работу.
    trace("INVALID ARGUMENT VALUES SPECIFIED");
    return;
  }
  trace("The name is " + first + " " + middle + " " + last);
}
```

Выход из функций и возвращение значений

В отсутствие иных указаний функция завершает работу естественным путем, когда интерпретатор завершает выполнение последней инструкции в теле функции. Можно, однако, завершить функцию, не доходя до последней инструкции. Кроме того, функция может возвращать результат (передавать вычисленное значение) в код, который ее вызвал. Посмотрим, как это делается.

Завершение выполнения функции

Инструкцию *return*, с которой мы познакомились в главе 6, можно использовать для завершения выполнения функции и возвращения результата. Когда интерпретатор во время выполнения функции сталкивается с инструкцией *return*, он пропускает все оставшиеся в функции инструкции. Рассмотрим такой пример:

```
function say(msg) {
  return;
  trace(msg);         // Эта строка никогда не достигается
}
```

Данный пример нереалистичен, потому что инструкция *return* всегда вызывает завершение функции раньше, чем будет достигнута инструкция *trace( )*. Поэтому инструкция *return* обычно является последней в функции, если только она не применяется внутри условной инструкции. В следующем примере *return* обеспечивает выход из функции, если введенный пароль неверен:

```
var correctPass = "cactus";
function enterSite(pass) {
  if (pass != correctPass) {
    // Выйти, если пароль неверен
    return;
  }
  // Этот код выполняется только при верном пароле
  gotoAndPlay("intro");
```

```
}
enterSite("hackAttack");  // Выход из функции будет досрочным
enterSite("cactus");      // Выход из функции будет естественным
```

Как предполагается самим именем, *return* приказывает интерпретатору вернуть управление в сценарий, откуда вызвана функция. Если инструкция *return* отсутствует, ActionScript действует так, как если бы последняя строка функции содержала инструкцию *return*:

```
function say(msg) {
  trace(msg);
  return;             // В таком контексте инструкция return не обязательна
}
```

Независимо от того, является инструкция *return* подразумеваемой или явной, по окончании работы функции выполнение возобновляется со строчки кода, следующей за вызовом функции. Например:

```
say ("Something");  // Выполняется код функции say()
// По завершении функции say() выполнение возобновляется здесь
trace ("Something else");
```

Возвращение значений функциями

Как мы видели, инструкция *return* всегда завершает функцию. Но она может также вернуть значение в сценарий, вызвавший функцию:

```
return expression;
```

Значение *expression* становится результатом вызова функции, например:

```
// Определим функцию, складывающую два числа
function combine(a, b) {
  return a + b;  // Возвратить сумму двух аргументов
}
// Вызвать функцию
var total = combine(2, 1);  // присваивает total значение 3
```

Выражение или результат, возвращаемое инструкцией *return*, называется *возвращаемым значением* функции.

Обратите внимание, что наша функция *combine( )* просто вычисляет и возвращает сумму двух чисел (а также конкатенирует две строки). Она не совершает явного действия, как *sayHi( )*, которая выводила сообщение, или *moveClip( )*, которая перемещала клип. Возвращаемое функцией значение можно присвоить переменной:

```
var total = combine (5, 6);                // Устанавливает total в 11
var greet = combine ("Hello ", "Cheryl")   // устанавливает переменную greet
                                           // равной "Hello Cheryl"
```

Результат вызова функции представляет собой обычное выражение. Поэтому его можно использовать в других выражениях. В следующем примере переменной phrase присваивается значение «11 people were at the party»:

```
var phrase = combine(5, 6) + " people were at the party";
```

Мы будем часто использовать значения, возвращаемые функциями, в сложных выражениях и даже как аргументы при вызове других функций. Например:

```
var a = 3;
var b = 4;
function sqr(x) {  // Возводит число в квадрат
  return x * x;
}
var hypotenuse = Math.sqrt(sqr(a) + sqr(b));
```

Обратите внимание, как в этом примере значения, возвращаемые функцией *sqr( )*, передаются функции *Math.sqrt( )*. По этому образцу наш более ранний пример можно переписать так:

```
var phrase = combine (combine(5,6), " people were at the party");
```

В этом примере внутреннее выражение *combine(5,6)*, значением которого является 11, становится аргументом другого вызова функции *combine( )*, где оно конкатенируется со строкой «people were at the party».

Если в инструкции *return* нет выражения, которое должно возвращаться, или инструкция *return* опущена целиком, то функция возвращает значение `undefined`. На практике это часто вызывает ошибки.

Например, следующий код не делает ничего полезного, поскольку отсутствует инструкция *return*:

```
function combine(a, b) {
  var result = a + b;  // Результат вычисляется, но не возвращается
}
```

Аналогично неверен и такой код:

```
function combine(a, b) {
  var result = a + b;
  return;  // Вы забыли указать возвращаемое значение
}
```

Корректная функция должна выглядеть так:

```
function combine (a, b) {
  var result = a + b;
  return result;
}
```

При создании функции, которая должна возвращать результат вычислений, не забудьте поместить в нее инструкцию *return*, действительно возвращающую нужное значение. В противном случае возвращаемым значением будет `undefined`, а все последующие расчеты, основанные на этом результате, почти наверняка окажутся некорректными.

Литералы функций

ActionScript позволяет создавать *литералы функций*, очень удобные, если функция нужна лишь временно или хотелось бы использовать функцию там, где ожидается выражение.

Литералы функций имеют такой же синтаксис, как обычные объявления функций, но опускается имя функции, а после блока инструкций ставится точка с запятой. Общий формат таков:

```
identifier = function (param1, param2, ... paramn) { statements };
```

где *param1, param2, ... paramn* – необязательный список параметров, а *statements* – одна или более инструкций, образующих тело функции. Имя функции отсутствует, поэтому определение литерала функции «теряется», если не сохранить его в переменной (или в элементе массива, или в свойстве объекта). Сохранить литерал функции в переменной, чтобы использовать его в дальнейшем, можно так:

```
// Сохранить литерал функции в переменной
var mouseCoords = function () { return [ _xmouse, _ymouse ]; };
// Теперь можно выполнить функцию
mouseCoords();
```

Обратите внимание, что поскольку ActionScript не поддерживает конструктор *Function( )*, как в JavaScript, динамические функции нельзя составлять на этапе исполнения, как это можно делать в JavaScript.

Как мы увидим в главе 12, литералы функций часто применяются для присваивания методов классам или объектам. Например, код

```
Ball.prototype.startMoving = function () {
  // Здесь код, выполняющий перемещение
};
```

равносилен следующему:

```
Ball.prototype.startMoving = startMoving;
function startMoving () {
  // Здесь код, выполняющий перемещение
}
```

Однако между литералами функций и обычными объявлениями функций есть три тонких различия:

- В случае присваивания в качестве метода литерал функции не оставляет внешней ссылки на функцию, определенной в области видимости объявления функции. Например, в первом примере *Ball* (предполагая, что код *Ball* прикреплен к главной временной диаграмме) литерал функции записывается в `Ball.prototype.startMoving`, а в `_root` не остается отдельной ссылки на функцию. Однако во втором примере ссылка на *startMoving( )* существует как в `Ball.prototype.startMoving`, так и в `_root`.
- Обычное объявление *function* не завершается, в отличие от литерала функции, точкой с запятой.

- Функции, создаваемые инструкциями *function*, доступны во всем сценарии, даже в инструкциях, предшествующих объявлению функции (в том же ключевом кадре, или в том же *onClipEvent( )*, или в обработчике события *on( )*). Литералы функций доступны только в тех операторах, которые располагаются в сценарии после литерала.

Доступность и срок жизни функций

Как и переменные, программные функции живут не вечно, и к ним нельзя непосредственно обращаться из любого места фильма. Чтобы с уверенностью вызывать функции, необходимо знать, как обращаться к ним из различных мест фильма, и перед вызовом убедиться в их существовании. В ActionScript вызов несуществующей функции не влечет никаких последствий. Когда вызов не выполняется, то не выдается никаких сообщений и не генерируется ошибка (в JavaScript вызов несуществующей функции генерирует ошибку). Чтобы проверить, выполнилась ли вызванная вами функция, поместите в нее инструкцию *trace( )*, как в следующем примере:

```
trace("Function someName executed.");
```

Доступность функций

Пользовательские функции (те, которые мы создаем в коде) *непосредственно* доступны для вызова только из:

- кода, прикрепленного к временной диаграмме клипа, в котором находится объявление функции;
- кода обработчика *onClipEvent( )* в стиле Flash 5 в клипе, в котором находится объявление функции;
- кода обработчика события *on( )* в стиле Flash 5 на кнопке, расположенной на временной диаграмме клипа, в котором находится объявление функции.

Говоря о «непосредственной доступности», мы имеем в виду, что функцию можно вызвать просто по имени, без ссылки на клип или объект, например:

```
myFunction();
```

Доступ к функциям можно получить *косвенно* (то есть удаленным образом) из любой точки фильма посредством синтаксиса с точкой. Как и при удаленной ссылке на переменные, необходимо задать путь к клипу, содержащему функцию, например:

```
myClip.myOtherClip.myFunction();
_parent.myFunction();
_root.myFunction();
```

Итак, предположим, что экземпляр клипа с именем `rectangle` на главной временной диаграмме содержит функцию с именем *area( )*. Можно вызвать функцию *area( )* из любого места в фильме, указав абсолютный путь к `rectangle`, например:

```
_root.rectangle.area();
```

Для обращения к функции из удаленной временной диаграммы клипа действуют те же правила, что и для ссылки на удаленные переменные, описанные в главе 2. (Подробнее о вызове функций из других клипов мы расскажем в главе 13.)

Не все функции прикрепляются к временным диаграммам клипов. Некоторые программные функции могут прикрепляться к объектам, которые встроены или определены пользователем. Если функция прикреплена к объекту, она доступна только через свой объект-хозяин. О доступности и сроке жизни *методов* (то есть функций, прикрепленных к объектам) мы расскажем в главе 12.

Для того чтобы создать функцию, непосредственно доступную в любом месте фильма, определите ее как литерал функции в объекте `_global`, например:

```
_global.functionName = function () {
  statements
}
```

Поддержка глобальных функций и переменных отсутствовала в версиях, предшествовавших Flash Player 6. Дополнительные сведения о глобальных объектах представлены в описании `_global` в «Справочнике по языку».

Как решить, где следует определять функции? Если планируется использовать функцию только для отдельного клипа, определите ее в этом клипе. Если функция будет использоваться несколькими клипами, определите ее на главной временной диаграмме как глобальную. Мы обсудим организацию кода при изучении объектно-ориентированного программирования в главе 12.

Срок жизни функций

Функция, определенная на временной диаграмме клипа, теряется, когда этот клип удаляется со сцены. Попытка вызвать функцию, определенную в несуществующем более клипе, окажется неудачной, но не вызовет сообщения об ошибке. Во Flash 5 лучший способ обеспечить постоянное присутствие функции состоит в том, чтобы определить ее на главной временной диаграмме клипа, потому что главная временная диаграмма сохраняется на протяжении всего фильма. Во Flash 6 самое надежное место для функции – объект `_global`, о чем говорилось в предыдущем разделе.

Обратите внимание, что функции в сценарии кадра инициализируются и становятся доступными, когда воспроизводящая головка впервые входит в кадр, содержащий сценарий. Поэтому можно вызвать функцию до ее объявления, если и вызов, и объявление находятся в сценарии одного и того же ключевого кадра. Например:

```
// Вызов функции tellTime() перед ее объявлением
tellTime();
```

```
// Объявление функции tellTime()
function tellTime() {
  var now = new Date();
  trace(now);
}
```

Этот прием действует внутри сценария одного ключевого кадра, но не между различными слоями временной диаграммы. Код в различных слоях выполняется независимо в соответствии с порядком выполнения временной диаграммы, о чем рассказывается в главе 13.

Поскольку литералы функций недоступны до момента их объявления даже в сценарии, в котором они определены, необходимо определять литералы функций перед их вызовом. Лучше всего определять функции в начале сценариев. Это гарантирует их доступность во всем сценарии даже при объявлении в виде литералов функций.

Область видимости функции

Как указывалось в разделе «Область видимости переменной» главы 2, у всех переменных ActionScript есть определенная *область видимости* (*scope*), или область программы, в которой они действительны. Местонахождение инструкции в фильме определяет:

- Область, где интерпретатор пытается найти переменные, ссылки на которые есть в инструкции.
- Область видимости переменных, создаваемых инструкцией.

Вот, например, инструкция, которая создает переменную `score`:

```
var score = 10;
```

Если эта инструкция прикреплена к клипу `clipA`, то интерпретатор присвоит значение переменной `score`, находящейся в `clipA`. Если эта инструкция прикреплена к клипу `clipB`, то интерпретатор присвоит значение переменной `score`, находящейся в `clipB`.

Инструкции в теле функции действуют в своей собственной отдельной области видимости, называемой локальной *областью видимости*. Локальная область видимости ограничивается этой функцией и отделена от области видимости клипа или объекта, к которым прикреплена функция. Новая локальная область видимости функции создается при каждом ее вызове и уничтожается по завершении выполнения функции. При нахождении переменных, на которые ссылаются инструкции в теле функции, интерпретатор ищет их сначала в локальной области видимости функции.

Например, параметры функции определяются в ее *локальной* области видимости, а не на временной диаграмме, содержащей функцию. Поэтому параметры доступны только инструкциям в теле функции, и только когда функция выполняется. Инструкции вне функции не имеют доступа к параметрам функции.

Локальная область видимости функции предоставляет пространство для временных «локальных» переменных, используемых исключительно внутри функции. Этим предотвращаются возможные конфликты имен между локальными переменными функции и переменными временной диаграммы, а также сокращается общий расход памяти.

Функции и цепочка областей видимости

Несмотря на то что функции действуют в собственных локальных областях видимости, обычные переменные временной диаграммы сохраняют свою доступность для инструкций в теле функции. Локальная область видимости функции – это *первое* место, куда смотрит интерпретатор при разрешении ссылок на переменные, но если ссылку на переменную нельзя обнаружить в локальной области видимости, поиск расширяется на клип, в котором объявлена функция, а затем (во Flash Player 6) на объект `_global`.

Допустим, например, что мы определили переменную `firstName` на временной диаграмме клипа `clipA`. Определим также функцию *getName( )* в клипе `clipA`. Внутри *getName( )* мы ссылаемся на переменную `firstName` в инструкции *trace( )*:

```
firstName = "Christine";
function getName () {
  trace(firstName);
}
getName();
```

Когда мы вызываем *getName( )*, интерпретатор должен найти значение `firstName`. Параметра или локальной переменной с именем `firstName` в локальной области видимости *getName( )* нет, поэтому интерпретатор станет искать переменную `firstName` в `clipA`. Там интерпретатор найдет `firstName` и выведет ее значение, «Christine».

Наша инструкция *trace( )* может ссылаться на переменную временной диаграммы из тела функции *getName( )*, потому что в самой *getName( )* не определен параметр или локальная переменная с именем `firstName`. Теперь посмотрим, что произойдет, если добавить параметр с именем `firstName` в нашу функцию *getName( )*:

```
firstName = "Christine";
function getName (firstName) {
  trace(firstName);
}
getName("Kathy");
```

На этот раз при вызове *getName( )* мы передаем «Kathy» в качестве первого и единственного аргумента, и интерпретатор присваивает значение «Kathy» параметру `firstName`. Когда интерпретатор выполняет инструкцию *trace( )*, он осуществляет поиск в локальной области видимости и находит там параметр `firstName` со значением «Kathy». Поэтому функция выводит на этот раз «Kathy», а не «Christine».

Несмотря на то что существует переменная временной диаграммы `firstName`, предпочтение отдается параметру с именем `firstName`. Этот параметр *преобладает* (*overrides*) над переменной временной диаграммы, или *скрывает* (*occludes*) ее.

Этот пример иллюстрирует действие *цепочки областей видимости* (*scope chain*) – иерархии объектов, используемых интерпретатором для разрешения ссылок на переменные. Для функций, прикрепленных к временным диаграммам, цепочка областей видимости имеет только три уровня: локальная область видимости функции, область видимости клипа, содержащего функцию, и глобальная область видимости (во Flash 5 глобальной области видимости нет). Но когда мы делаем функцию вложенной, цепочка областей видимости может иметь значительно больше уровней, как мы увидим позже.

Доступ к переменным и свойствам, не являющимся локальными для функции, осуществляется посредством синтаксиса с точкой за счет создания явной ссылки (называемой также *квалифицированной*). Например:

```
function dynamicGoto() {
  // Намеренный выход за пределы локальной области видимости
  _root.myClip.gotoAndPlay(_root.myClip.targetFrame);
}
```

Обратите внимание, что цепочка областей видимости функции определяется фиксированно на этапе компиляции относительно инструкции, содержащей *объявление* функции, а не инструкции, содержащей ее *вызов*. То есть областью видимости кода в теле функции является клип, в котором находится объявление функции, а не клип, содержащий инструкцию с вызовом функции. Область видимости такого типа называется лексической или *статической* (в отличие от *динамической области видимости*, которая может изменяться на этапе исполнения). Мы вернемся к теме лексической области видимости при изучении вложенных функций.

Вот пример ошибочного использования цепочки областей видимости:

```
// КОД В ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ ФИЛЬМА
// Цепочка областей видимости этой функции содержит главный фильм
function rotate(degrees) {
  _rotation += degrees;
}
```

Если попробовать повернуть `clipA` с помощью `_root.rotate`, то будет повернут весь главный фильм, а не только `clipA`:

```
// КОД ВО ВРЕМЕННОЙ ДИАГРАММЕ clipA
_root.rotate(30);  // Ой! Весь фильм поворачивается!
```

Эту ситуацию можно исправить, передав клип, подлежащий повороту, в качестве аргумента функции *rotate( )*:

```
function rotate (theClip, degrees) {
  theClip._rotation += degrees;
```

```
}
// Из временной диаграммы _root вызвать rotate() со ссылкой на clipA.
rotate(clipA, 30);

// Либо из временной диаграммы clipA вызвать rotate() со ссылкой на clipA.
_root.rotate(_root.clipA, 30);
```

Возможная альтернатива – определить функцию *rotate( )* как метод класса *MovieClip*, чтобы из нее можно было с помощью ключевого слова `this` обратиться к клипу, через который функция была вызвана:

```
MovieClip.prototype.rotate = function (degrees) {
  this._rotation += degrees;
}

// КОД ВО ВРЕМЕННОЙ ДИАГРАММЕ clipA
this.rotate(30);
```

Функции, оперирующие клипами, лучше всего определять в виде методов класса *MovieClip*, как описывается в разделе «Добавление пользовательских методов и свойств во встроенные классы и объекты» главы 12.

Из-за ошибки во Flash Player 5 литералы функций, назначаемые в качестве методов клипов, не имели лексической области видимости. Например:

```
// Создать переменную на текущей временной диаграмме.
var x = 10;
// Создать переменную на временной диаграмме theClip.
theClip.x = 20;
// Назначить литерал функции методом theClip.
theClip.traceX = function () {
  // Flash Player 5 ошибочно выводит значение x в theClip: 20
  // Flash Player 6 правильно выводит значение x на временной диаграмме,
  // содержащей объявление функции: 10
  trace("x is: " + x);
}
// Вызвать метод traceX().
theClip.traceX();
```

Для обеспечения обратной совместимости Flash Player 6 правильно устанавливает области видимости только при воспроизведении файлов *.swf* в формате Flash 6. Поведение файлов *.swf* формата Flash 5 не меняется при воспроизведении их во Flash Player 6.

Локальные переменные

Переменные, находящиеся в локальной области видимости функции, называются *локальными переменными*. Локальные переменные, в том числе параметры, доступны только инструкциям в теле функции, в которой они определены, и существуют только во время выполнения этой функции. Для создания локальной переменной внутри любой функции (помимо парамет-

ров, которые автоматически становятся локальными переменными) применяется инструкция *var*, например:

```
function funcName() {
  var temp = "just testing!";  // Объявляет локальную переменную temp
}
```

Локальные переменные полезны для временного хранения информации. В следующем примере мы сохраняем промежуточный результат в локальной переменной `lastSpacePlusOne`. Как и все локальные переменные, она уничтожается по окончании выполнения функции:

```
function getLastWord(text) {
  var lastSpacePlusOne = text.lastIndexOf(" ") + 1;              // Локальная
  var lastWord = text.subString(lastSpacePlusOne, text.length);  // Локальная
  return lastWord;
}
// Выводит: "word"
trace(getLastWord("Tell me the last word"));
// Выводит: undefined. lastSpacePlusOne локальна и недоступна
// вне функции getLastWord().
trace(lastSpacePlusOne);
```

Когда в конце функции истекает срок жизни локальных переменных, связанная с ними память помечается для автоматического освобождения. Хранение всех временных значений в локальных переменных позволяет сберечь память. Кроме того, при объявлении локальной переменной не надо беспокоиться о том, что она может вызвать конфликт с одноименной переменной временной диаграммы.

Конечно, не все переменные, используемые в функциях, должны быть локальными. Ранее мы видели, что внутри функции можно создавать, читать и присваивать значения переменным временной диаграммы. Но можно также создавать их и осуществлять в них запись. Внутри функции присваивания переменным, которых нет в локальной области видимости, выполняются в области видимости временной диаграммы, содержащей объявление функции. В следующем примере `x` представляет собой локальную переменную, но `y` и `z` – это переменные временной диаграммы:

```
var z = 1;

function createVars() {
  var x = 10;    // Создать локальную переменную x
  y = 13;        // Создать переменную временной диаграммы y
  z = 2;         // Изменить переменную временной диаграммы z
}
createVars();    // Вызвать функцию
trace(x);        // x не определен (x умер после завершения функции)
trace(y);        // y равен 13 (y существует после конца функции)
trace(z);        // z равен 2 (z был устойчиво изменен функцией)
```

Правила создания переменных временной диаграммы применимы, даже если наша функция является методом объекта. Мы не станем рассказывать

об объектах, пока не доберемся до главы 12, но для тех, кто знаком с объектно-ориентированным программированием, в примере 9.2 приведен код, доказывающий это. Обратите внимание, что переменная `x` определяется на временной диаграмме, потому что она не существует в локальной области видимости *newFunc( )*.

Пример 9.2. Область видимости переменных внутри методов объекта

```
newObj = new Object();                    // Создать объект
newObj.newFunc = function() { x = 12; };  // Прикрепить новый метод
newObj.newFunc ();                        // Вызвать метод

// Теперь проверим x
trace("x равен " + x);                       // x равен 12
trace("newObj.x равен " + newObj.x);         // newObj.x не определен
```

Обращение внутри функции к переменным временной диаграммы с помощью неквалифицированных ссылок может показаться удобным, но зачастую оно вносит путаницу, особенно в больших проектах. Разумнее избавляться от неоднозначности при обращении к переменным с помощью квалифицированных ссылок (то есть предварять переменную явной ссылкой на объект – `this.x` , а не `x`, как говорится в разделе «Доступ к переменным других временных диаграмм» главы 2).

Еще раз о параметрах функции

Итак, мы освоились с функциями, и теперь вернемся к обсуждению их параметров. Для этого требуется некоторое знание объектов, вот почему новичкам в программировании сначала может потребоваться прочесть главу 12.

Количество параметров

Ранее мы узнали, что параметры функции объявляются при ее создании. Вспомним синтаксис:

```
function funcName (param1, param2, ...paramn) {
  statements
}
```

На первый взгляд удивительно, что количество параметров, передаваемых функции, может отличаться от количества параметров, указанных в формальном объявлении функции. Функции могут принимать любое количество параметров – как больше, так и меньше «расчетного». Если функция вызывается с меньшим количеством параметров, чем объявлено, то значения недостающих параметров становятся равными `undefined`. Например:

```
function viewVars (x, y, z) {
  trace ("x is " + x);
  trace ("y is " + y);
  trace ("z is " + z);
}
```

```
viewVars(10);  // Во Flash Player 6 выводит:
               // x is 10
               // y is undefined
               // z is undefined
               // (Во Flash Player 5 неопределенные переменые выводятся
               // в окно Output как пустые строки.)
```

Если функция вызывается с количеством параметров, *большим* объявленного, то значения лишних параметров можно получить с помощью объекта *arguments*. (Очевидно, что к лишним параметрам нельзя обратиться по имени, как к тем, которые были объявлены непосредственно.)

Объект arguments

Во время выполнения любой функции через встроенный объект *arguments* можно получить доступ к следующей информации:

- количество параметров, переданных функции;
- массив, содержащий значения всех этих параметров;
- ссылка на выполняемую функцию;
- ссылка на функцию, из которой был сделан вызов, если таковая есть.

Объект *arguments* на самом деле представляет собой некоторый особый гибрид массива с объектом, обладающим дополнительными свойствами. Во Flash 5 *arguments* не был экземпляром массива. Во Flash MX, однако, *arguments* является экземпляром класса *Array* и потому поддерживает методы и свойства *Array*.

Извлечение значений параметров из массива аргументов

Массив `arguments` позволяет получить значение любого аргумента, с которым была вызвана функция, независимо от того, был ли он определен формально в объявлении функции. Доступ к параметру осуществляется по индексу массива `arguments`:

```
arguments[n]
```

где *n* представляет собой индекс аргумента, к которому мы обращаемся. Первый аргумент (самый левый в вызове функции) хранится под индексом 0, и обращение к нему выглядит как `arguments[0]`. Последующие аргументы хранятся по порядку: вторым аргументом является `arguments[1]`, третьим – `arguments[2]` и т. д.

Находясь внутри функции, можно сказать, сколько параметров было передано функции, выполняющейся в данный момент, определив число элементов в `arguments`:

```
numArgs = arguments.length
```

Можно легко организовать цикл по всем аргументам, переданным функции, и вывести их в окно Output, как показано в примере 9.3.

Пример 9.3. Вывод неизвестного количества аргументов

```
function showArgs() {
  for (var i = 0; i < arguments.length; i++) {
    trace("Argument " + (i + 1) + " is " + arguments[i]);
  }
}
showArgs(123, 23, "skip intro");
// Выводит...
Argument  1 is 123
Argument  2 is 23
Argument  3 is skip intro
```

Массив `arguments` позволяет создавать очень гибкие функции, принимающие произвольное количество аргументов.

Вот общая функция, которая удаляет со сцены имеющиеся на ней дубликаты экземпляров клипов:

```
function killClip () {
  for (var i = 0; i < arguments.length; i++) {
    arguments[i].removeMovieClip();
  }
}
killClip(clip10, clip5, clip13);
```

Упражнение для читателя: модифицируйте нашу прежнюю функцию *combine( )*, чтобы она принимала произвольное количество входных данных. Какие еще функции могут выиграть благодаря принятию произвольного количества параметров? Напишите функцию, возвращающую среднее значение для некоторого количества чисел? (Подсказка: сложите все аргументы, а затем разделите на количество аргументов.)

Свойства callee и caller

Как мы видели, массив `arguments` позволяет извлекать аргументы функции. У объекта *arguments* есть два свойства – `callee`, которое возвращает ссылку на выполняемую функцию, и `caller`, которое хранит ссылку на функцию, вызвавшую текущую выполняемую функцию (или `null`, если ее не вызывала никакая функция).

Как правило, нам известно имя вызываемой функции, но если выполняется анонимная функция, созданная с помощью литерала функции, то свойство `callee` может оказаться полезным. В примере 9.4 показана функция, созданная с помощью литерала, которая выполняется рекурсивно, не зная собственного имени.

Пример 9.4. Обратный счет с помощью callee

```
count = function (x) {
  trace(x);
  if (x > 1) {
    arguments.callee(x - 1);
  }
```

```
}
count(25);
```

Очевидно, обратный счет можно произвести и без рекурсивных анонимных функций. Более реалистичные примеры рекурсии функций мы увидим в разделе «Рекурсивные функции» далее в этой главе. Ссылка на текущую функцию с помощью `arguments.callee`, а не ее фактического имени, облегчает модификацию кода при изменении имени функции.

С помощью `arguments.caller` можно выполнить или определить вызвавшую функцию, как описывается в «Справочнике по языку».

Передача параметров по значению и по ссылке

Имеется еще одна тонкость, касающаяся параметров, которую мы должны рассмотреть: различие между передачей данных в функцию *по значению* и *по ссылке*.

Когда мы передаем функции элементарные данные в качестве аргументов, функция получает копию данных, а не их оригинал. Изменения, произведенные с параметром в функции, не оказывают никакого воздействия на исходный аргумент, находящийся вне функции. В примере 9.5 значение `variableName` первоначально равно 25. Изменение его на 10 внутри функции *setValue( )* не оказывает никакого влияния на значение `y`.

Пример 9.5. Передача элементарных данных по значению

```
var y = 25;
function setValue(variableName) {
  variableName = 10;
}
setValue(y);
trace("y равен " + y);  // Выводит: "y равен 25"
```

Поэтому говорят, что элементарные данные передаются *по значению*. Однако когда мы передаем в качестве аргумента функции *составные* данные, функция получает не дубликат данных, а *ссылку*, указывающую на те же данные, что и исходный аргумент. Изменение данных через переменную параметра воздействует на исходные данные, а потому и на все переменные, указывающие на те же данные, даже за пределами функции. Поэтому говорят, что составные данные передаются *по ссылке*.

В примере 9.6 изменения переменной параметра `myArray` воздействуют на внешний массив `boys`, потому что обе переменные указывают на одни и те же данные в памяти.

Пример 9.6. Модификация аргументов составных данных, передаваемых по ссылке

```
// Создадим массив
var boys = ["Andrew", "Graham", "Derek"];

// setValue() устанавливает значение первого элемента массива
```

```
function setValue(myArray) {
  myArray[0] = "Sid";   // установить значение первого элемента массива
}

// Передать наш массив в функцию
setValue(boys);

// Проверим значения элементов нашего массива
trace("Boys: " + boys);  // Выводит: "Boys: Sid,Graham,Derek"
```

Обратите внимание, что присваивание новых значений отдельным элементам массива внутри функции отражается также вне функции. Однако предположим, что вместо присваивания новых значений одному или нескольким элементам мы присвоим новое значение самому параметру. Это разорвет его связь с первоначальным аргументом. Последующие изменения переменной параметра, как показывает пример 9.7, не окажут влияния на исходный аргумент. В этом примере массив `boys` передается в качестве аргумента, но переменной параметра `myArray` немедленно присваивается значение `girls`. Последующие изменения `myArray` затрагивают массив `girls`, а не массив `boys`.

Пример 9.7. Разрыв связи между аргументом и параметром

```
// Создадим два массива
var boys  = ["Andrew", "Graham",  "Derek"];
var girls = ["Alisa",  "Gillian", "Daniella"];

// setValue() игнорирует переданный массив и модифицирует массив girls
function setValue(myArray) {
  myArray = girls;      // Пусть myArray указывает на girls, а не на boys

  myArray[0] = "Mary";  // Изменяет первый элемент girls
}

// Передать массив boys функции setValue()
setValue(boys);
trace("Boys: " + boys);   // Выводит: "Boys: Andrew,Graham,Derek"
trace("Girls: " + girls); // Выводит: "Girls: Mary,Gillian,Daniella"
```

Дополнительные сведения об элементарных и составных данных можно найти в главе 3.

Рекурсивные функции

Рекурсивной называется функция, вызывающая саму себя (используя собственное имя в теле функции). Вот простой пример, показывающий принцип рекурсии. Поскольку в коде сказано, что функция *trouble( )* должна выполняться повторно (как изображение, бесконечно отражающееся в двух зеркалах, стоящих одно напротив другого), Flash быстро превзойдет допустимую глубину рекурсии, вызвав сообщение об ошибке:

```
function trouble() {
  trouble();
}
```

Рекурсивная функция, которая вызывает саму себя без всякой проверки условий, как это делает *trouble( )*, демонстрирует *бесконечную рекурсию* (то есть эта функция могла бы вызывать себя неограниченное количество раз). Бесконечная рекурсия приводит к быстрому исчерпанию памяти компьютера. Чтобы этого не произошло, Flash устанавливает максимальное количество уровней рекурсии равным 256, а при его превышении вызов рекурсивной функции прекращается.

Реальные рекурсивные функции вызывают себя только тогда, когда удовлетворяется заданное условие (предотвращая таким образом бесконечную рекурсию). В примере 9.4 рекурсия применялась для обратного отсчета от заданного числа до 1, но очевидно, что это можно сделать и без рекурсии.

Классическим примером рекурсии служит вычисление математического *факториала* числа. Факториал 3 (записываемый в математических обозначениях как 3!) есть 3 * 2 * 1 = 6. Факториал 5 есть 5 * 4 * 3 * 2 * 1 = 120. В примере 9.8 приведена рекурсивная функция, вычисляющая факториал.

Пример 9.8. Вычисление факториала с использованием рекурсии

```
function factorial(x) {
    if (x < 0) {
      return undefined;  // Условие ошибки
    } else if (x <= 1) {
      return 1;
    } else {
      return x * factorial(x-1);
    }
}
trace (factorial(3));  // Выводит: 6
trace (factorial(5));  // Выводит: 120
```

Как правило, одну и ту же задачу можно решить несколькими способами. С помощью цикла можно вычислить факториал без рекурсии, как показано в примере 9.9.

Пример 9.9. Вычисление факториалов без рекурсии

```
function factorial(x) {
    if (x < 0) {
      return undefined; // Условие ошибки
    } else {
      var result = 1;
      for (var i = 1; i <= x; i++) {
        result = result * i;
      }
      return result;
    }
}
```

Такие два способа и представлены в примерах 9.8 и 9.9. В методе рекурсии решение формулируется так: «Факториал 6 – это 6, умноженное на факториал 5. Факториал 5 – это 5, умноженное на факториал 4...», и т. д. Нерекурсивный метод проходит в цикле числа от 1 до x и перемножает их.

Какой подход лучше выбрать, зависит от задачи. Некоторые легче решить с помощью рекурсии, но рекурсия может оказаться медленнее нерекурсивных решений. Лучше всего применять рекурсию, когда не известно, насколько глубоко вложенной может быть структура данных. Допустим, что требуется перечислить все файлы во всех вложенных подкаталогах, до бесконечности. Трудно было бы написать общее решение, которое действует для любого количества подкаталогов, не прибегая к рекурсии. На псевдокоде решение может выглядеть так:

```
function listFiles (directoryName) {
  do (проверить очередной элемент directoryName) {
    if (этот элемент сам является подкаталогом) {
      // Рекурсивно вызвать функцию с новым подкаталогом
      listFiles(subDirectoryName);
    } else {
      // Вывести имя этого файла
      trace (filename);
    }
  } while (еще имеются элементы для проверки);
}
```

В «Справочнике по языку», в статье, посвященной объекту *XML*, рассматривается применение рекурсии для перечисления всех элементов документа XML.

Вложенные функции

ActionScript позволяет объявлять одни функции внутри других. В примере 9.10 создается функция *showFullName( )* с двумя вложенными функциями – *showFirstName( )* и *showLastName( )*.

Пример 9.10. Функция с двумя вложенными функциями

```
function showFullName () {
  function showFirstName () {
    trace("Colin");
  }
  function showLastName () {
    trace("Moock");
  }

  showFirstName();
  showLastName();
}

showFullName();
```

Внутри *showFullName( )* вложенные функции вызываются как обычно. Однако, как и локальные переменные, вложенные функции доступны *только* в своей родительской функции. Код за пределами родительской функции не может выполнить вложенную функцию.

Вложенные функции часто применяются для объединения ряда сложных вычислений в один пакет, закрывая в то же время доступ извне к внутреннему механизму его работы. Интересный пример вложенных функций (иногда называемых *внутренними*), которые применяются для моделирования закрытых членов в стиле Java, можно найти здесь:

http://www.crockford.com/javascript/private.html

Область видимости вложенных функций

При выполнении вложенной функции интерпретатор добавляет ее локальную область видимости в цепочку областей видимости родительской функции. В примере 9.10 цепочка областей видимости *showFullName( )* такова:

- Глобальный объект (в нем интерпретатор завершает поиск переменных)
- Клип, содержащий функцию (здесь была определена *showFullName( )*)
- Локальная область видимости *showFullName( )* (здесь интерпретатор начинает поиск переменных)

Цепочка областей видимости *showFirstName( )* следующая:

- Глобальный объект (в нем интерпретатор завершает поиск переменных)
- Клип, содержащий функцию (здесь была определена *showFullName( )*)
- Локальная область видимости *showFullName( )*
- Локальная область видимости *showFirstName( )* (здесь интерпретатор начинает поиск переменных)

Область видимости вложенной функции такая же, как у ее родительской функции, с добавлением собственной локальной области видимости. Если переменная отсутствует в локальной области видимости *showFirstName( )*, то интерпретатор проверит локальную область видимости *showFullName( )* и т. д. вверх по цепочке. Кроме того, все переменные, определенные во внешней функции, доступны вложенным функциям даже после завершения выполнения внешней функции. В следующем примере функция *enableZoom( )* создает обработчик события *onMouseDown( )* в `theClip`, используя литерал вложенной функции.

```
function enableZoom (theClip) {
  var zoomFactor = 1.5;
  theClip.onMouseDown = function () {
    this._xscale *= zoomFactor;
    this._yscale *= zoomFactor;
  }
}

// Разрешить изменение масштаба текущего клипа.
enableZoom(this);
```

Обработчик *onMouseDown( )* будет выполняться при каждом щелчке мыши даже после того, как *enableZoom( )* завершит собственное выполнение. Однако несмотря на то что выполнение *enableZoom( )* завершено, ее переменная

zoomFactor остается доступной вложенной функции *onMouseDown( )* через цепочку областей видимости.

Приведем два более сложных примера, показывающих мощь областей видимости вложенных функций.

В примере 9.11 создается глобальная функция *setTimeout( )*, которая вызывает передаваемую ей функцию *func* по прошествии *delay* миллисекунд. Вложенная функция *callFunc( )* вызывает *func*, которая передается в качестве параметра в *setTimeout( )*. Посредством цепочки областей видимости функция *callFunc( )* может обращаться как к параметру *func* функции *setTimeout( )*, так и к ее локальной переменной id.

Пример 9.11. Запоминание ID интервала

```
_global.setTimeout = function (func, delay) {
  var id = setInterval(callFunc, delay);
  function callFunc () {
    clearInterval(id);
    func();
  }
}

// Применение:
setTimeout(sayHi, 1000);

function sayHi () {
  trace("hello world");
}
```

Пример 9.12 представляет собой вариацию на ту же тему. В нем цепочка областей видимости используется для доступа к параметрам *slideTo( )* из вложенной функции, присвоенной вызывающему обработчику события клипа *onEnterFrame( )*.

Пример 9.12. Реализация метода slideTo( )

```
//    Метод: MovieClip.slideTo()
// Описание: Анимирует движение клипа в точку (leaderX, leaderY)
//           с заданной скоростью.
// Параметры: leaderX     x-координата цели.
//            leaderY     y-координата цели.
//            speed       скорость движения (пикселов на кадр).
MovieClip.prototype.slideTo = function (leaderX, leaderY, speed) {
  // Перемещаться, только если клип не достиг цели.
  if (this._x != leaderX && this._y != leaderY) {
    // Создать вложенную функцию и назначить ее в качестве
    // обработчика события onEnterFrame() клипа.
    this.onEnterFrame = function () {
      // Определить расстояние между клипом и его конечной точкой.
      // Доступ к параметрам leaderX и leaderY осуществляется
      // через цепочку областей видимости.
      var deltaX = this._x - leaderX;
      var deltaY = this._y - leaderY;
      var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));
```

```
        // Распределить скорость по осям x и y.
        var moveX = speed * (deltaX / dist);
        var moveY = speed * (deltaY / dist);

        // Если скорость клипа достаточна, чтобы перескочить цель,
        // просто переместить его туда. В противном случае перемещаться
        // со скоростью, заданной в параметре slideTo().
        if (speed >= dist) {
          this._x = leaderX;
          this._y = leaderY;
          // Клип прибыл; остановиться, удалив onEnterFrame().
          delete this.onEnterFrame;
        } else {
          this._x -= moveX;
          this._y -= moveY;
        }
      }
    }
}

// Пример использования:
// Перемещает клип в позицию курсора при щелчке мыши.
theClip_mc.onMouseDown = function () {
  this.slideTo(_root._xmouse, _root._ymouse, 10);
}
```

Встроенные функции

Мы выяснили, как вызывать пользовательские функции, но не будем забывать, что с ActionScript поставляется коллекция встроенных функций (родственных глаголам в нашей языковой аналогии). Мы знакомы с некоторыми встроенными функциями, позволяющими обрабатывать данные. Мы также коснулись функций, управляющих фильмом Flash и окружением пользователя.

Например, для управления воспроизводящей головкой клипа можно вызвать функцию *gotoAndPlay( )* с параметром, содержащим номер кадра:

```
gotoAndPlay(5);
```

Если вы новичок в программировании, для вас может наступить прозрение. Обратите внимание: встроенные функции вызываются с помощью оператора вызова функции (круглые скобки) и списка параметров (в данном случае значения 5) в точности, как обычные функции, определенные пользователем! Встроенные функции, например *gotoAndPlay( )*, применяются точно так же, как функции, которые мы объявляем сами. Пользовательские функции обычно делают то, чего не умеют встроенные функции ActionScript. Как и у любой пользовательской функции, у каждой встроенной есть имя, ноль или более необязательных или необходимых параметров и возвращаемое значение (иногда равное `undefined`).

Хотя Flash давно называет *gotoAndPlay* «действием» («Action»), мы теперь наблюдаем его в подлинном виде – как встроенную функцию. В главе 6 мы узнали, что некоторые действия Flash являются инструкциями, а некоторые – функциями. Теперь, тщательно изучив те и другие, вы можете отличить их друг от друга.

Рассматривая некоторые действия как *функции*, вам будет легче запомнить и использовать их синтаксис. Например, чтобы загрузить фильм в проигрыватель Flash, надо знать о существовании функции *loadMovie( )* и ожидаемых ею параметрах. Заглянув в «Справочник по языку», мы находим нужные данные и можем легко составить инструкцию, подобную следующей:

```
loadMovie("myMovie.swf", "_level1");
```

Если вы освоились с функциями, то дальше дело пойдет быстрее.

Встроенные функции ActionScript многочисленны и разнообразны. Они позволяют управлять элементами фильма, наделяя возможностью исследовать и изучить все – от громкости звука до объема текста, выделенного в поле для редактирования. Исчерпывающее описание встроенных функций ActionScript приведено в «Справочнике по языку». Периодически просматривайте его, чтобы познакомиться с имеющимися функциями, хотя запоминать их специфический синтаксис нет необходимости. В дополнение к встроенным функциям, составляющим ядро языка ActionScript, Macromedia и другие разработчики предлагают библиотеки специальных функций, которые можно включать в фильмы и использовать точно так же, как встроенные. Дополнительные сведения о расширении Flash можно найти в ресурсах, перечисленных в приложении А.

Доступность внутренних функций

Одни встроенные функции доступны всюду, а другие могут вызываться только через объект. Если функция *глобальная*, то ее можно использовать везде; однако если функция представляет собой *метод* объекта, то ее следует применять вместе с надлежащим объектом. Так как мы еще не освещали объекты, а большинство встроенных функций ActionScript являются методами объектов, отложим рассмотрение встроенных методов до главы 12.

Функции как объекты

В ActionScript функции формально являются особым типом встроенного объекта. Класс *Function*, формально ставший доступным ActionScript во Flash Player 6 и документированный в «Справочнике по языку», создает отдельные объекты функций. Посмотрим, что это значит и как влияет на действия, которые можно осуществить с помощью функций.

Передача функций в функции

Любая функция может выступать в качестве аргумента другой функции, например так:

```
function1(function2);
```

Обратите внимание, что если за *function2* нет круглых скобок, то интерпретатор не выполняет *function2( )*, а просто передает ее «объектную ссылку» в качестве аргумента в *function1( )*. Это значит, что *function1( )* получает саму *function2*, а не значение, возвращаемое *function2( )*. Поскольку объекты передаются по ссылке, можно передать идентификатор функции в другую функцию, куда он попадет целым и невредимым. Переданную функцию можно выполнить следующим образом:

```
function doCommand(command) {
  command();      // Выполняет переданную функцию
}

// Несколько примеров:
doCommand(stop);  // Передает внутреннюю функцию stop()
                  // (останавливает текущий фильм)
doCommand(play);  // Передает внутреннюю функцию play()
                  // (воспроизводит текущий фильм)
```

С помощью этого приема можно присвоить функцию обратного вызова группе объектов текстовых полей:

```
function initFormFields (fields, callbackFunc) {
  // Установить функции обратного вызова при получении фокуса текстовыми полями.
  for (var i = 0; i < fields.length; i++) {
    fields[i].onSetFocus = callbackFunc;
  }
}

// Создать список текстовых полей.
editFields = [firstNameField, lastNameField, emailField,
                      phoneField, companyField, commentsField];

// Функция, окрашивающая текстовое поле в голубой цвет.
function handleFieldFocus () {
  this.background = true;
  this.backgroundColor = 0x333399;
}

initFormFields(editFields, highlightField);
```

Хранение функций в переменных

Поскольку функции являются типом объекта, можно обращаться с ними, как с любыми другими данными. В следующем примере встроенная функция *gotoAndPlay* присваивается переменной `gp`, что дает более короткий способ ссылки на функцию.

```
var gp = gotoAndPlay;  // Создает краткую ссылку на gotoAndPlay()
gp(25);                // Вызвать gotoAndPlay() посредством нашей ссылки
```

Снова обратите внимание на отсутствие круглых скобок после *gotoAndPlay* в первой строке примера. Как мы увидим в главах 10 и 12, функции часто записываются в свойства объектов для создания обработчиков событий и методов.

Прикрепление к функциям свойств и методов

Помимо передачи и хранения функций как объектов, «объектности» функций можно найти применение, прикрепляя к ним свойства, например:

```
// Создать функцию
function myFunction () {
  trace(myFunction.x);
}

// Прикрепить к ней свойство
myFunction.x = 15;

// Проверить значение свойства, вызвав функцию
myFunction();  // Выводит: 15
```

Прикрепляя к функциям свойства, можно сохранять некоторую информацию между вызовами функций, не загромождая временную диаграмму переменными. Однако обычно свойства следует записывать в объекты, а не в функции.

Свойства функций предоставляют выгоды локальных переменных, но только свойства не теряются между вызовами функции. Это полезно, когда надо вызвать функцию с уникальным идентификатором. С помощью свойств функция может хранить состояние. Обычно функция «забывает» все, что происходило в предыдущих ее вызовах, но свойства позволяют ей «помнить», что было раньше. Это очень удобно, когда требуется следить за тем, сколько раз выполнялась операция. Вот, например, общая функция, которая создает копию клипа и дает этой копии уникальные имя и уровень благодаря тому, что запоминает в переменной `makeClip.count`, сколько клипов уже скопировано:

```
makeClip.count = 0;    // Определить свойство makeClip() (вспомните,
                       // что makeClip() уже существует, так как
                       // функции определяются до выполнения кода)

  // Создать копию переданного клипа и присвоить новому клипу автоматическое имя
  function makeClip (theClip) {
     // Прибавить единицу к счетчику клипов
     makeClip.count++

    // Предыдущую строку можно записать, сделав ее более переносимой:
    // arguments.callee.count++

    // Теперь копировать клип, присвоив ему уникальные имя и глубину
```

```
    theClip.duplicateMovieClip(the clip._name + makeClip.count, makeClip.count);
}

makeClip(square);   // Создать дубликат square с помощью makeClip()

square1._x += 100;  // Переместить продублированный квадрат вправо
```

Переменная, сохраняющая свое значение между вызовами функции, в некоторых языках, например в C, называется *статической*. Статические переменные не следует путать с так называемыми *статическими функциями* языка C, где слово static означает «закрытая». Обратите внимание, что в начале нашего сценария мы инициализируем переменную `makeClip.count` значением 0, но она не обнуляется (не инициализируется повторно) при каждом вызове функции. По общему правилу, свойства функции следует использовать только для хранения информации, относящейся к этой функции. Более общую информацию надо хранить в классе или в глобальных переменных.

К функциям ActionScript можно также прикреплять другие функции, что позволяет эмулировать статические методы класса в стиле Java. Например:

```
function SomeClass () {
  // Здесь находится код конструктора.
}
SomeClass.staticMethod = function () {
  // Здесь находится код статического метода.
}
```

Здесь `SomeClass` представляет собой имя класса, а `staticMethod` — имя статического метода.

Примеры методов функций можно найти в описаниях встроенных методов *Function.call( )* и *Function.apply( )* в «Справочнике по языку». Эти методы выполняют функцию в контексте заданного объекта. Мы снова вернемся к статическим методам и свойству `prototype` в главе 12.

Возврат функций из функций

Наконец, поскольку объекты функций представляют собой данные, они могут возвращаться функциями. В следующем примере показана функция *makeGotoHandler( )*, возвращающая литерал вложенной функции для использования в качестве функции обратного вызова обработчика события клипа. Литерал функции направляет воспроизводящую головку клипа в кадр, заданный аргументом `frameNumber`.

```
function makeGotoHandler (frameNumber) {
  return function () {
    this.gotoAndStop(frameNumber);
  };
}

// Применение:
_root.onMouseDown = makeGotoHandler(30);
_root.onMouseUp = makeGotoHandler(40);
```

С помощью *makeGotoHandler( )* удобно создавать функцию обратного вызова, не вводя ее код каждый раз, когда надо его присвоить. Обратите внимание, что мы снова пользуемся цепочкой областей видимости для извлечения значения `frameNumber` из функции обратного вызова обработчика события, когда *makeGotoHandler( )* уже завершила свое выполнение. Подробности см. в разделе «Область видимости вложенных функций» ранее в этой главе.

Централизация кода

Вероятно, самое важное свойство функций состоит в том, что их можно использовать повторно, то есть создавать многократно выполняемые участки кода, которые могут быть вызваны из любого места кода. Признаками хорошего кода служат его модульность и централизация. Модули можно заменять, не нарушая работы других модулей или обращений к модифицированному модулю. Кроме того, код оказывается в одном центральном месте, что облегчает его поиск и сопровождение. Важно избегать наличия нескольких сценариев, содержащих идентичный или почти идентичный код. Вместо этого следует вызывать функции для выполнения операций, централизованных в библиотеке обобщенных функций. Модульность и централизация облегчают применение всего кода или его части в других проектах. Функции могут выполняться удаленно, поэтому их можно хранить в одном центральном месте, что облегчает сопровождение и модификацию кода.

Допустим, например, что в трех разных клипах есть три кнопки, имеющие одинаковое назначение. Вместо того чтобы создавать три экземпляра кода, помещая его в каждую кнопку, можно поместить один экземпляр кода в функцию на главной временной диаграмме и вызывать его из каждой кнопки по мере необходимости. Это бережет время, уменьшает вероятность ошибок и позволяет легко изменить поведение сразу трех кнопок. Благодаря централизации кода тестирование становится более надежным, а выявление ошибок значительно упрощается. Если одна из кнопок работает, а другая нет, то, скорее всего, проблема связана с вызовом функции второй кнопкой, а не с самой функцией.

Еще раз вопросник с вариантами ответов

В примере 1.1 мы познакомили программистов-новичков с простым фильмом со сценарием – вопросником с вариантами ответов. Вернемся к нему и посмотрим, как можно централизовать его код.

Структура слоев нашего нового вопросника остается точно такой же, как прежде. Мы изменим код только в первом кадре `quizEnd` и кнопках.

Получить файл *.fla* для исходной и модифицированной версий вопросника можно из сетевого хранилища кода.

Реорганизация кода вопросника в функции

В нашей первой попытке создания вопросника с несколькими вариантами выбора код был разбросан по всему фильму. Мы разместили логику вопросника в трех местах: в первом кадре, где инициализировали наш вопросник, в кадре `q2`, где назначали обработчики событий кнопкам вопроса 2, и в кадре `quizEnd`, где подводили счет правильным ответам. Теперь мы хотим централизовать логику вопросника, выполняя все эти задачи в первом кадре. Для инициализации обратимся к тому же ряду простых инструкций, что и в первой версии, но для сохранения ответов пользователя и подсчета результата применим две функции – *answer( )* и *gradeUser( )*. Мы также воспользуемся литералами функций, чтобы определить поведение кнопок ответов.

Пример 9.13 показывает код в первом кадре вопросника. Там мы будем хранить всю логику, необходимую для выполнения вопросника. Посмотрите на код в целом, а затем мы разберем его по частям. В этом коде пока не нашли места самые эффективные приемы, такие как использование массивов для хранения групп взаимосвязанных элементов данных. В значительной мере код по-прежнему преследует лишь иллюстративные цели – в данном случае он показывает, как централизовать код с помощью функций. Этот пример в дальнейшем будет усовершенствован.

Пример 9.13. Вопросник с несколькими вариантами выбора, версия 2

```
//  Остановить фильм на первом вопросе
stop();

//  Инициализировать переменные вопросника
var quizTimeline = this;     // Сохранить ссылку на временную диаграмму,
                             // содержащую интерфейс пользователя вопросника
var numQuestions = 2;        // Количество вопросов
var q1answer;                // Ответ пользователя на вопрос 1
var q2answer;                // Ответ пользователя на вопрос 2
var correctAnswer1 = 3;      // Правильный вариант ответа для вопроса 1
var correctAnswer2 = 2;      // Правильный вариант ответа для вопроса 2
answer.currentQuestion = 0;  // Свойство функции: текущий вопрос

//    Функция: answer()
//   Описание: регистрирует ответы пользователя
//             и переходит к новому вопросу.
//  Параметры: choice           ответ пользователя на текущий вопрос
//  Свойства: currentQuestion  текущий вопрос
function answer (choice) {
  // Функция answer() вызывается один раз для каждого вопроса. Прибавляем 1
  // к currentQuestion при каждом вызове answer(), благодаря чему знаем,
  // на какой вопрос сейчас отвечаем. Поскольку currentQuestion
  // представляет собой свойство функции,
  // оно сохраняет значение в промежутке между вызовами.
  answer.currentQuestion++;

  // Показать вопрос и ответ в окне Output для отладки.
  trace("Question " + answer.currentQuestion
```

```
        + ". The user answered: " + choice + ".");

  // Записать ответ пользователя на вопрос. Функция set()
  // динамически создает переменные вида
  // q1answer, q2answer,...qnanswer.
  set("q" + answer.currentQuestion + "answer", choice);

  // Если вопрос последний...
  if (answer.currentQuestion == numQuestions) {
    // ...перейти в кадр quizEnd.
    quizTimeline.gotoAndStop("quizEnd");
    // Оценить ответы пользователя.
    gradeUser();
  } else {
    // ...в ином случае перейти к кадру со следующим вопросом.
    quizTimeline.gotoAndStop("q"+ (answer.currentQuestion + 1));
  }
}

//   Функция: gradeUser()
//  Описание: Подсчитывает результаты пользователя в конце опроса.
function gradeUser () {
  // Сообщить в окне Output, что мы собираемся оценить ответы.
  trace("Quiz complete. Now grading...");

  // Создать локальную переменную для хранения
  // количества правильных ответов.
  var totalCorrect = 0;

  // Подсчитать количество правильных ответов.
  // Для каждого вопроса...
  for (var i = 1; i <= numQuestions; i++) {
    // Если ответ пользователя совпадает с правильным.
    if (eval("q" + i + "answer") == eval("correctAnswer" + i)) {
      // Засчитать пользователю очко.
      totalCorrect++;
    }
    // Вывести правильный ответ и ответ пользователя
    // в окне Output для отладки.
    trace("Question " + i
         + ". Correct answer: " + eval("correctAnswer" + i)
         + ". User's answer: "  + eval("q" + i + "answer"));
  }

  // Показать окончательный результат в окне Output для отладки.
  trace("User's score: " + totalCorrect + "/" + numQuestions);

  // Создать на экране текстовое поле для показа результата пользователя.
  quizTimeline.createTextField("totalOutput_txt", 1, 150, 200, 200, 20);

  // Вывести результат пользователя в текстовом поле на экране.
  quizTimeline.totalOutput_txt.text = "Your final score is: "
                                    + totalCorrect + "/" + numQuestions;

  // Настроить гарнитуру шрифта, высоту и цвет знаков для текстового окна.
```

```
    var formatObj = new TextFormat();
    formatObj.size = 16;
    formatObj.color = 0xC7FF9C;
    formatObj.font = "_sans";
    formatObj.bold = true;
    quizTimeline.totalOutput_txt.setTextFormat(formatObj);
  }

  // Код, выполняемый при нажатии кнопки 1.
  choice1_btn.onRelease = function () {
    // Вызвать answer() для записи ответа пользователя (передается как аргумент)
    // и перехода к следующему вопросу.
    this._parent.answer(1);
  };

  // Код, выполняемый при нажатии кнопки 2.
  choice2_btn.onRelease = function () {
    this._parent.answer(2);
  };

  // Код, выполняемый при нажатии кнопки 3.
  choice3_btn.onRelease = function () {
    this._parent.answer(3);
  };
```

Прежде всего, останавливаем фильм с помощью функции *stop( )*, чтобы он не прокрутил все вопросы. Затем инициализируем переменные временной диаграммы вопросника. Ответы `q1answer` и `q2answer` известны нам из первой версии. Мы также добавили:

- `numQuestions` — переменную, которую будем использовать в функциях *answer( )* и *gradeUser( )*.
- `correctAnswer1` и `correctAnswer2` — переменные, необходимые для выставления оценки.
- `answer.currentQuestion` — свойство функции, указывающее на текущий вопрос при вызове *answer( )*.
- `quizTimeline` — переменную, служащую ссылкой на клип, в котором будет показываться интерфейс вопросника. Мы пользуемся `quizTimeline`, а не `_root`, чтобы можно было загружать фильм в клип, не нарушая его работы.

Инициализировав переменные, можно создать функцию *answer( )*, которая будет регистрировать ответы пользователя и перемещать воспроизводящую головку к следующему вопросу. Функция *answer( )* ожидает один параметр, `choice`, который является номером ответа, выбранного пользователем для каждого вопроса, поэтому объявление функции начинается так:

```
function answer (choice) {
```

При каждом ответе функция увеличивает `currentQuestion` — свойство функции, хранящее номер вопроса, на который дается ответ:

```
answer.currentQuestion++;
```

Далее мы записываем выбор пользователя в динамически именуемую переменную временной диаграммы, соответствующую вопросу, на который дается ответ. Мы обращаемся к значению свойства currentQuestion, чтобы определить имя переменной временной диаграммы – q1answer, q2answer и т. д. (в следующей версии вопросника мы задействуем более элегантный массив):

```
set ("q" + answer.currentQuestion + "answer", choice);
```

Сохранив ответ пользователя в соответствующей переменной, необходимо решить, что показывать пользователю дальше:

- Если это был последний вопрос, мы показываем кадр quizEnd временной диаграммы quizTimeline и выполняем функцию *gradeUser( )*.
- В ином случае показываем следующий вопрос, находящийся в кадре с меткой "*q*" + *(answer.currentQuestion + 1)*:

```
if (answer.currentQuestion == numQuestions) {
  quizTimeline.gotoAndStop("quizEnd");
  gradeUser();
} else {
  quizTimeline.gotoAndStop("q"+ (answer.currentQuestion + 1));
}
```

Это составляет логику наших вопросов-ответов. Функция *answer( )* готова обработать ответы на любой вопрос в вопроснике. Теперь построим функцию, оценивающую эти ответы, – *gradeUser( )*.

Функция *gradeUser( )* не принимает параметры. Она должна сравнить каждый ответ пользователя с каждым правильным ответом и вывести результат. Для подсчета количества набранных очков служит переменная totalCorrect, которая в первой версии вопросника была переменной временной диаграммы, но в этой версии правильнее сделать ее локальной переменной (totalCorrect используется только функцией *gradeUser( )*, поэтому лучше хранить ее в этой функции).

Мы сравниваем ответы пользователя с правильными в цикле *for*, проходя по телу цикла столько раз, сколько вопросов в вопроснике:

```
for (i = 1; i <= numQuestions; i++) {
```

Внутри цикла выражение сравнения сверяет ответы пользователя с правильными ответами. С помощью *eval( )* мы динамически извлекаем значение каждой переменной ответа пользователя и каждой переменной правильного ответа. (Позднее мы избавимся от *eval( )* с помощью массива.) Если эти две переменные равны, инкрементируем totalCorrect:

```
if (eval("q" + i + "answer") == eval("correctAnswer" + i)) {
  totalCorrect++;
}
```

После завершения цикла в totalCorrect будет содержаться количество вопросов, на которые пользователь ответил правильно. Мы выводим это число в динамически создаваемом текстовом поле:

```
quizTimeline.createTextField("totalOutput_txt", 1, 150, 200, 200, 20);
```

Здесь totalOutput_txt – имя текстового поля, 1 – глубина поля в стеке содержимого quizTimeline, 150 и 200 – координаты поля, а 200 и 20 – его высота и ширина. Чтобы вывести значение totalCorrect в текстовом поле, присваиваем его свойству text:

```
quizTimeline.totalOutput_txt.text = "Your final score is: "
                                  + totalCorrect + "/" + numQuestions;
```

Наконец, применяем к текстовому полю специальное форматирование. Подробности представлены в описании класса *TextFormat* в «Справочнике по языку».

```
var formatObj = new TextFormat();
formatObj.size = 16;
formatObj.color = 0xC7FF9C;
formatObj.font = "_sans";
formatObj.bold = true;
quizTimeline.totalOutput_txt.setTextFormat(formatObj);
```

Вот! Теперь, когда созданы обе функции, *answer( )* и *gradeUser( )*, а переменные вопросника инициализированы, логика нашего вопросника завершена. Обратите внимание, насколько проще следить за действием вопросника, когда большая часть кода содержится в одном кадре. Осталось только вызывать функцию *answer( )* из надлежащих кнопок вопросника.

Вызов функции answer()

Мы будем вызывать функцию *answer( )* из кнопок, по которым щелкает пользователь при выборе ответа. Вот код для кнопки «Answer 1»:

```
choice1_btn.onRelease = function () {
  this._parent.answer(1);
};
```

Этот код присваивает обработчику *onRelease( )* кнопки choice1_btn литерал функции, которая будет автоматически вызываться при нажатии кнопки. Литерал функции содержит единственную команду: «На временной диаграмме клипа, содержащего кнопку this._parent, вызвать *answer( )*, передав в качестве аргумента выбранный пользователем ответ на этот вопрос (1): this._parent.answer(1);».

Обратите внимание, что обработчик choice1_btn применяется к любому вопросу, поэтому мы не должны назначать новые обработчики кнопок для каждого вопроса, как мы это делали в первой версии. Задача кнопки состоит в том, чтобы сообщить функции *answer( )*, какой выбор сделал пользователь. Функция *answer( )* знает, на какой вопрос отвечает пользователь, поэтому она записывает ответ в нужную переменную.

Если код каждой кнопки аккуратно упакован в функции *answer( )*, размер кода кнопки становится минимальным. Кроме того, создание новых кнопок сводится всего лишь к передаче в *answer( )* правильного аргумента. Вот так, например, можно добавить четвертую кнопку:

```
choice4_btn.onRelease = function () {
  this._parent.answer(4);
};
```

Это достаточно гибко, но мы по-прежнему дублируем литерал функции. В объектно-ориентированной версии нашего вопросника мы увидим, как избавиться от такой избыточности кода.

Итоги создания вопросника

Поместив весь наш код в первый кадр фильма, мы отделили *отображение* системы от ее *логики*. В результате мы не потеряем эту логику в той путанице кадров и кнопок, которую часто представляют фильмы Flash.

Проверьте вопросник и убедитесь, что он работает. Поэкспериментируйте с ним: добавляйте вопросы, удаляйте вопросы и изменяйте все, что придет в голову. По мере дальнейшего изложения материала задумывайтесь над тем, какие из изученных вами приемов могут быть применены к нашему примеру вопросника. Какие наглядные особенности можно добавить для пользователя? Какие невидимые изменения можно внести в код, чтобы достичь большей гибкости и элегантности, даже если ощущения пользователя при этом не изменятся? Сумеете ли вы изменить вопросник так, чтобы вопросы допускали более одного правильного ответа?

Вперед!

Мы рассмотрели одно из наиболее мощных средств ActionScript – функции. В следующей главе мы узнаем об особой разновидности функций, называемой *обработчиками событий*, которые автоматически выполняются интерпретатором. Обработчики событий представляют собой один из краеугольных камней интерактивности.

10

События и обработка событий

Вы много узнали о составлении команд, которые должен выполнять интерпретатор ActionScript. К настоящему времени вы вполне можете сообщать интерпретатору, что от него требуется, но как сообщить ему, *когда* должны быть выполнены соответствующие действия? Код ActionScript не выполняется сам по себе – что-то должно вызвать его выполнение.

Этим «чем-то» является либо синхронное воспроизведение фильма, либо возникновение предопределенного асинхронного *события*.

Синхронное выполнение кода

Во время воспроизведения фильма воспроизводящая головка временной диаграммы переходит от одного кадра к другому. Когда она входит в новый кадр, интерпретатор выполняет код, прикрепленный к этому кадру. После выполнения кода кадра обновляется изображение на экране и воспроизводятся звуки. Затем воспроизводящая головка перемещается в следующий кадр.

Например, если поместить код непосредственно в кадр 1 фильма, этот код выполняется перед выводом содержимого кадра 1. Если мы поместим блок кода в ключевой кадр в кадре 5 того же фильма, то будут выведены кадры с 1 по 4, затем выполнен код кадра 5, затем выведено содержимое кадра 5. О коде, выполняемом в кадрах 1 и 5, говорят, что он выполняется *синхронно*, поскольку это происходит линейным предсказуемым образом.

Весь код, прикрепленный к кадрам фильма, выполняется синхронно. Даже если некоторые кадры воспроизводятся в ином порядке благодаря выполнению команд *gotoAndPlay( )* или *gotoAndStop( )*, код всех кадров выполняется в предсказуемом порядке, синхронно с перемещением воспроизводящей головки.

Асинхронное выполнение кода, управляемое событиями

Некоторая часть кода выполняется в непредсказуемом порядке. Этот код выполняется, когда интерпретатор ActionScript обнаруживает, что произошло

какое-то *событие*. Многие события связаны с определенными действиями пользователя, такими как щелчок мыши, изменение размеров фильма или нажатие клавиши. Другие события генерируются без участия пользователя, например, при завершении воспроизведения звука или загрузки документа XML. Так же как вход воспроизводящей головки в новый кадр вызывает выполнение синхронного кода, прикрепленного к этому кадру, события могут вызывать выполнение кода, *управляемое событиями* (*event-based*). О коде, управляемом событиями (то есть выполняющемся в ответ на некоторое событие), говорят, что он выполняется *асинхронно*, потому что события могут происходить в любое время.

Синхронное программирование требует, чтобы время выполнения кода было задано заранее. Напротив, асинхронное программирование позволяет динамически реагировать на события по мере их возникновения. Асинхронное выполнение кода очень важно для ActionScript и интерактивности.

В этой главе рассматривается асинхронное (управляемое событиями) программирование во Flash.

Типы событий

Теоретически события можно разбить на две категории:

Пользовательские события

Действия, выполняемые пользователем (например, щелчок мыши или нажатие клавиши)

Системные события

Внутренние события во время воспроизведения фильма (например, появление клипа на сцене или загрузка переменных из внешнего файла)

Синтаксически ActionScript не различает пользовательские и системные события. Событие, генерируемое самим фильмом, не менее осязаемо, чем щелчок кнопкой мыши. Хотя обычно удаление, скажем, экземпляра клипа со сцены может не восприниматься как достойное внимания «событие», возможность реагировать на системные события позволяет нам лучше управлять фильмом.

События ActionScript можно разделить на категории более практично – по объектам, к которым они относятся. Все события происходят относительно некоторого объекта в среде Flash. Мы подробно рассмотрим объекты в главе 12, а пока просто примем к сведению, что содержимое Flash (кнопки, клипы, текстовые поля и т. д.) обычно представляется в виде объектов ActionScript, а мы определяем события по отношению к этим объектам.

Основными объектами и классами ActionScript, которые поддерживают события, являются:

- *Button*
- *Key*
- *LoadVars*

- *LocalConnection*
- *Mouse*
- *MovieClip*
- *Selection*
- *SharedObject*
- *Sound*
- *Stage*
- *TextField*
- *XML*
- *XMLSocket*

Подробные сведения о конкретных событиях, поддерживаемых этими объектами, можно найти в описании каждого объекта в «Справочнике по языку».

Обработка событий

Не все события запускают какой-либо код или оказывают воздействие на фильм. Например, пользователь может генерировать десятки событий, многократно щелкая по кнопке, но эти щелчки будут игнорироваться. Почему? Потому что сами по себе события не могут вызвать выполнение кода, – надо написать код, явно реагирующий на события. Чтобы указать интерпретатору выполнить некоторый код в ответ на событие, мы должны создать *обработчик события* (*event handler*) или *приемник события* (*event listener*), описывающий действия, которые необходимо совершить при возникновении указанного события.

В силу особенностей эволюции ActionScript в нем существует четыре способа обработки событий:

- Свойства-обработчики событий
- Приемники событий
- Обработчики событий кнопок с синтаксисом:

```
on (eventName) {
  statements
}
```

- Обработчики событий клипов с синтаксисом:

```
onClipEvent (eventName) {
  statements
}
```

Рассмотрим отдельно каждый из этих механизмов обработки событий. Последующее обсуждение предполагает понимание читателем объектов, о которых рассказывается в главе 12.

Свойства-обработчики событий

Обработчики событий названы так потому, что они *перехватывают*, или *обрабатывают*, события в фильме. Обработчик событий – это функция, автоматически вызываемая при возникновении определенного события. Мы знаем, что все события связаны с какими-либо объектами. Поэтому для создания обработчика события нужно задать три параметра:

- Объект, к которому применимо событие
- Имя обработчика события
- Функцию, которая должна быть выполнена при возникновении события (называемую *функцией обратного вызова* обработчика)

Допустим, что мы хотим анимировать клип с именем `ball` путем изменения его положения при показе Flash Player каждого нового кадра. Если мы определим для клипа `ball` обработчик *onEnterFrame( )*, то последний будет выполняться на каждом такте смены кадров. Поэтому для клипа `ball` обработчик события *onEnterFrame( )* можно создать так:

1. Создать новый фильм Flash.
2. Создать новый символ с именем *ballSymbol*, содержащий круг.
3. Поместить экземпляр клипа *ballSymbol* на слой *1*.
4. Дать экземпляру *ballSymbol* имя `ball`.
5. К кадру 1 главной временной диаграммы прикрепить такой код:

```
ball.onEnterFrame = moveDown;
function moveDown () {
  // В каждом кадре перемещать ball вниз на 5 пикселов.
  this._y += 5;
}
```

В этом коде `ball` представляет собой объект, к которому относится событие, *onEnterFrame* – событие, которое мы хотим обрабатывать, а *moveDown* – функцию, которая должна выполняться при возникновении *onEnterFrame( )*.

Обратите внимание, что мы присваиваем *onEnterFrame* ссылку на функцию *moveDown*, а не значение, возвращаемое при вызове *moveDown( )*. Поэтому оператор вызова функции `()` указывать не надо.

```
// Неверный код! Оператор () использовать не надо.
ball.onEnterFrame = moveDown();
// Это правильный код.
ball.onEnterFrame = moveDown;
```

Для краткости при назначении обработчика событий можно задать функцию обратного вызова с помощью литерала, например:

```
ball.onEnterFrame = function () {
  this._y += 5;
};
```

Пример ball показывает, как задать обработчик для клипа, находясь за его пределами (например, с главной временной диаграммы). Чтобы определить обработчик события для текущего клипа из кадра на его собственной временной диаграмме, применяется другой синтаксис:

```
this.theEventHandler = function () {
  statements
};
```

В целом синтаксис обработчика событий будет либо таким:

```
someObject.someEventHandler = function () {
  statements
};
```

и тогда обработчик события определяется как литерал функции, либо таким:

```
someObject.someEventHandler = someFunction;
```

и тогда *someFunction* тоже должна быть где-то определена:

```
function someFunction () {
  statements
}
```

Последний подход предпочтительнее в тех ситуациях, когда обработчик события многократно добавляется и удаляется, либо если одна и та же функция служит обработчиком событий для нескольких объектов.

Обработчики событий могут также получать параметры, генерируемые событием. Например, в следующем коде текстовое поле userName_txt определяет обработчик события *onSetFocus( )*, который получает ссылку на текстовое поле, находившееся в фокусе до него, через параметр oldFocus. Обработчик использует oldFocus, чтобы покрасить в черный цвет предыдущее текстовое поле, которое находилось в фокусе.

```
userName_txt.onSetFocus = function (oldFocus) {
  // Покрасить в черный цвет предыдущее текстовое поле, которое находилось в фокусе
  oldFocus.backgroundColor = 0x000000;
  // Покрасить в оранжевый цвет текстовое поле, получившее фокус
  this.backgroundColor = 0xF5BF70;
}
```

Те, кому знаком объектно-ориентированный подход в программировании, должны заметить совпадение синтаксиса обработчика событий с синтаксисом определения *метода* объекта. Такое совпадение не случайно. Обработчики событий, в сущности, представляют собой методы, автоматически вызываемые интерпретатором. Поскольку в ActionScript методы являются свойствами объектов, в которых хранятся функции, формат обработчика событий, показанный в примере *onSetFocus( )*, называют *свойством-обработчиком события*.

В теле функции обратного вызова обработчика события ключевое слово this указывает на объект, для которого было сгенерировано событие. В нашем

обработчике *onEnterFrame( )* мы воспользовались `this` для ссылки на объект клипа `ball`, который переместили вниз на 5 пикселов:

```
this._y += 5;
```

Аналогично в обработчике *userName_txt.onSetFocus( )* `this` ссылается на `userName_txt`:

```
this.backgroundColor = 0xF5BF70;
```

Чтобы удалить обработчик события из объекта, используйте оператор *delete*:

```
delete theObject.theEventHandler;
```

Например, после выполнения следующего кода с главной временной диаграммы клип `ball` перестает реагировать на событие *onEnterFrame( )*:

```
delete ball.onEnterFrame;
```

Если бы код был прикреплен к самому клипу, мы могли бы написать так:

```
delete this.onEnterFrame;
```

Список обработчиков событий, предлагаемых конкретным классом или объектом, можно найти в «Справочнике по языку». В частности, обязательно изучите классы *Button* и *MovieClip*, которые определяют большинство событий, связанных с взаимодействием с пользователем. Например, чтобы заставить функцию *submitForm( )* выполняться при нажатии кнопки Submit, используем код:

```
submit_btn.onRelease = submitForm;
function submitForm () {
  // Проверить данные и отправить их на сервер...
}
```

здесь *submit_btn* – это объект кнопки, событие *onRelease* генерируется при нажатии и последующем освобождении кнопки, а *submitForm( )* – это пользовательская функция, посылающая данные формы на сервер. Законченный пример передачи данных формы представлен в главе 17.

Обработчики событий удобны, но обладают рядом недостатков: на этапе исполнения можно зарегистрировать (назначить) только одну функцию обратного вызова для обработчика события. Это означает, что при использовании свойств-обработчиков событий для каждого события может быть выполнено только одно действие. В следующем разделе мы узнаем, как преодолеть это ограничение с помощью *событий приемников* (*listener events*).

События приемников

Теоретически события приемников аналогичны свойствам-обработчикам событий. Подобно свойствам-обработчикам событий события приемников обрабатываются методом объекта, названным особым образом (то есть свойством, содержащим функцию обратного вызова), который вызывается при

возникновении данного события. Однако события приемников вводят в обработку событий две новые существенные возможности:

- Объект может реагировать на события, генерируемые другими объектами, а не только им самим.
- На событие могут реагировать несколько объектов, а не один.

Допустим, например, что мы хотим перемещать текстовое поле `title_txt` в центр фильма при каждом изменении размеров фильма. Обратившись к «Справочнику по языку», мы узнаем, что при изменении размеров фильма генерируется событие приемника *Stage.onResize( )*. Наше текстовое поле не является экземпляром *Stage*, тем не менее, мы хотим, чтобы оно реагировало на события *Stage.onResize( )*. Проблем нет. Поскольку *Stage.onResize( )* представляет собой событие приемника, оно позволяет полю `title_txt` (и любому другому объекту) зарегистрироваться, чтобы получать уведомление, когда происходит *onResize( )*.

Рассмотрим это подробнее. Сначала мы создаем метод *onResize( )* непосредственно в `title_txt` – так же, как мы ранее создавали свойства-обработчики событий:

```
title_txt.onResize = function () {
  // Выровнять текстовое поле по центру.
  // Stage.width и Stage.height всегда хранят текущие размеры фильма.
  // В теле функции обратного вызова ссылаемся на title_txt с помощью this.
  this._x = Stage.width/2 - this._width/2;
  this._y = Stage.height/2 - this._height/2;
}
```

Затем мы должны сообщить объекту *Stage*, что хотим выполнять метод *onResize( )* для `title_txt` при возникновении события *onResize( )*. Этот шаг известен как «добавление приемника». Так называемый приемник – это объект (в нашем случае `title_txt`), желающий получать уведомления о событии от *источника события*, называемого также *передатчиком события* (в нашем случае *Stage*). Все источники событий приемников предоставляют специальный метод *addListener( )* для регистрации новых приемников. Вот, например, как можно добавить поле `title_txt` в число приемников *Stage*:

```
Stage.addListener(title_txt);
```

В результате при возникновении события *Stage.onResize( )* вызывается *title_txt.onResize( )*. На практике приемник *Stage*, каким является `title_txt`, будет получать уведомления о событиях для всех событий приемников *Stage* (хотя в случае *Stage* определено только одно событие приемника). Некоторые источники событий определяют и свойства-обработчики, и события приемника. В этом случае объект приемника будет получать уведомления только о событиях приемника, а не о свойствах-обработчиках событий. События приемника и свойства-обработчики событий для каждого объекта перечислены в «Справочнике по языку».

Обычно событие реализуется в виде свойства-обработчика, если оно представляет интерес только для объекта, который его генерирует. События

приемников чаще встречаются тогда, когда одним событием интересуются несколько объектов разного рода (например, клипу, кнопке и текстовому полю может понадобиться знать об изменении размера фильма, поэтому *Stage.onResize( )* является событием приемника).

Вот общий синтаксис создания приемника события (объекта, в котором определен метод, обрабатывающий событие приемника):

```
listenerObject.eventName = function () {
  statements
}
eventSource.addListener(listenerObject);
```

Объект `listenerObject` может быть любым. Свойство `eventName` представляет собой имя предопределенного события, уведомления о котором следует ждать (например, *onResize* в *title_txt.onResize( )*). Объект `eventSource` уведомляет `listenerObject` при возникновении `eventName`. Следовательно, метод приемника будет выполнен только в том случае, если `eventName` входит в число событий, о которых оповещает `eventSource`. Подробные сведения о конкретных источниках событий представлены в описаниях объектов *Key*, *Mouse*, *Selection*, *TextField* и *Stage* в «Справочнике по языку».

Вернемся к нашему примеру. Допустим, что мы хотим выровнять по центру второе текстовое поле, `subtitle_txt`, расположив его под `title_txt`. Мы выполним ту же процедуру, что и с `title_txt`. Сначала создадим метод *onResize( )* непосредственно в `subtitle_txt`:

```
subtitle_txt.onResize = function () {
  this._x = Stage.width/2 - this._width/2;
  this._y = (Stage.height/2) + (title_txt._height/2) + 10;
}
```

Затем зарегистрируем `subtitle_txt` как приемник *Stage*:

```
Stage.addListener(subtitle_txt);
```

Обратите внимание, что следует корректно написать методы *onResize( )* для каждого объекта, который мы хотим переместить. В данном случае мы перемещаем `title_txt`, исходя из ширины и высоты *Stage* и самого текстового объекта. Сместим `subtitle_txt` по вертикали, учтя центр *Stage* и высоту `title_txt`.

Для правильной работы событие *Stage.onResize( )* требует, чтобы значения `Stage.scaleMode` и `Stage.align` были установлены, как показано в примере 10.1.

Добавление объект в список приемников источника событий иногда называют *«подпиской»* (*subscribing*). Теперь у нас есть два текстовых поля со своими независимыми методами *onResize( )*, которые реагируют на одно событие – *Stage.onResize( )*. В качестве приемников событий *Stage* можно регистрировать клипы, кнопки и любые другие объекты, что позволяет динамически настраивать расположение всех объектов фильма при изменении его размеров.

При регистрации нескольких приемников у источника событий их методы выполняются в том порядке, в каком они добавлялись.

Удалить, или «*вычеркнуть*» (*unsubsribe*), любой объект из списка приемников источника событий можно с помощью метода *removeListener( )*. Например, следующий код прекращает получение объектом `subtitle_txt` событий *onResize*, поэтому он перестанет перемещать себя при изменении размеров фильма:

```
Stage.removeListener(subtitle_txt);
```

Общий формат удаления приемника такой:

```
eventSource.removeListener(listenerObject);
```

В примере 10.1 приводится код для размещения текстового поля в центре фильма. Чтобы протестировать его, прикрепите код к первому кадру главной временной диаграммы нового фильма; затем экспортируйте фильм и попробуйте изменить размер окна фильма. Полная информация о построении интерфейсов с изменяемыми размерами представлена в описании объекта *Stage* в «Справочнике по языку».

Пример 10.1. Привязка двух текстовых полей к центру фильма

```
// Установить режим масштабирования фильма
Stage.scaleMode = "noScale";
Stage.align = "LT";

// Создать первое текстовое поле
this.createTextField("title_txt", 1, 0, 0, 180, 20);
title_txt.text = "Welcome to my website.";
title_txt.border = true;

// Назначить для него функцию приемника событий
title_txt.onResize = function () {
  this._x = Stage.width/2 - this._width/2;
  this._y = Stage.height/2 - this._height/2;
}

// Создать второе текстовое поле
this.createTextField("subtitle_txt", 2, 0, 0, 180, 20);
subtitle_txt.text = "Thanks for visiting.";
subtitle_txt.border = true;

// Назначить для него функцию приемника событий
subtitle_txt.onResize = function () {
  this._x = Stage.width/2 - this._width/2;
  this._y = (Stage.height/2) + (title_txt._height/2) + 10;
}

// Зарегистрировать текстовые поля как приемники событий Stage
Stage.addListener(subtitle_txt);
Stage.addListener(title_txt);
```

Обработчики событий on() и onClipEvent() во Flash 5

Ранее в этой главе мы обсуждали, как назначить функции обратного вызова свойствам-обработчикам событий *MovieClip.onEnterFrame( )* и *TextField.onSetFocus( )*. Во Flash 5 такого рода свойства-обработчики событий доступны только в объектах *XML* и *XMLSocket*. Для клипов и кнопок применялся особый синтаксис обработчика событий, прикрепляемого прямо к экземплярам клипов или кнопкок в среде разработки Flash. Обработчики событий кнопок определялись с помощью *on(eventName)*, а обработчики событий клипов – с помощью *onClipEvent(eventName)*, где *eventName* указывало на событие, которое должно обрабатываться. Во Flash MX применение этих старых форматов не рекомендовано. Свойства-обработчики событий и приемники событий предпочтительней, так как они способствуют отделению кода от видимых элементов фильма. Однако старый синтаксис по-прежнему поддерживается, а в нескольких особых случаях он действительно необходим (примеры можно найти в «Справочнике по языку» для *MovieClip.onLoad( )* и *Button keyPress*).

Обработчики событий кнопок *on( )* в стиле Flash 5 имеют следующий формат:

```
on (eventName) {
  statements
}
```

eventName представляет собой имя соответствующего свойства-обработчика Flash MX, но не содержит слова «on» и начинается со строчной буквы. Например, свойство *onRelease( )* из Flash MX реализовывалось во Flash 5 так:

```
on (release) {
  statements
}
```

При экспорте в формат Flash 5 необходимо соблюдать синтаксис событий в стиле Flash 5. Обработчик для одной кнопки может отвечать на несколько событий, которые разделяются запятыми. Например:

```
on (rollOver, rollOut) {
  // Вызвать пользовательскую функцию в ответ на оба события - rollOver и rollOut
  playRandomSound();
}
```

Модифицировать такой код в соответствии с предпочтительным синтаксисом Flash MX, чтобы один и тот же обработчик обратного вызова реагировал на оба события, можно так:

```
theButton.onRollOver = playRandomSound;
theButton.onRollOut = playRandomSound;
```

или еще более кратко:

```
theButton.onRollOver = theButton.onRollOut = playRandomSound;
```

Обработчики событий клипов *onClipEvent( )* в стиле Flash 5 имеют вид:

```
onClipEvent (eventName) {
  statements
}
```

Как и раньше, *eventName* – это имя соответствующего свойства-обработчика события Flash MX, но без слова «on». Например, обработчик события Flash MX *MovieClip.onEnterFrame( )* во Flash 5 реализовывался так:

```
onClipEvent (enterFrame) {
  statements
}
```

В отличие от обработчиков кнопок, обработчики клипов могут реагировать только на одно событие.

События, появившиеся после Flash 5, не доступны в формате *on( )* или *onClipEvent( )*. Например, не существует *onClipEvent(setFocus)*, потому что событие *onSetFocus( )* появилось лишь во Flash MX и впервые стало поддерживаться во Flash Player 6.

Во Flash MX экземпляры клипов могут иметь обработчики событий как вида *onClipEvent( )*, так и *on( )* в стиле кнопок. Подробные сведения представлены в разделе «Использование клипов в качестве кнопок» главы 13.

Прикрепление обработчиков событий к кнопкам и клипам

Для того чтобы прикрепить обработчик события *on( )* или *onClipEvent( )* в стиле Flash 5 к кнопке или клипу, надо физически расположить код обработчика в кнопке или клипе. Сделать это можно только в среде разработки Flash, выбрав объект на сцене и введя соответствующий код в панели Actions, как показано на рис. 10.1.

Рис. 10.1. *Прикрепление обработчика событий к кнопке*

Попробуем создать две простые функции обработчиков событий, одну для кнопки, а другую – для клипа. Сначала для кнопки:

1. Создайте новый пустой фильм Flash.
2. Создайте кнопку и перетащите ее экземпляр на основную сцену.
3. Не снимая выделения с кнопки, введите в панели Actions следующий код:

```
on (release) {
  trace("You clicked the button");
}
```

4. Выберите пункт меню Control → Test Movie.
5. Щелкните по кнопке. В окне Output появится сообщение «You clicked the button».

Когда во время воспроизведения фильма мы нажимаем и отпускаем кнопку мыши, интерпретатор обнаруживает событие *release* и запускает обработчик события *on(release)*. Сообщение «You clicked the button» появляется в окне Output при каждом нажатии и отпускании кнопки мыши.

Теперь попробуем создать более интересный обработчик события для клипа.

1. Создайте новый пустой фильм Flash.
2. На главной сцене фильма нарисуйте прямоугольник.
3. Выберите пункт меню Insert → Convert to Symbol.
4. В диалоговом окне Convert to Symbol присвойте новому символу имя *rectangle* и выберите Movie Clip в качестве Behavior.
5. Нажмите кнопку OK и завершите создание клипа `rectangle`.
6. Выделите на сцене клип `rectangle` и введите в панели Actions код:

```
onClipEvent (keyDown) {
  this._visible = false;
}
onClipEvent (keyUp) {
  this._visible = true;
}
```

7. Выберите пункт меню Control → Test Movie.
8. Щелкните по фильму, чтобы он стал получать данные с клавиатуры, затем нажмите и удерживайте любую клавишу. При каждом нажатии клавиши клип `rectangle` исчезает. При каждом освобождении нажатой клавиши `rectangle` появляется снова.

В отличие от Flash 6, во Flash 5 обработчики событий *on( )* и *onClipEvent( )* нельзя прикреплять и удалять с помощью ActionScript во время воспроизведения фильма. Поэтому следующее присваивание свойству-обработчику событий во Flash 5 не допускается:

```
theClip.onKeyDown = function () {
  _visible = 0;
};
```

Позже, в разделе «Динамические обработчики событий клипов», мы увидим, как обойти это ограничение.

Срок жизни обработчиков событий

Срок жизни обработчиков событий связан с существованием объектов, с которыми они ассоциированы. Когда клип или кнопка удаляются со сцены или прекращает существование объект *XML* или *TextField*, заканчивается и существование всех связанных с ними обработчиков событий. Объект должен присутствовать на сцене или в коде, чтобы его обработчики событий оставались действующими.

Область видимости обработчиков событий

Как и любые функции, инструкции в обработчике событий или приемнике выполняются в определенной области видимости. Область видимости указывает интерпретатору, относительно чего разрешать неквалифицированные ссылки на переменные, встречаемые им в теле обработчика. Рассмотрим области видимости обработчиков для четырех реализаций событий, поддерживаемых Flash.

Область видимости свойств-обработчиков событий и методов приемников событий

Методы приемников событий и свойства-обработчики событий подчиняются тем же правилам областей видимости, что и функции. Функция обратного вызова имеет цепочку областей видимости функции, которая определяется в момент определения функции. Кроме того, свойства-обработчики событий и методы приемников событий определяют локальную область видимости. Все правила для областей видимости функций, описанные в главе 9, применимы к свойствам-обработчикам событий и методам приемников событий.

Область видимости обработчиков onClipEvent()

В отличие от свойств-обработчиков событий и методов приемников событий, обработчики *on( )* и *onClipEvent( ) не определяют локальной области видимости*. Когда мы прикрепляем к клипу обработчик *onClipEvent( )*, областью видимости обработчика становится клип, а не сам обработчик. Это означает, что все переменные извлекаются из временной диаграммы клипа. Например, если мы прикрепляем обработчик *onClipEvent(enterFrame)* к клипу с именем `navigation` и пишем *trace(x)*; то интерпретатор ищет значение `x` на временной диаграмме `navigation`:

```
onClipEvent (enterFrame) {
  trace(x);  // Выводит значение navigation.x
}
```

Интерпретатор не начинает поиск с локальной области видимости, потому что ее не существует. Если мы напишем *var y = 10;* в обработчике *onClipEvent()*, то y будет определена на временной диаграмме navigation, несмотря на то что ключевое слово *var* в функции обычно объявляет локальную переменную.

Проще всего запомнить правила области видимости обработчика *onClipEvent( )*, если рассматривать инструкции в обработчике так, как если бы они были прикреплены к кадру клипа, к которому прикреплен этот обработчик. Допустим, например, что на главной временной диаграмме есть клип с именем ball, в котором объявлена переменная xVelocity. Для доступа к xVelocity внутри обработчика *onClipEvent( )* объекта ball мы просто ссылаемся на нее непосредственно, например:

```
onClipEvent (mouseDown) {
  xVelocity += 10;
}
```

Нет необходимости задавать путь к переменной в виде _root.ball.xVelocity, поскольку интерпретатор уже предполагает, что имеется в виду переменная xVelocity в ball. То же самое справедливо в отношении свойств и методов; вместо ball._x можно написать _x, а вместо *ball.gotoAndStop(5)* – *gotoAndStop(5)*. Например:

```
onClipEvent (enterFrame) {
  _x += xVelocity;                  // Переместить шарик
  gotoAndPlay(_currentframe - 1);   // Двухкадровый цикл
}
```

Можно даже определить функцию в клипе ball с помощью объявления функции в обработчике *onClipEvent( )*, например:

```
onClipEvent (load) {
  function hideMe () {
    _visibility = 0;
  }
}
```

Однако неквалифицированные ссылки неоднозначны. По сравнению с их поведением в обработчике *onClipEvent( )*, они ведут себя иначе в свойстве-обработчике события или приемнике событий. Поэтому при обращении к клипу, в котором определен обработчик, разумно употреблять ключевое слово this:

```
// Предпочтительное обращение к клипу с обработчиком onClipEvent():
this._x
// Допустимое, но не рекомендуемое обращение к клипу с обработчиком onClipEvent():
_x

// Предпочтительнее:
this.gotoAndStop(12);
// Не рекомендуется:
gotoAndStop(12);
```

Кроме того, this необходимо, когда динамически генерируется имя одного из свойств текущего клипа (имя переменной или вложенного клипа). Вот как заставить проигрываться один из вложенных клипов ball.stripe1, ball.stripe2 и т. д. в зависимости от текущего кадра клипа ball:

```
onClipEvent (enterFrame) {
  this["stripe" + this._currentframe].play();
}
```

Ключевое слово this также необходимо использовать с методами клипов, которым при вызове требуется явная ссылка на объект клипа. Любой метод клипа, имя которого совпадает с какой-либо глобальной функцией ActionScript, должен употребляться с явной ссылкой на клип. Таким образом, ключевое слово this необходимо указывать при вызове следующих функций как методов внутри обработчика *onClipEvent( )*:

duplicateMovieClip( )
loadMovie( )
loadVariables( )
removeMovieClip( )
startDrag( )
unloadMovie( )

Например:

```
this.duplicateMovieClip("ball2", 1);
this.loadVariables("vars.txt");
this.startDrag(true);
this.unloadMovie();
```

О двойственной природе этих функций подробно рассказано в разделе «Проблемы перекрытия методов и глобальных функций» главы 13.

Обратите внимание, что ключевое слово this позволяет ссылаться на текущий клип, даже если этому клипу не присвоено имя экземпляра в среде разработки или когда имя клипа нам не известно. На практике с помощью ключевого слова this можно передать текущий клип в виде ссылки в функцию, даже не зная имени текущего клипа. Проиллюстрируем это следующим кодом:

```
// КОД В ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
// Вот общая функция, перемещающая любой клип
function move (clip, x, y) {
  clip._x += x;
  clip._ y += y;
}

// КОД В КЛИПЕ
// Вызвать функцию главной временной диаграммы, чтобы переместить
// текущий клип, передав ей ссылку с помощью ключевого слова this
onClipEvent (enterFrame) {
  _root.move(this, 10, 15);
}
```

Во Flash Player 5.0.30.0 есть ошибка, из-за которой *gotoAndStop( )* и *gotoAndPlay( )* не работают внутри обработчика событий клипа, если им передается метка в виде строкового литерала. Такие команды просто игнорируются. Например, не будет работать такой код:

```
onClipEvent (load) {
    gotoAndStop("intro"); // Не работает во Flash 5.0.30.0
 }
```

Чтобы обойти эту ошибку, используйте в явном виде ссылку на клип, которым управляете, то есть this:

```
onClipEvent(load) {
    this.gotoAndStop("intro");
 }
```

Помните, что инструкции в обработчиках событий клипа имеют областью видимости временную диаграмму *клипа*, а не локальную область видимости функции обработчика и не *родительскую* временную диаграмму клипа (ту, на которой расположен клип).

Например, допустим, что клип ball помещен на главную временную диаграмму фильма, на которой (а не на временной диаграмме ball) определена функция *moveBall( )*. Можно по рассеянности вызвать *moveBall( )* из обработчика события для ball следующим образом:

```
onClipEvent (enterFrame) {
  moveBall();  // Ничего не делает! Функции moveBall() в ball нет.
               // Функция moveBall() определена в _root
}
```

Необходимо явным образом сослаться на функцию *moveBall( )* в главной временной диаграмме через _root:

```
onClipEvent (enterFrame) {
  _root.moveBall();  // Теперь работает!
}
```

Область видимости обработчиков on()

Область видимости обработчиков *on( )* зависит от типа объектов, к которым они прикреплены. Область видимости обработчиков *on( )*, прикрепленных к кнопкам, ограничена временной диаграммой, на которой располагается кнопка. Если обработчик *on( )* прикреплен к клипу, то его область видимости ограничена клипом, в котором находится обработчик, как и в случае с обработчиком *onClipEvent( )*. До Flash MX клипы не поддерживали обработчики *on( )* (во Flash 5 это делали только кнопки).

Например, если поместить кнопку на главную временную диаграмму и объявить в обработчике для этой кнопки переменную speed, то областью видимости speed будет главная временная диаграмма (_root):

```
// КОД ДЛЯ ОБРАБОТЧИКА КНОПКИ
on (release) {
  var speed = 10;  // Определяет speed в _root
}
```

В то же время, если поместить обработчик *on( )* в объект клипа `ball` и объявить переменную `speed` в обработчике, то областью видимости `speed` становится `ball`:

```
// КОД ДЛЯ ОБРАБОТЧИКА ball
on (release) {
  var speed = 10;  // Определяет speed в ball, а не в _root
}
```

Внутри обработчика *on( )* объекта кнопки ключевое слово `this` ссылается на временную диаграмму, на которой расположена кнопка:

```
// КОД ДЛЯ ОБРАБОТЧИКА КНОПКИ
on (release) {
  // Сделать клип, на котором расположена кнопка, прозрачным на 50%
  this._alpha = 50;
  // Переместить клип, на котором расположена кнопка, на 10 пикселов вправо
  this._x += 10;
}
```

Но внутри обработчика *on( )* клипа ключевое слово `this` ссылается на клип, в котором находится обработчик:

```
// КОД ДЛЯ ОБРАБОТЧИКА КЛИПА
on (release) {
  // Сделать клип с обработчиком прозрачным на 50%
  this._alpha = 50;
  // Переместить клип с обработчиком на 10 пикселов вправо
  this._x += 10;
}
```

Подробные сведения о клипах и кнопках представлены в описаниях классов *MovieClip* и *Button* в «Справочнике по языку».

Общая сводка по областям видимости

Различные области видимости, рассмотренные ранее, могли вас запутать. Чтобы определить область видимости конкретного обработчика во Flash MX, руководствуйтесь общими правилами, приведенными в табл. 10.1. Обратите внимание, что из представленных механизмов обработки событий Flash 5 поддерживал только «MovieClip с обработчиком *onClipEvent( )*» и «Кнопка с обработчиком *on( )*».

Таблица 10.1. Область видимости обработчиков событий

Механизм обработки событий	Пример	Область видимости

Свойство-обработчик события	`someTextField.onSetFocus = function () {`
	`  statements`
	`}`
		Клип, в котором находится объявление функции*

Таблица 10.1 (продолжение)

Механизм обработки событий	Пример	Область видимости

Метод приемника событий	`GUIManager.onResize = function () {`
	`  statements`
	`}`
	`Stage.addListener(GUIManager);`
		Клип, в котором находится объявление функции*

MovieClip со свойством-обработчиком событий	`theClip_mc.onPress = function () {`
	`  statements`
	`}`
		Клип, в котором находится объявление функции*

MovieClip с обработчиком *on( )*	`on (press) {`
	`  statements`
	`}`
		Клип с обработчиком в среде разработки

MovieClip с обработчиком *onClipEvent( )*	`onClipEvent (enterFrame) {`
	`  statements`
	`}`
		Клип с обработчиком в среде разработки

Кнопка со свойством-обработчиком события	`submit_btn.onRelease = function () {`
	`  statements`
	`}`
		Клип, в котором находится объявление функции*

Кнопка с обработчиком *on( )*	`on (release) {`
	`  statements`
	`}`
		Клип, на временной диаграмме которого находится кнопка

* Для обработчиков и приемников, назначаемых внутри функции, областью видимости является эта функция (в соответствии с правилами областей видимости вложенных функций, которые обсуждались в главе 9).

Далее следует сценарий, призванный помочь вам проверить, как вы усвоили понятия области видимости обработчиков. Прежде чем прочесть ответ, попробуйте найти его самостоятельно.

Предположим, что экземпляр клипа с именем `intro_mc` помещен на главную временную диаграмму фильма, к которой прикреплен следующий код:

```
intro_mc.onPress = function () {
  trace(this);  // Здесь this ссылается на intro_mc.
  this.play()   // Эта строка вызывает воспроизведение intro_mc.
  play();       // Почему эта строка не вызовет воспроизведение intro_mc?
}
```

Вопрос: почему в функции обратного вызова `intro_mc` *onPress( )* ключевое слово `this` ссылается на `intro_mc`, а неквалифицированные ссылки – нет? Иными словами, почему строка 3 в теле обработчика не вызовет воспроизведение `intro_mc`?

Ответ: `this` представляет собой ссылку на объект, через который вызывалась функция *onPress( )*: `intro_mc`. Но инструкции внутри функции *onPress( )* выполняются в области видимости временной диаграммы, на которой была определена функция, – главной временной диаграммы. Поэтому отдельная

команда *play( )* запускает воспроизведение главной временной диаграммы, а не `intro_mc`.

Мораль этой истории такова: очевидно, что местонахождение объявления функции может измениться, поэтому применение неквалифицированных ссылок внутри обработчиков небезопасно. Квалифицируйте внутри обработчиков все идентификаторы, не являющиеся локальными, с помощью `this` или других явных ссылок на объект.

Значения ключевого слова this

Изучая средства обработки событий ActionScript, мы несколько раз сталкивались с ключевым словом `this`. Внутри обработчика события оно обычно указывает на объект, который получил уведомление о событии. Однако развитие Flash и создание нескольких систем событий осложнило ситуацию. В табл. 10.2 приведены значения ключевого слова `this` в различных механизмах обработки событий ActionScript.

Таблица 10.2. Значения ключевого слова this

Механизм обработки событий	Значение this
Свойство-обработчик события	Объект, который определяет свойство обработчика
Метод приемника события	Объект, зарегистрированный как приемник
MovieClip со свойством-обработчиком события	Клип, в котором определено свойство-обработчик
MovieClip с обработчиком *on( )*	Клип с обработчиком в среде разработки
MovieClip с обработчиком *onClipEvent( )*	Клип с обработчиком в среде разработки
Кнопка со свойством-обработчиком события	Кнопка, которая определяет свойство обработчика
Кнопка с обработчиком *on( )*	Клип, на временной диаграмме которого находится кнопка

Порядок выполнения onClipEvent() в стиле Flash 5

В некоторых фильмах код разбросан по нескольким временным диаграммам и обработчикам *on( )* и *onClipEvent( )*. Поэтому нередко один кадр требует выполнения нескольких отдельных блоков кода: одних в обработчиках событий, других в кадрах на временных диаграммах, а третьих – на главных временных диаграммах документов в плеере. В таких случаях порядок выполнения различных частей кода может стать довольно сложным и оказать значительное влияние на поведение программы. Чтобы избежать неприятных сюрпризов и обеспечить поведение кода, соответствующее замыслу разработчика, необходимо знать, в каком порядке выполняются обработчики событий в отношении различных временных диаграмм фильма.

Обработчики событий, вызываемых вводом данных пользователем, выполняются независимо от кода на временных диаграммах фильма. Например, обработчики событий кнопок выполняются немедленно, когда происходит событие, которое они обрабатывают. Это относится к обработчикам событий *onMouseDown( )*, *onMouseUp( )*, *onMouseMove( )*, *onKeyDown( )* и *onKeyUp( )*.

Однако обработчики событий *onClipEvent( )* выполняются в очередности, соответствующей ходу фильма, как показано в табл. 10.3.

Таблица 10.3. Порядок выполнения onClipEvent( ) в стиле Flash 5

Обработчик события	Момент выполнения
load	Выполняется в первом кадре, где клип присутствует на сцене, после выполнения кода родительской временной диаграммы, но до выполнения внутреннего кода клипа и вывода кадра. См. особые примечания к *MovieClip.onLoad( )* в «Справочнике по языку»
unload	Выполняется в первом кадре, где клип отсутствует на сцене, до выполнения кода родительской временной диаграммы
enterFrame	Выполняется во втором и всех последующих кадрах, где клип присутствует на сцене. Он выполняется до выполнения кода родительской временной диаграммы и до выполнения внутреннего кода клипа
data	Выполняется в любом кадре, где клип получил данные. При запуске выполняется до внутреннего кода клипа и до кода *enterFrame*

Результат действия правил табл. 10.3 проще рассмотреть на практическом примере. Допустим, есть фильм с одним слоем и четырьмя ключевыми кадрами на главной временной диаграмме. К каждому ключевому кадру прикрепим некоторый код. Затем создадим второй слой, в который поместим клип в кадрах с 1 по 3, но не в кадре 4. Добавим в клип обработчики *onClipEvent( )* событий *load*, *enterFrame* и *unload*. Наконец, внутри клипа создадим три ключевых кадра, каждый из которых также содержит блок кода. На рис. 10.2 показано, как выглядит фильм.

При воспроизведении фильма порядок выполнения кода будет таким:

```
========КАДР 1=======
1) Выполняется код главной временной диаграммы
2) Выполняется обработчик load
3) Выполняется внутренний код клипа в кадре 1

========КАДР 2=======
1) Выполняется обработчик enterFrame
2) Выполняется внутренний код клипа в кадре 2
3) Выполняется код главной временной диаграммы

========КАДР 3=======
1) Выполняется обработчик enterFrame
2) Выполняется внутренний код клипа в кадре 3
3) Выполняется код главной временной диаграммы
```

```
========КАДР 4=======
1) Выполняется обработчик unload
2) Выполняется код главной временной диаграммы
```

Очередность выполнения кода в нашем примере фильма демонстрирует некоторые важные практические правила, о которых следует помнить при написании обработчиков событий клипов:

- Код обработчика события *load* выполняется раньше, чем внутренний код клипа, поэтому обработчик *load* можно использовать для инициализации переменных для кадра 1 связанного с ним клипа, но только если обработчик создан в синтаксисе *onClipEvent( )*. Подробности представлены в описании *MovieClip.onLoad( )* в «Справочнике по языку».
- Перед созданием экземпляра клипа в кадре выполняется код этого кадра. В результате переменные и функции, определяемые пользователем в клипе, не станут доступны коду в его родительской временной диаграмме, пока на сцене не появится впервые кадр, *следующий* за клипом, даже если эти переменные и функции объявлены в обработчике *load* клипа. Решение этой проблемы есть в описании `#initclip` в «Справочнике по языку».
- Событие *enterFrame* не может произойти в одном кадре с событиями *load* или *unload*. События *load* и *unload* заменяют *enterFrame* в тех кадрах, где клип появляется на сцене и исчезает с нее.
- Во всех кадрах код обработчика события клипа *enterFrame* выполняется раньше кода родительской временной диаграммы клипа. Поэтому с помо-

Рис. 10.2. *Фильм для проверки очередности выполнения кода*

щью обработчика *enterFrame* можно изменить свойства родительской временной диаграммы клипа, а затем тотчас же использовать новые значения в коде этой временной диаграммы, причем все в одном кадре.

Копирование обработчиков событий клипа

Маленькое замечание, имеющее большие последствия: обработчики событий *onClipEvent( )* и *on( )* копируются, когда происходит копирование клипа с помощью функции *duplicateMovieClip( )*. Однако это не относится к свойствам-обработчикам, таким как *onEnterFrame( )*! Допустим, например, что на сцене находится клип с именем `square`, в котором определен обработчик *onClipEvent(load)*:

```
onClipEvent (load) {
  trace("movie loaded");
}
```

Что произойдет в результате копирования `square` для создания `square2`?

```
square.duplicateMovieClip("square2", 0);
```

Ответ: при копировании `square` обработчик *onClipEvent(load)* копируется в `square2`, поэтому появление `square2` влечет выполнение его обработчика *load*, который выводит в окно Output сообщение «movie loaded». Пользуясь автоматическим сохранением обработчиков, можно создавать хитрые рекурсивные функции для получения мощных результатов. Это показано в описании *MovieClip.onLoad( )* в «Справочнике по языку».

Допустим теперь, что на сцене находится клип с именем `circle`, которому мы назначим свойство-обработчик события *onEnterFrame( )*:

```
circle.onEnterFrame = function () {
  trace(this._name);
}
```

Что произойдет в результате копирования `circle` в `circle2`?

```
circle.duplicateMovieClip("circle2", 1);
```

Ответ: в окне Output появится только имя `circle`. Свойство-обработчик события *onEnterFrame( )* не копируется в `circle2`, поэтому имя `circle2` не появится в Output. Для того чтобы заставить каждый дубликат клипа `circle` определять свойство-обработчик события *onEnterFrame( )*, можно присвоить свойству значение в кадре временной диаграммы `circle`:

```
this.onEnterFrame = function () {
  trace(this._name);
}
```

Еще более выразителен такой прием: определить `circle` как подкласс *MovieClip* и определить обработчик *onEnterFrame( )* в прототипе конструктора `circle`. На этой теме мы подробнее остановимся в главе 14.

Обновление экрана с помощью updateAfterEvent()

Обработчики событий *MovieClip onMouseDown( )*, *onMouseUp( )*, *onMouseMove( )*, *onKeyDown( )* и *onKeyUp( )* выполняются немедленно, когда происходят соответствующие им события. Именно немедленно, даже если соответствующее событие происходит в промежутке между выводом кадров.

Благодаря такой срочности фильм может обладать высокой скоростью реакции, но ее легко потерять при ошибочном программировании. По умолчанию зрительные эффекты, создаваемые *onMouseDown( )*, *onMouseUp( )*, *onMouseMove( )*, *onKeyDown( )* и *onKeyUp( )*, не отображаются Flash-проигрывателем, пока не будет выведен следующий кадр. Чтобы увидеть, как это происходит, создайте фильм из одного кадра со скоростью 1 кадр в секунду. Создайте клип с именем `circle_mc` и прикрепите к кадру 1 главной временной диаграммы такой код:

```
circle_mc.onMouseDown = function () {
  this._x += 10;
}
```

После этого запустите фильм на воспроизведение и щелкайте мышью как можно быстрее. Вы увидите, что регистрируются все щелчки, но клип перемещается только один раз в секунду. Поэтому если между кадрами вы щелкнете 6 раз, то клип переместится вправо на 60 пикселов, когда будет выведен очередной кадр. Если щелкнуть 3 раза, то клип переместится на 30 пикселов. *onMouseDown( )* обладает кумулятивным свойством – каждое его выполнение «запоминается» между кадрами, но результаты отображаются только при показе каждого кадра. В некоторых типах интерактивных действий это может привести к поразительным результатам.

К счастью, можно заставить Flash немедленно воспроизводить все визуальные изменения, происходящие во время работы обработчика данных, вводимых пользователем, не ожидая, пока подойдет следующий кадр. Мы просто вызовем в обработчике события функцию *updateAfterEvent( )*, например:

```
circle_mc.onMouseDown = function () {
  this._x += 10;
  updateAfterEvent();
}
```

Функцию *updateAfterEvent( )* можно применять только с событиями *MovieClip*, *onMouseDown( )*, *onMouseUp( )*, *onMouseMove( )*, *onKeyDown( )*, *onKeyUp( )* и функцией *setInterval( )*. Часто это нужно для того, чтобы действия пользователя вызывали быстрые и плавные зрительные изменения. Ниже, в примере 10.3, мы обращаемся к функции *updateAfterEvent( )* для плавного вывода пользовательского указателя. Однако для событий кнопок не требуется явно вызывать функцию *updateAfterEvent( )*. События кнопок автоматически обновляют экран в промежутках между кадрами.

Приемники *onMouseDown( )*, *onMouseMove( )* и *onMouseUp( )* были добавлены в объект *Mouse* во Flash Player 6. Аналогично в объект *Key* были добавлены приемники *onKeyDown( )* и *onKeyUp( )*. Обычно приемники *Mouse* и *Key* использовать предпочтительней, чем аналогичные события *onClipEvent*. Однако во Flash Player 6 функцию *updateAfterEvent( )* можно применять только в событиях *MovieClip*. Поэтому, если требуется выполнять обновление экрана между кадрами, следует задействовать обработчики событий *MovieClip*. В следующие версии Player будет, вероятно, включена поддержка *updateAfterEvent( )* в приемниках *Mouse* и *Key*.

Повторное использование кода

При использовании обработчиков и приемников событий не забывайте о принципах централизации кода, рассмотренных в главе 9. Старайтесь избегать лишнего дублирования кода и распределения его по разным элементам фильма. Обнаружив, что вы помещаете один и тот же код в тело нескольких обработчиков событий, попробуйте определить его как одну функцию и адресовать к ней каждый обработчик события. Попытайтесь обобщить код, оторвав его от объекта и поместив в хранилище кода в каком-либо месте фильма; глобальный код, используемый во всех частях фильма, следует прикрепить к объекту `_global`.

Как правило, размещение инструкций внутри обработчиков *on( )* или *onClipEvent( )* нельзя признать удачным решением. Помните, что инкапсуляция кода в функции позволяет повторно использовать код и легко находить его. Особенно это касается кнопок, – лично я редко помещаю какой-либо код непосредственно в кнопку в среде Flash MX и редко использую более одной инструкции с вызовом функции в обработчиках *on( )* Flash 5. Имея дело с клипами, следует проявлять большую рассудительность, так как код, размещаемый непосредственно в клипах, часто составляет весомую часть хорошо сложенной независимой архитектуры кода. Попробуйте различные подходы, пока не найдете правильное соотношение между своими потребностями и уровнем мастерства. В любом случае полезно помнить о проблемах избыточности и повторного использования кода.

Динамические обработчики событий клипов

Ранее в этой главе мы видели, что обработчики событий *on( )* и *onClipEvent( )* в стиле Flash 5 нельзя изменять и удалять во время воспроизведения фильма. Кроме того, обработчики *onClipEvent( )* нельзя прикрепить к главной временной диаграмме (`_root`) какого-либо фильма.

Чтобы обойти эти ограничения, можно в случае *onClipEvent(enterFrame)* и событий клипа, связанных с мышью и клавиатурой, использовать экземпляры пустых клипов для моделирования динамического удаления или изменения обработчика событий. Пустые клипы даже позволяют моделировать обработчики *onClipEvent( )* для главной временной диаграммы. Чтобы

посмотреть, как это работает, выполните следующие действия (учтите, что эта технология относится только к Flash 5, поскольку во Flash MX свойства-обработчики событий можно удалить непосредственно с помощью *delete*):

1. Создать пустой клип с именем `process`.
2. Поместить в `process` другой пустой клип с именем `eventClip`.
3. В клипе `eventClip` установить желаемый обработчик события. Код в обработчике `eventClip` должен быть «нацелен» на временную диаграмму, содержащую клип `process`, например:

```
onClipEvent (mouseMove) {
  this._parent._parent.doSomeFunction();
}
```

4. Чтобы экспортировать `process` для использования с функцией *attachMovie( )*, выделите его в Library и выберите пункт меню Options → Linkage. Установите Linkage в Export This Symbol и назначьте подходящий идентификатор (например, *mouseMoveProcess*).
5. Наконец, чтобы включить обработчик события, прикрепите клип `process` к соответствующей временной диаграмме с помощью *attachMovie( )*.
6. Чтобы отключить обработчик, удалите клип `process` с помощью *removeMovieClip( )*.

Чтобы посмотреть на эту технологию в действии, загрузите «Event Loop, Controllable» (цикл события, управляемый) из сетевого хранилища кода.

Применение обработчиков событий

Завершим наше изучение событий и обработчиков событий ActionScript несколькими практическими примерами. Это простые приложения, но они позволяют почувствовать, насколько гибким может оказаться программирование на основе событий. Дополнительные примеры кода можно найти в «Справочнике по языку», а именно в описании событий, связанных с перечисленными в начале главы объектами.

В примере 10.2 клип с именем `box_mc` сжимается и расширяется.

Пример 10.2. Колебательное изменение размеров клипа

```
// Установить параметры колебаний размеров
box_mc.sizeIncrement = 10;
box_mc.maxHeight = 200;
box_mc.minHeight = 20;

// В каждом кадре устанавливать размеры клипа
box_mc.onEnterFrame = function () {
  if (this._height >= this.maxHeight || this._height <= this.minHeight) {
    this.sizeIncrement = -this.sizeIncrement;
  }
  this._height += this.sizeIncrement;
  this._width  += this.sizeIncrement;
}
```

Пример 10.3 моделирует пользовательский курсор мыши, пряча обычный системный курсор и заставляя клип следовать за положением мыши на экране. В этом примере обработчики *onMouseDown( )* и *onMouseUp( )* несколько изменяют размер пользовательского указателя, чтобы обозначить щелчок мыши. Код, помещенный в клип, действует как пользовательский курсор.

Пример 10.3. Пользовательский указатель мыши

```
Mouse.hide();

this.onMouseMove = function () {
  // При перемещении мыши поместить текущий клип
  // в точке с координатами курсора мыши.
  this._x = _root._xmouse;
  this._y = _root._ymouse;
  updateAfterEvent();  // Обновить экран для плавности перемещения.
}

this.onMouseDown = function () {
  // При щелчке мыши сжать клип.
  this._width  *= .5;
  this._height *= .5;
  updateAfterEvent();
}

this.onMouseUp = function () {
  // При щелчке мыши восстановить размер клипа.
  this._width  *= 2;
  this._height *= 2;
  updateAfterEvent();
}
```

В примере 10.4 на языке ActionScript создается клип с поведением кнопки. Клип динамически генерируется с помощью *MovieClip.createEmptyMovieClip( )*, затем в клипе вычерчивается квадрат с помощью методов рисования *MovieClip* и, наконец, создаются обработчики кнопок для состояний «включено», «выключено», «нажато» и «отжато». Дополнительные сведения об API для рисования представлены в разделе «Рисование в клипе на этапе исполнения» главы 13.

Пример 10.4. Клип-кнопка, создаваемый с нуля

```
// Создать клип
this.createEmptyMovieClip("optionButton_mc", 1);

// Нарисовать темно-красный квадрат со стороной 16 пикселов
// и с вершинами в точках (-8,-8) и (8,8)
optionButton_mc.lineStyle(undefined);
optionButton_mc.moveTo(-8,-8);
optionButton_mc.beginFill(0x990000);
optionButton_mc.lineTo(8, -8);
optionButton_mc.lineTo(8, 8);
optionButton_mc.lineTo(-8, 8);
optionButton_mc.lineTo(-8, -8);
```

```
optionButton_mc.endFill();

// При прохождении мыши над клипом сделать его ярко-красным
optionButton_mc.onRollOver = function () {
  var c = new Color(this);
  c.setRGB(0xFF0000);
}

// Убрать ярко-красный цвет при уходе мыши с клипа
optionButton_mc.onRollOut = function () {
  var c = new Color(this);
  c.setRGB(0x990000);
}

// Сжать клип на 20% при нажатии кнопки мыши
optionButton_mc.onPress = function () {
  this._xscale = this._yscale = 80;
}

// Восстановить первоначальные размеры клипа при освобождении кнопки
optionButton_mc.onRelease = function () {
  this._xscale = this._yscale = 100;
}
```

Вперед!

Освоив инструкции, операторы, а теперь и события вместе с обработчиками, мы научились работать со всеми внутренними инструментами ActionScript. Завершая изучение языка, в последующих трех главах мы рассмотрим три крайне важных типа данных: массивы, объекты и клипы.

11

Массивы

В главе 3 мы уже говорили о том, что элементарные типы данных – строки, числа, булевы значения, `null` и `undefined` – представляют в сценариях основную информацию. Было также отмечено, что ActionScript поддерживает несколько *составных* типов данных, которые могут объединять несколько фрагментов данных в один элемент данных.

Первым изучаемым составным типом данных станет *массив* (*array*). Массивы предназначены для хранения и работы с упорядоченными списками информации и потому представляют собой основополагающее средство в программировании. Они применяются для решения таких задач, как хранение значений, полученных из формы, заполненной пользователем, создание выпадающих меню, слежение за неприятельским космическим кораблем в игре и т. д. В простейшем виде массив можно представить как список элементов, например список необходимых покупок или записей в бухгалтерской книге.

Что такое массив?

Массив – это структура данных, которая может заключать в себе несколько отдельных значений данных, подобно тому как здание – физическая структура, которая может содержать несколько этажей. В отличие от элементарных типов данных, массив может содержать более одного значения данных. Вот простой пример, в котором показаны две строки по отдельности, а затем массив, содержащий две строки:

```
"oranges"              // Одно элементарное строковое значение
"apples"               // Еще одно элементарное строковое значение
["oranges", "apples"]  // Один массив, содержащий две строки
```

Массив представляет собой контейнер общего назначения. В нем может содержаться любое количество элементов, в том числе и элементы разных типов. Массив может даже содержать другие массивы. Вот простой пример, демонстрирующий массив, содержащий как строки, так и числа. В нем может быть представлен перечень покупок и количество единиц каждого товара, который вы собираетесь приобрести:

```
shoppingList = ["oranges", 6, "apples", 4, "bananas", 3]
```

Вот еще одна аналогия, позволяющая лучше почувствовать идею массива. Представьте себе комод с ящиками. В каждом отдельном ящике есть некоторое содержимое (носки, рубашки и т. д.). Но в самом комоде нет *содержимого*, только *ящики*. А уж ящики хранят содержимое. Комод группирует ящики в единое целое.

При работе с массивом (комодом) нас обычно интересуют значения, хранящиеся в массиве (содержимое ящиков). Но можно работать и с самой структурой хранения. Можно, например, изменить размер массива (добавить или вытащить ящики) или упорядочить значения (обменять содержимое ящиков).

Хотя мы и говорим, что массив содержит много значений, важно понять, что сам массив является одним элементом данных, так же как строка, содержащая несколько символов, остается одной строкой, а число из нескольких цифр считается одним числом. В качестве одиночного элемента данных массив можно присвоить переменной или задействовать его в более сложном выражении:

```
product = ["ladies downhill skis", 475]  // Сохранить массив в переменной
display(product);                        // Передать массив в функцию
```

Анатомия массива

Каждый отдельный предмет, хранящийся в массиве, называется *элементом* массива, и у каждого элемента есть уникальный номер (*индекс*), по которому на него можно ссылаться.

Элементы массива

Как и переменная, каждый элемент массива может хранить любой допустимый элемент данных. Поэтому массив в целом похож на совокупность последовательно пронумерованных переменных, но вместо того, чтобы давать всем элементам разные имена, каждому элементу присваивается номер (номер первого элемента 0, а не 1).

На рис. 11.1 схематически изображена структура массива из трех элементов. Элемент 0 содержит значение «Erica», элемент 1 – «Slavik», а элемент 2 – «Gary».

При работе со значениями, хранящимися в элементах массива, следует запрашивать их по номерам. В нашей аналогии с комодом можно попросить ActionScript сохранить что-то в первом ящике или извлечь то, что находится во втором.

Индексация элементов массивов

Положение элемента в массиве называют его *индексом*. Так же, как можно обратиться к седьмому символу строки, можно обратиться к седьмому элементу массива по его индексу (в данном случае индекс равен 6). Индекс элемента служит для присвоения элементу значения или извлечения значения,

Рис. 11.1. *Пример структуры массива*

или для какой-либо другой обработки элемента. Например, некоторые функции при работе с массивами с помощью индексов массивов задают диапазон элементов, подлежащих обработке.

Кроме того, можно вставлять элементы в начало, конец и даже середину массива и, само собой, удалять элементы. В массиве могут быть пустые промежутки (то есть некоторые элементы могут отсутствовать). Так, элементы могут присутствовать в позициях 0 и 4, но это не значит, что в позициях 1, 2 и 3 они должны быть обязательно. Массивы с пустыми промежутками называются *разреженными* (*sparse*).

Размер массива

В каждом массиве в любой момент его существования содержится определенное количество элементов. Количество элементов, которые могут быть размещены в массиве, называется его *длиной* (*length*); об этом поговорим позже.

Создание массивов

Создать новый массив можно с помощью литерала данных (то есть перечислив все элементы через запятую и заключив их в квадратные скобки) или с помощью встроенной функции – конструктора массивов *Array( )*.

Литералы массивов

Обычно создавать массивы с помощью литералов удобнее, чем с помощью конструктора *Array( )*. Напомним, что *литерал* – это непосредственное представление некоторых фиксированных данных. Например:

```
"beaver"      // Строковый литерал
234.2034      // Числовой литерал
true          // Булев литерал
```

В литерале массива начало и конец массива обозначаются квадратными скобками. Внутри квадратных скобок значения элементов массива задаются в виде списка выражений, разделяемых запятыми. Вот общий синтаксис:

```
[expression1, expression2, expression3]
```

Массивы в других языках программирования

Почти любой язык программирования высокого уровня поддерживает тот или иной тип массива или похожих на массив конструкций. При этом в способах реализации массивов в разных языках есть различия. Например, во многих языках не допускается хранение в массиве данных разных типов. Во многих языках массив может содержать числа или строки, но не те и другие в одном массиве. Интересно, что в языке C отсутствует элементарный тип данных *string*. Вместо этого в C поддерживается тип односимвольных данных с именем *char*, а строки считаются составным типом данных и реализованы в виде массива элементов типа *char*.

В ActionScript размер массива автоматически изменяется при добавлении или удалении элементов. Во многих языках размер массива должен быть указан при первом *объявлении* массива или задании его *размерности* (при этом для хранения данных массива *выделяется* память). В Lingo, языке сценариев для Macromedia Director, массивы называются *lists* (*списки*). Как и ActionScript, Lingo разрешает хранить в массивах данные разных типов и при необходимости автоматически изменяет размер массива. В отличие от ActionScript и C, в которых нумерация элементов массива начинается с нуля (*zero-relative*), в списке Lingo первый элемент имеет номер 1 (*one-relative*).

Языки различаются и по тому, что происходит при попытке доступа к элементу, находящемуся за границами массива. ActionScript и Lingo просто добавят элемент, если попытаться присвоить значение элементу вне существующих границ массива. Если попытаться получить значение элемента за пределами границ массива, ActionScript возвратит `undefined`, а Lingo сообщит об ошибке. Язык C не проверяет, является ли допустимым номер элемента, к которому происходит обращение. Он позволяет извлекать и устанавливать значения элементов за пределами массива, что обычно влечет изменение в памяти других данных или чтение бессмысленных данных, не принадлежащих массиву. Часто возможности, предоставляемые языком C, становятся веревкой, на которой можно повеситься. Этот язык, в отличие от ActionScript, даже позволяет задавать отрицательные индексы.

Выражения вычисляются, и полученные значения сохраняются в виде элементов определяемого массива. Могут использоваться любые допустимые выражения, в том числе вызовы функций, переменные, литералы и даже другие массивы (массив, находящийся внутри другого массива, называется *вложенным*, или *двумерным*).

Вот несколько примеров:

```
[4, 5, 63];                          // Простые числовые элементы
["jeremy", "janice", "eman"]         // Простые строковые элементы
[1, 4, 6 + 10]                       // Числовые выражения и операции
```

```
[firstName, lastName, "tall", "skinny"]  // Переменные и строки в качестве элементов
["month end days", [31, 30, 28]]         // Вложенный литерал массива
```

Для сравнения сделаем то же самое с помощью конструктора *Array( )*:

```
new Array(4, 5, 63)
new Array("jeremy", "janice", "eman")
new Array(1, 4, 6 + 10)
new Array(firstName, lastName, "tall", "skinny")
new Array("month end days", new Array(31, 30, 28))
```

Как показано в этом примере и отмечено во врезке «Массивы в других языках программирования», в ActionScript элементы одного массива могут содержать данные разных типов.

Конструктор массивов

Для создания массива с помощью конструктора *Array( )* применяется оператор *new*, за которым следуют ключевое слово `Array` и круглые скобки, в результате чего получается пустой (то есть не содержащий элементов) массив. Как мы увидим в главе 12, функция-конструктор создает новый объект данных, в нашем случае экземпляр *Array*. Обычно вновь созданный массив присваивается переменной или другому контейнеру данных, которые позволят обращаться к массиву в будущем. Например:

```
var myList = new Array();// Сохранить пустой массив в переменной myList
```

Часто требуется присвоить элементам массива начальные значения. Сделать это можно путем передачи параметров конструктору *Array( )* при его вызове. В зависимости от передаваемых параметров вызов конструктора приводит к различным результатам.

При передаче конструктору *Array( )* более одного аргумента или одного нечислового аргумента каждый аргумент становится одним из значений элементов нового массива, например:

```
var frameLabels = new Array("intro", "section1", "section2", "home");
```

Массив, хранящийся в переменной `frameLabels`, будет содержать следующие элементы:

```
0: "intro"
1: "section1"
2: "section2"
3: "home"
```

При передаче конструктору *Array( )* только одного числового аргумента создается массив с указанным количеством пустых элементов-заместителей:

```
var myList = new Array(14); // Создает массив из 14 пустых элементов
```

Аргументы, передаваемые конструктору *Array( )*, могут быть любыми допустимыми выражениями, в том числе составными:

```
var x = 10;
```

```
var y = 5;
var myNumbers = new Array(x + 1, x * y, Math.random());
```

В результате переменная `myNumbers` будет хранить массив с такими элементами:

```
0: 11
1: 50
2: число с плавающей точкой между 0 до 1
```

Мы упомянули здесь конструктор *Array( )*, поскольку он может создавать массив с необходимым количеством свободных мест под элементы, который труднее организовать с помощью литерала массива. Более того, описание конструктора *Array( )* служит вступительным словом к описанию функций конструкторов. На практике массивы обычно создаются с помощью литералов массивов или импортируются из внешних источников. Чаще всего конструктор *Array( )* применяется для создания пустого массива (возможно, с заданным количеством свободных мест под элементы).

Обращение к элементам массива

Когда массив создан, неизбежно потребуется извлечь или изменить значения его элементов. Для этого можно воспользоваться *оператором доступа к массиву* (то есть квадратными скобками `[]`), с которым мы познакомились в главе 5.

Извлечение значения элемента

Для обращения к отдельному элементу надо указать идентификатор массива и индекс элемента, заключенный в квадратные скобки, например:

```
arrayName[elementNumber]
```

где *arrayName* обычно представляет собой переменную, указывающую на массив, а *elementNumber* – целое число, задающее индекс элемента. Номер первого элемента равен 0, а номер последнего на единицу меньше длины массива. Если задать номер элемента, превышающий номер последнего действительного элемента, интерпретатор возвратит значение `undefined`. Например:

```
// Создать массив с помощью литерала массива и сохранить его в trees
var trees = ["birch", "maple", "oak", "cedar"];

// Вывести первый элемент trees в окно Output
trace(trees[0]);  // Displays: "birch"

// Присвоить значение третьего элемента переменной favoriteTree
// (индексы начинаются с 0, поэтому индекс 2 задает третий элемент!)
var favoriteTree = trees[2];  // favoriteTree получает значение "oak"
```

Теперь интересная особенность. Так как индекс элемента можно задать в виде любого выражения, дающего число, то в качестве индекса можно использовать переменную или сложное выражение точно так же, как это делается с помощью числа, например:

```
var i = 3;
```

```
var lastTree = trees[i];  // Присвоить lastTree значение "cedar"
```

Можно даже использовать в этом качестве выражения, содержащие вызовы функций, которые возвращают числовые значения:

```
// Присвоить randomTree значение случайно выбранного элемента trees,
// получив случайное число между 0 и 3
var randomTree = trees[Math.floor(Math.random() * 4)];
```

Сила! Аналогичный подход можно применить для выбора случайного вопроса из массива простых вопросов или случайной карты из массива, представляющего колоду.

Обратите внимание на сходство между доступом к элементу массива и доступом к значению переменной. Элементы массива можно включать в состав сложных выражений, например так:

```
var myNums = [12, 4, 155, 90];
var myTotal = myNums[0] + myNums[1] + myNums[2] + myNums[3];  // Сложить элементы
                                                              // массива
```

Суммирование значений элементов массива вручную не служит эталоном оптимального кода. Ниже будет показан значительно более удобный способ последовательного доступа к элементам массива.

Присваивание элементу значения

В качестве левого операнда в выражении присваивания значения элементу массива выступает *arrayName*[*elementNumber*]:

```
// Создать массив
var cities = ["Toronto", "Montreal", "Vancouver", "Waterloo"];
// Теперь в массиве cities: ["Toronto", "Montreal", "Vancouver", "Waterloo"]

// Присвоить значение первому элементу массива
// cities теперь выглядит так: ["London", "Montreal", "Vancouver", "Waterloo"]
cities[0] = "London";

// Присвоить значение четвертому элементу массива
// cities теперь выглядит так: ["London", "Montreal", "Vancouver", "Hamburg"]
cities[3] = "Hamburg";

// Присвоить значение третьему элементу массива
// массив cities: ["London", "Montreal", 293.3, "Hamburg"]
cities[2] = 293.3;  // Обратите внимание, что можно изменять тип данных
```

Обратите внимание, что для присваивания значения элементу массива можно использовать в качестве индекса любое неотрицательное числовое выражение:

```
var i = 1;
// Установить значение элемента с номером i
// cities теперь выглядит так: ["London", "Prague", 293.3, "Hamburg"]
cities[i] = "Prague";
```

Определение размера массива

У всех массивов есть встроенное свойство `length`, указывающее на текущее количество элементов (включая неопределенные). Для доступа к свойству `length` массива применяется оператор «точка»:

```
arrayName.length
```

Вот несколько примеров:

```
myList = [34, 45, 57];
trace(myList.length);          // Выводит: 3
myWords = ["this", "that", "the other"];
trace(myWords.length);         // Выводит 3, количество элементов,
                               // а не количество слов или символов
frameLabels = new Array(24);   // Обратите внимание на один числовой аргумент,
                               // передаваемый в конструктор Array()
trace(frameLabels.length);     // Выводит: 24
```

Длина массива всегда на единицу больше индекса его последнего элемента. Например, массив, состоящий из элементов с индексами 0, 1 и 2, имеет длину 3. Массив, состоящий из элементов с индексами 0, 1, 2 и 50, имеет длину 51. 51? Да, 51. Несмотря на то что индексы с 3 по 49 пусты, они учитываются в длине массива. Последним элементом массива всегда является *myArray*[*myArray*.length - 1], так как нумерация индексов начинается с 0, а не с 1.

При добавлении или удалении элементов свойство массива `length` обновляется, отражая сделанные изменения. На самом деле можно даже присваивать значение свойству `length`, тем самым добавляя или удаляя элементы в конце массива. Здесь видно отличие от свойства `length` строки, доступного только для чтения (об этом говорилось в главе 4). При уменьшении длины массива уничтожаются элементы, выходящие за пределы нового значения `length`. Свойство `length` относится только к элементам с номерами, а не к элементам с именами, как описывается в следующем разделе «Именованные элементы массива».

С помощью свойства `length` можно создать цикл для перебора всех элементов массива, подобный циклу, показанному в примере 8.1. Перебор элементов массива является базовой задачей, возникающей в программировании. Чтобы почувствовать возможности, имеющиеся при комбинировании циклов и массивов, изучите пример 11.1, где организован поиск элемента со значением «hip hop» в массиве `soundtracks`. Цикл *for* должен быть вам знаком по главе 8, а оператор инкрементирования – по главе 5. Что касается написания кода для доступа к массиву, то вы только что узнали, как это делается.

Пример 11.1. Поиск в массиве

```
// Создать массив
var soundtracks = ["electronic", "hip hop", "pop", "alternative", "classical"];

// Проверить каждый элемент, не содержит ли он "hip hop"
for (var i = 0; i < soundtracks.length; i++) {
  trace("Now examining element: " + i);
```

```
    if (soundtracks[i] == "hip hop") {
      trace("The location of 'hip hop' is index: " + i);
      break;
    }
  }
```

Разработаем на основе примера 11.1 обобщенную функцию поиска, которая может искать в *любом* массиве *любой* элемент. Она приведена в примере 11.2. Эта функция возвращает позицию в массиве, где был найден элемент, или `null`, если он не был найден.

Пример 11.2. Обобщенная функция поиска в массиве

```
function searchArray (whichArray, searchElement) {

  // Проверить каждый элемент, не содержит ли он searchElement
  for (var i = 0; i < whichArray.length; i++) {
    if (whichArray[i] == searchElement) {
      return i;
    }
  }
  return null;
}
```

Вот как с помощью этой новой функции можно проверить, есть ли имя «Fritz» в массиве `userNames` разрешенных имен пользователей:

```
if (searchArray (userNames, "Fritz") == null) {
  trace ("Извините, такого пользователя нет");
} else {
  trace ("Добро пожаловать в игру.");
}
```

Это радует! Наступил один из тех моментов, когда вознаграждаются титанические усилия, – мы собрали все воедино. Сочетание тривиальных на первый взгляд отдельных операций может приводить к образованию чрезвычайно мощных и гибких программ. Из строительных элементов, находящихся в вашем распоряжении, можно создавать любые мыслимые конструкции, как из букв алфавита или оснований нуклеиновых кислот в цепочке ДНК. В оставшейся части главы более подробно рассказывается о технике работы с массивами, в том числе о применении встроенных функций, уже готовых к тому, чтобы вместо вас выполнять часто встречающиеся задачи. Если задачу нельзя решить с помощью встроенных функций, напишите собственную, такую, например, как функция *searchArray( )* из предыдущего примера.

Именованные элементы массива

Нумерованные элементы массива обычно одинаковы по своей природе. Например, в массиве имен пользователей каждый элемент хранит имя отдельного пользователя. Но иногда желательно хранить в массиве некоторые взаимосвязанные элементы информации, причем не обязательно эти элементы имеют один тип. Например, в массиве, описывающем компьютер, один эле-

мент может содержать тактовую частоту процессора, другой – объем оперативной памяти, а третий – название операционной системы. Элементы связаны между собой, потому что описывают одну и ту же машину, но сами по себе они разнородны. Поэтому описание элементов как имеющих номера 0, 1 и 2 будет точным, но не слишком содержательным. Обычно элементы нумеруют, но, к счастью, их можно также именовать. Это позволяет обращаться к элементам массива с помощью осмысленных меток, таких как «processorSpeed», «ramInMB» или «osName», а не просто целых чисел. В результате не только облегчается чтение кода, но и появляется возможность обращаться к элементам массивов, не зная их индекса. В Lingo массивы, поддерживающие именованные элементы, называются *списками свойств* (*property lists*). Во многих языках их называют *хэшами* (*hashes*). В Perl они называются *ассоциативными массивами* (*associative arrays*).

В отличие от некоторых языков, допускающих обращение к элементам массивов как по имени, так и по номеру, в ActionScript нельзя обращаться к именованному элементу по номеру. Кроме того, именованные элементы массива нельзя обрабатывать методами Array (*push( )*, *pop( )* и т. д., о которых будет рассказано ниже в этой главе). Именованные элементы не входят в состав списка нумерованных элементов массива. Например, массив из одного именованного элемента и двух нумерованных имеет длину 2, а не 3. Поэтому для перебора всех именованных элементов массива ActionScript следует применять цикл *for-in* (о котором рассказывалось в главе 8), перечисляющий как именованные, так и нумерованные элементы.

Именованные элементы массива представляют собой просто свойства объекта. В ActionScript общий объект может с тем же успехом хранить группу взаимосвязанных свойств, как массив – группу именованных элементов. Распространены оба подхода.

Создание именованных элементов массива и обращение к ним

Для добавления в массив элемента, который потом можно будет извлекать по имени, применяются уже знакомые вам квадратные скобки, содержащие строку вместо индекса элемента:

```
arrayName[elementName] = expression
```

Здесь *elementName* – это строка. Например:

```
var importantDates = new Array();                // Создать пустой массив
importantDates["dadsBirthday"] = "June 1";";     // Добавить именованный элемент
importantDates["mumsBirthday"] = "January 16";   // Добавить еще один
                                                 // именованный элемент
```

Прежде чем добавлять в массив элементы, не забудьте создать его с помощью конструктора *new Array( )*. Для указания именованного элемента массива можно также воспользоваться оператором «точка», например:

```
arrayName.elementName = expression
```

В этом случае *elementName* представляет собой идентификатор, а *не* строку. Например:

```
var importantDates = new Array();
importantDates.dadsBirthday = "June 1";
importantDates.mumsBirthday = "January 16";
```

Предположим, что идентификатор элемента известен (например, dadsBirthday в массиве importantDates), тогда к нему можно обратиться одним из следующих двух способов:

```
var goShopping = importantDates["dadsBirthday"];
var goShopping = importantDates.dadsBirthday;
```

Как в случае присваивания значения именованному элементу, так и в случае чтения его значения, в синтаксисе с квадратными скобками *elementName* представляет собой строку или выражение, возвращающее строку. Аналогично, в синтаксисе с *точкой* *elementName* – это *идентификатор* (то есть имя элемента без кавычек), а не строка. Обратите внимание, что в случае применения квадратных скобок не требуется выдерживать соответствие *elementName* правилам формирования допустимых идентификаторов. Вот пример идентификатора *elementName*, не являющегося допустимым идентификатором, но вполне допустимого в операторе []:

```
importantDates["dad's Birthday"] = "June 1";
```

Эквивалентное выражение с точкой генерирует ошибку компиляции:

```
importantDates.dad's Birthday = "June 1";  // ОШИБКА!
```

Есть хорошее правило – старайтесь не употреблять недопустимые идентификаторы в качестве имен элементов, несмотря на гибкость оператора [].

Удаление именованных элементов

Чтобы избавить массив от ненужного именованного элемента, применяется оператор *delete*, который упоминался в главе 5:

```
delete arrayName.elementName
```

Удаление именованного элемента разрушает как значение элемента, так и его контейнер, освобождая при этом память, занимаемую элементом и его содержимым. (А оператор *delete* при удалении *нумерованных* элементов лишь сбрасывает значение, оставляя контейнер нетронутым.)

Добавление элементов в массив

Добавить элементы в массив можно путем задания значения для нового элемента, увеличения значения свойства length или с помощью одной из встроенных функций для работы с массивами.

Непосредственное добавление новых элементов

Добавить в существующий массив новый элемент по заданному индексу можно простым присваиванием значения этому элементу:

```
// Создать массив и присвоить ему три значения
var myList = ["apples", "oranges", "pears"];

// Добавить четвертое значение
myList[3] = "tangerines";
```

Новый элемент не обязательно помещать сразу за последним элементом прежнего массива. Если поместить новый элемент дальше, чем на один элемент после конца массива, ActionScript автоматически создаст неопределенные элементы для промежуточных индексов:

```
// Оставить пустыми индексы с 4 по 38
myList[39] = "grapes";

trace (myList[12]); // Выводит: undefined
```

Если элемент уже существует, он будет заменен новым значением. Если элемент не существует, он будет создан.

Добавление новых элементов с помощью свойства length

Для того чтобы расширить массив, не присваивая значений новым элементам, достаточно увеличить свойство `length`, и ActionScript добавит нужное количество элементов для достижения указанной длины:

```
// Создать массив из трех элементов
var myColors = ["green", "red", "blue"];
// Добавить в массив 47 пустых элементов с номерами от 3 до 49
myColors.length = 50;
```

С помощью такого подхода можно создать ряд пустых элементов, чтобы сохранить данные, которые вы рассчитываете получить, например результаты сдачи экзаменов студентами.

Добавление элементов с помощью метода класса Array

Встроенные методы массивов позволяют осуществлять более сложные операции. (В главе 12 разъясняется, что *метод* – это функция, выполняющая действия с объектом.)

Метод push()

Метод *push( )* добавляет один или более элементов в конец массива. Он автоматически дописывает данные вслед за последним нумерованным элементом массива, поэтому нет нужды беспокоиться о реальном количестве существующих элементов. Кроме того, метод *push( )* может одновременно добавлять в массив несколько элементов. Для вызова метода *push( )* с массивом

указывается имя массива, затем оператор «точка», ключевое слово *push* и ноль или более параметров, заключенных в круглые скобки:

```
arrayName.push(item1, item2,...itemn);
```

Здесь *item1*, *item2*,...*itemn* – разделяемый запятыми список элементов, добавляемых в конец массива в качестве новых элементов. Приведем примеры:

```
// Создать массив из двух элементов
var menuItems = ["home", "quit"];

// Добавить один элемент
// menuItems becomes ["home", "quit", "products"]
menuItems.push("products");

// Добавить еще два элемента
// menuItems становится ["home", "quit", "products", "services", "contact"]
menuItems.push("services", "contact");
```

Метод *push( )* возвращает новую длину изменившегося массива (то есть значение свойства `length`, не учитывающего именованные элементы):

```
var myList = [12, 23, 98];
trace(myList.push(28, 36));  // Добавляет в myList 28 и 36 и выводит 5
```

Элементы, добавляемые в список, могут быть любыми выражениями. Перед добавлением элементов выражения вычисляются:

```
var temperature = 22;
var sky = "sunny";
var weatherListing = new Array();
weatherListing.push(temperature, sky);
trace (weatherListing); // Выводит: "22,sunny", а не "temperature,sky"
```

Метод unshift()

Метод *unshift( )* сходен с *push( )*, но один или несколько элементов добавляются в *начало* массива, проталкивая дальше все существующие элементы (то есть индексы существующих элементов увеличиваются, чтобы предоставить место новым элементам в начале массива). В синтаксисе метода *unshift( )* соблюдается тот же стиль, что и во всех методах массивов:

```
arrayName.unshift(item1, item2,...itemn);
```

Здесь *item1*, *item2*,...*itemn* составляют разделяемый запятыми список элементов, добавляемых в начало массива в качестве новых элементов. Если добавляется несколько элементов, они помещаются в массив в том порядке, в котором указаны. Вот некоторые примеры:

```
var flashVersions = new Array();
flashVersions[0] = 6;
flashVersions.unshift(5);        // flashVersions теперь [5, 6]
flashVersions.unshift(2,3,4);    // flashVersions теперь [2, 3, 4, 5, 6]
```

Метод *unshift( )*, как и *push( )*, возвращает новую длину увеличившегося массива.

Push, pop и стеки

Имя метода *push( )* берет начало от понятия программирования, называемого *стеком*. Стек можно представить себе в виде вертикального массива, подобного стопке тарелок. Тем, кто бывает в кафетериях и ресторанах с буфетом, должны быть знакомы подпружиненные стойки для тарелок. Новые чистые тарелки буквально *проталкиваются* (*pushed*) на самый верх стопки (stack), а старые тарелки проваливаются глубже в стойку. Тарелка с вершины стопки, которую клиент *выталкивает* (*pops*), была помещена в стопку последней. Такой стек называют «последним пришел – первым ушел» (last-in-first-out, LIFO) и обычно применяют, например, при организации буферов команд. Например, если в броузере щелкнуть по кнопке «Back», он переместит вас на предыдущую страницу. Если снова щелкнуть по этой кнопке, вы переместитесь назад еще на одну страницу и т. д. Это осуществляется путем добавления (*pushing*) URL каждой посещаемой страницы на вершину стека и выталкивания (*popping*) ее из стека при щелчке по кнопке «Back».

Стеки LIFO встречаются и в реальной жизни. Тот, кто последним сдает свой багаж в самолет, обычно первым получает его после приземления, поскольку очередность разгрузки багажа обратна очередности его погрузки. Тот «пострел», который успел сдать свой багаж первым, обречен дольше всех ждать его на конвейере после посадки самолета. Стек типа «первым пришел – первым ушел» (first-in-first-out, FIFO) обеспечивает большее равноправие. Он сначала обслуживает того, кто пришел первым. Стек FIFO подобен очереди, например в банке. Вместо того чтобы взять последний элемент массива, стек FIFO в качестве очередного элемента обрабатывает первый элемент массива. Затем он удаляет первый элемент, а все остальные элементы «сдвигаются вверх», так же как вы передвигаетесь вперед в очереди, когда удаляется тот, кто стоял перед вами (уходит, потому что его обслужили или он просто устал ждать). Поэтому слово *push* обычно подразумевает работу со стеком типа LIFO, тогда как *append* означает стек типа FIFO. В любом случае элементы добавляются в «конец» стека. Разница в том, с какого конца массива берется элемент для очередной операции.

Метод splice()

Метод *splice( )* может добавлять элементы в массив или удалять их из него. Обычно он применяется для вставки элементов в середину массива (последующие элементы перенумеровываются, чтобы освободить пространство) или удаления элементов из середины массива (последующие элементы перенумеровываются, чтобы ликвидировать разрыв). Если *splice( )* используется для выполнения обеих задач за один вызов, то в результате некоторые элементы заменяются новыми (хотя количество удаляемых и вставляемых элементов не обязательно должно совпадать). Вот синтаксис метода *splice( )*:

```
arrayName.splice(startIndex, deleteCount, item1, item2,...itemn)
```

Здесь *startIndex* – это число, задающее индекс, с которого должны начаться удаление элементов и необязательная их вставка (помните, что индекс первого элемента равен 0); *deleteCount* – необязательный аргумент, определяющий количество удаляемых элементов (включая элемент с индексом *startIndex*). Если *deleteCount* опущен, удаляются все элементы, начиная со *startIndex* включительно. Необязательные параметры *item1*, *item2*,...*itemn* представляют элементы, добавляемые в качестве элементов массива, начиная с номера *startIndex*.

Пример 11.3 демонстрирует многосторонность метода *splice( )*.

Пример 11.3. Использование метода массива splice( )

```
// Создать массив...
months = new Array("January", "Friday", "April", "May", "Sunday", "Monday", "July");
// Н-да. Что-то не так с нашим массивом. Надо его исправить.
// Сначала избавимся от "Friday"
months.splice(1,1);
  // Теперь months:
  // ["January", "April", "May", "Sunday", "Monday", "July"]

// Теперь добавим два месяца перед "April".
// Обратите внимание, что ничего не будет удалено (deleteCount равен 0).
months.splice(1, 0, "February", "March");
  // Теперь months: ["January", "February", "March", "April", "May",
  // "Sunday", "Monday", "July"]

// Наконец, удалим "Sunday" и "Monday", вставив вместо них "June"
months.splice(5, 2, "june");
  // Теперь months:
  // ["January", "February", "March", "April", "May", "June", "July"]

// Теперь, когда массив months исправлен, обрежем его так, чтобы в нем
// содержались месяцы только первого квартала года, удалив все элементы
// с индексами, большими 3 (то есть начиная с "April")
months.splice(3); // Теперь months: ["January", "February", "March"]
```

Другая полезная особенность *splice( )* состоит в том, что он возвращает массив тех элементов, которые *удалил*. Поэтому его можно применять для извлечения последовательности элементов из массива:

```
myList = ["a", "b", "c", "d"];
trace(myList.splice(1, 2));  // Выводит: "b, c"
                             // myList теперь ["a", "d"]
```

Если ни один элемент не удален, *splice( )* возвращает пустой массив (то есть массив без элементов).

Метод concat()

Подобно *push( )*, метод *concat( )* добавляет элементы в конец массива. В отличие от *push( )*, метод *concat( )* не изменяет массив, к которому был применен, а возвращает новый массив. Кроме того, он может объединять два или более массивов в один новый массив. Вот синтаксис этого метода:

```
origArray.concat(elementList)
```

Элементы, содержащиеся в *elementList*, добавляются по одному в конец массива *origArray*, а результат возвращается как *новый* массив, при этом *origArray* остается в неприкосновенности. Обычно возвращаемый массив сохраняется в переменной. Ниже в качестве элементов, добавляемых в массив, выступают обычные числа:

```
var list1 = new Array(11, 12, 13);
var list2 = list1.concat(14, 15);  // list2 стал [11, 12, 13, 14, 15]
```

В следующем примере *concat( )* применяется для объединения двух массивов:

```
var guests = ["Panda", "Dave"];
var registeredPlayers = ["Gray", "Doomtrooper", "TRK9"];
var allUsers = registeredPlayers.concat(guests);
// allUsers теперь: ["Gray", "Doomtrooper", "TRK9", "Panda", "Dave"]
```

Обратите внимание, что *concat( )* извлек элементы из массива `guests`, чего не стал бы делать *push( )*. Если выполнить такой код:

```
var allUsers = registeredPlayers.push(guests);
```

то получится следующий вложенный массив:

```
["Gray", "Doomtrooper", "TRK9", ["Panda", "Dave"]]
```

Кроме того, *push( )* изменил бы массив `registeredPlayers`, чего *concat( )* не делает.

Отметим, однако, что *concat( )* не разбивает на элементы *вложенные* массивы (элементы, сами являющиеся массивами внутри главного массива), что можно видеть из следующего кода:

```
var x = [1, 2, 3];
var y = [[5, 6], [7, 8]];
var z = x.concat(y);  // Результат [1, 2, 3, [5, 6], [7, 8]].
                      // Элементы y 0 и 1 не были "выровнены".
```

Удаление элементов из массива

Удалить из массива элементы можно так: применить оператор *delete*, уменьшить свойство массива `length` или задействовать один из встроенных методов массивов.

Удаление нумерованных элементов с помощью оператора delete

Оператор *delete* присваивает элементу массива значение `undefined` с соблюдением следующего синтаксиса:

```
delete arrayName[index]
```

Здесь *arrayName* — это любой массив, а *index* представляет собой номер или имя элемента, значение которого требуется установить в `undefined`. Честно говоря, в случае с нумерованными, а не именованными элементами название оператора *delete* (*удалить*) вводит в заблуждение. Метод *не* удаляет элемент из массива, а только устанавливает его значение в `undefined`. Поэтому операция *delete* совпадает по результату с присваиванием элементу значения `undefined`. В этом можно убедиться, проверив свойство массива `length` после удаления одного из элементов массива:

```
var myList = ["a", "b", "c"];
trace(myList.length);  // Выводит: 3
delete myList[2];
trace(myList.length);  // Снова выводит 3...элемент с индексом 2 имеет
                       // значение undefined, а не "c", но существует
```

Для того чтобы действительно удалить элементы, следует использовать метод *splice( )* (для удаления из середины массива) или методы *shift( )* и *pop( )* (для удаления из начала или конца массива). Обратите внимание, что *delete* по-разному ведет себя с именованными и нумерованными элементами. Применение *delete* к именованным элементам разрушает их навсегда, не оставляя никаких следов. Как мы видели в главе 5, наряду с именованными элементами массивов метод *delete* может уничтожать свойства объектов (на самом деле именованные элементы являются свойствами объекта *Array*).

Удаление элементов с помощью свойства length

Ранее свойство `length` применялось для добавления элементов в массив. Можно также присвоить свойству массива `length` значение, меньшее текущей длины, чтобы удалить элементы из массива (то естьвыполнить усечение массива):

```
var toppings = ["pepperoni", "tomatoes", "cheese", "green pepper", "broccoli"];
toppings.length = 3;
trace(toppings);  // Выводит: "pepperoni,tomatoes,cheese"
                  // Были отрезаны элементы 3 и 4 (два последних)
```

Удаление элементов с помощью методов класса Array

Есть несколько встроенных методов удаления элементов массива. Уже было показано, как с помощью метода *splice( )* удалить последовательность элементов из середины массива. Методы *pop( )* и *shift( )* применяются для удаления элементов из конца или из начала массива.

Метод pop()

Метод *pop( )* противоположен методу *push( )*: он удаляет последний элемент массива. Синтаксис *pop( )* прост:

```
arrayName.pop()
```

Не знаю почему, но мне всегда кажется, что «выталкивание» из массива выглядит забавно. Тем не менее, *pop( )* уменьшает длину массива на 1 и возвращает удаленный элемент. Например:

```
x = [56, 57, 58];
trace(x.pop());  // Выводит: 58 (значение взятого со стека элемента)
                 // x теперь [56, 57]
```

Как мы уже видели, *pop( )* часто применяется в сочетании с *push( )* в операциях со стеком LIFO. В примере 11.4 мы создаем массив `siteHistory` для слежения за перемещениями пользователя по сайту. Когда пользователь переходит в новый кадр, его местонахождение добавляется в массив с помощью *push( )*. При перемещении обратно из `siteHistory` его предыдущее местонахождение выталкивается с помощью *pop( )*, и пользователь направляется туда. Пример 11.4 можно загрузить из сетевого хранилища кода.

Пример 11.4. Кнопка Back с буфером памяти

```
// КОД В КАДРЕ 1
stop();
var siteHistory = new Array();

function goto(theLabel) {
  // Если мы еще не в требуемом кадре...
  if (theLabel != siteHistory[siteHistory.length - 1]) {
    // ...добавляем запрос в буфер и переходим в нужный кадр
    siteHistory.push(theLabel);
    this.gotoAndStop(siteHistory[siteHistory.length - 1]);
  }
  trace(siteHistory);
}

function goBack() {
  // Удаляем последний элемент в буфере
  siteHistory.pop();
  // Если в буфере еще что-то осталось...
  if (siteHistory.length > 0) {
    // ...переходим в самый последний кадр
    this.gotoAndStop(siteHistory[siteHistory.length - 1]);
  } else {
    // ...иначе идем домой
    this.gotoAndStop("home");
  }
  trace(siteHistory);
}
```

```
// КНОПКА НАВИГАЦИИ
gallery_btn.onRelease = function () {
  this._parent.goto("gallery");
}

// КНОПКА BACK
back_btn.onRelease = function () {
  this._parent.goBack();
}
```

Метод shift()

Помните метод *unshift( )*, применявшийся для добавления элемента в начало массива? Познакомьтесь с его alter ego, методом *shift( )*, удаляющим элемент из начала массива:

```
arrayName.shift()
```

Не так смешно, как *pop*. Ну да ладно.

Как и *pop( )*, *shift( )* возвращает значение удаленного им элемента. Оставшиеся элементы перемещаются вверх в начало массива с сохранением порядка. Например:

```
var sports = ["hackey sack", "snowboarding", "inline skating"];
trace(sports.shift());  // Выводит: hackey sack
                        // sports теперь ["snowboarding", "inline skating"]
trace(sports.shift());  // Выводит: snowboarding
                        // sports теперь ["inline skating"]
```

Метод *shift( )* действительно удаляет один элемент, поэтому его полезнее применять для удаления первого элемента массива, чем *delete*. Можно также применять *shift( )* для ограничения количества элементов в списке. Допустим, например, что мы рассчитываем частоту смены кадров фильма. После вывода каждого кадра мы отправляем текущее время в массив с помощью *push( )*. Чтобы ограничить размер массива последними 10 отсчетами времени, при необходимости выполняем *shift( )* для самого старого отсчета. Чтобы найти скорость воспроизведения фильма, усредняем значения, хранящиеся в массиве. Этот способ продемонстрирован в примере 11.5.

Пример 11.5. Расчет скорости воспроизведения фильма

```
// Создать клип для слежения за частотой кадров
this.createEmptyMovieClip("fpsTracker", 1);

// Создать массив для измерения времени в клипе fpsTracker
fpsTracker.elapsedTime = new Array();

// С помощью обработчика события onEnterFrame() измерять время после каждого кадра
fpsTracker.onEnterFrame = function () {
  // Добавить текущее время в elapsedTime
  this.elapsedTime.push(getTimer());

  // Если выборка достаточна для вычисления среднего...
  if (this.elapsedTime.length > 10) {
    // ...удалить самые старые значения из elapsedTime
```

```
        this.elapsedTime.shift();

        // Вычислить среднее количество миллисекунд на кадр
        this.elapsedAverage = (this.elapsedTime[this.elapsedTime.length - 1] -
                               this.elapsedTime[0]) / this.elapsedTime.length;

        // Чтобы найти fps, разделить 1 секунду на среднюю продолжительность
        this.fps = 1000 / this.elapsedAverage;
        trace("Current fps " + this.fps);
    }
}
```

Метод splice()

Ранее отмечалось, что метод *splice( )* может как удалять элементы из массива, так и добавлять их. Этот метод уже был подробно рассмотрен, поэтому не будем здесь возвращаться к его изучению. Однако в данном контексте особо продемонстрируем его возможности по удалению элементов:

```
var x = ["a", "b", "c", "d", "e", "f"];
x.splice(1,3);  // Удаляет элементы 1, 2 и 3, сохраняя ["a", "e", "f"]
x.splice(1);    // Удаляет элементы с 1 и до конца, оставляя ["a"]
```

Общие средства обработки массивов

Теперь мы знаем, как применять методы массивов для удаления элементов из массивов и добавления элементов в массивы. ActionScript предлагает также встроенные методы для упорядочения и сортировки элементов, преобразования элементов массивов в строки и извлечения массивов из других массивов.

Метод reverse()

Метод *reverse( )*, в соответствии со своим названием, изменяет порядок элементов в массиве на противоположный. Этот метод прост, но впечатляет. Вот его синтаксис:

```
arrayName.reverse()
```

А вот результаты, получаемые с его помощью:

```
var x = [1, 2, 3, 4];
x.reverse();
trace(x);  // Выводит: "4,3,2,1"
```

Обычно *reverse( )* применяется для изменения порядка сортировки в списке. Например, если есть список товаров, отсортированный по возрастанию цены, можно вывести его в порядке от самого дешевого к самому дорогому, а можно изменить порядок на обратный и вывести от самого дорогого к самому дешевому.

Упражнение для читателя: попробуйте написать собственную функцию для инвертирования порядка элементов в массиве. Это не только труднее, чем кажется: скорее всего, встроенный метод *reverse( )* окажется намного быстрее.

Методы sort() и sortOn()

Метод *sort( )* упорядочивает элементы массива в соответствии с любым заданным правилом. Если правило не задано, то *sort( )* по умолчанию размещает элементы в алфавитном порядке (приблизительно). Отсортировать массив по алфавиту действительно просто:

```
arrayName.sort()
```

При вызове метода массива *sort( )* без аргументов его элементы временно преобразуются в строки и сортируются в соответствии с кодовыми позициями Unicode, эквивалентными значениям ASCII для кодов, меньших 128. Кодовые позиции большинства западноевропейских языков можно найти в приложении B. См. также раздел «Порядок символов и сравнение по алфавиту» главы 4.

```
// Работает в соответствии с ожиданиями...
var animals = ["zebra", "ape"];
animals.sort();
trace(animals);  // Выводит: "ape,zebra"
                 // Здорово! Какой удобный маленький метод.

// Осторожно, порядок сортировки не строго алфавитный...Прописная "Z"
// в zebra идет раньше строчной "a" в "ape".
var animals = ["Zebra", "ape"];
animals.sort();
trace(animals);  // Выводит: "Zebra,ape". Увы. См. приложение B.
```

Можно также применять *sort( )* для организации элементов массива в соответствии с собственным правилом. Это более сложный прием, но он весьма мощный. Сначала надо создать *функцию сравнения* (*compare function*), которая определяет, как интерпретатор должен сортировать любые два элемента массива. Затем эта функция передается методу *sort( )* при его вызове, например:

```
arrayName.sort(compareFunction)
```

где *compareFunction* – это имя функции, сообщающей интерпретатору, как принимать решение при сортировке.

Для того чтобы построить функцию сравнения, надо создать новую функцию, которая принимает два аргумента (представляющие любые два элемента массива). В теле функции определяется, который из элементов должен появиться раньше в списке элементов после операции сортировки. Если требуется, чтобы первый элемент появился *раньше* второго, функция должна возвратить *отрицательное* число. Если надо, чтобы первый элемент появился *после* второго, функция должна возвратить *положительное* число. А если элементы должны остаться в том же положении, то функция должна возвратить 0. На псевдокоде этот подход в общем виде выглядит так:

```
function compareElements (element1, element2) {
  if (условие для появления element1 раньше element2) {
    return -1;
  } else if (условие для появления element1 после element2) {
```

```
    return 1;
  } else {
    return 0;  // Оставить элементы в покое
  }
}
```

Например, отсортировать элементы по возрастанию числовых значений можно посредством такой функции:

```
function sortAscendingNumbers (element1, element2) {
  if (element1 < element2) {
    return -1;
  } else if (element1 > element2) {
    return 1;
  } else {
    return 0;  // Элементы равны
  }
}

// Подготовив функцию сравнения, опробуем ее
var x = [34, 55, 33, 1, 100];
x.sort(sortAscendingNumbers);
trace(x);    // Выводит: "1,33,34,55,100"
```

Функции сравнения чисел можно в действительности формулировать значительно более кратко. Предыдущую функцию *sortAscendingNumbers( )* можно записать так:

```
function sortAscendingNumbers (element1, element2) {
  return element1 - element2;
}
```

В оптимизированной версии отрицательное число возвращается, если `element1` меньше `element2`; положительное число возвращается, если `element1` больше `element2`, а 0 возвращается, если два числа равны. Вот это элегантно! А теперь представим версию для сортировки по убыванию:

```
function sortDescendingNumbers (element1, element2) {
  return element2 - element1;
}
```

В примере 11.6 показана функция сравнения, которая немного изменяет поведение *sort( )* при обычном алфавитном сравнении – так, чтобы прописные и строчные буквы сортировались независимо от регистра.

Пример 11.6. Алфавитная сортировка массива без учета регистра

```
var animals = ["Zebra", "ape"];

function sortAscendingAlpha (element1, element2) {
  return (element2.toLowerCase() < element1.toLowerCase());
}
animals.sort(sortAscendingAlpha);

trace(animals);  // Displays: "ape,Zebra"
```

Конечно, в сравнении не обязательно должны участвовать только простые строки и числа. Здесь мы сортируем массив клипов в убывающем порядке соответственно площади в пикселах:

```
var clips = [square1, square2, square3];

function sortByClipArea (clip1, clip2) {
  clip1area = clip1._width * clip1._height;
  clip2area = clip2._width * clip2._height
  return clip1area - clip2area;
}

clips.sort(sortByClipArea);
```

Это тонкий механизм сортировки!

Flash MX вводит еще один полезный метод сортировки, отсутствовавший во Flash 5. Если элементами массива являются объекты, имеющие свойство с одинаковым именем, то можно быстро отсортировать объекты по значению этого свойства с помощью метода *sortOn( )*. Подробности представлены в описании *Array.sortOn( )* в «Справочнике по языку».

Метод slice()

Будучи своего рода подвидом *splice( )*, метод *slice( )* возвращает из массива ряд элементов. В отличие от *splice( )*, метод *slice( )* только *возвращает* элементы. Он создает новый массив и не изменяет оригинал, для которого был вызван. Синтаксис метода *slice( )*:

```
origArray.slice(startIndex, endIndex)
```

Здесь `startIndex` задает первый извлекаемый элемент, а `endIndex` указывает элемент *за* последним элементом, который нужно извлечь. Метод *slice( )* возвращает новый массив, содержащий копию последовательности элементов от `origArray[startIndex]` до `origArray[endIndex - 1]`. Если `endIndex` опущен, то по умолчанию принимается значение `origArray.length`, и возвращаемый массив содержит элементы от `origArray[startIndex]` до `origArray[origArray.length-1]`. Вот два примера:

```
var myList = ["a", "b", "c", "d", "e", "f"];
myList.slice(1, 3);  // Возвращает ["b", "c"], not ["b", "c", "d"]
myList.slice(2);     // Возвращает ["c", "d", "e", "f"]
```

Метод join()

Метод *join( )* можно применять для создания строки, представляющей все нумерованные элементы массива. Метод *join( )* начинает работу с преобразования каждого элемента указанного массива в строку. Неопределенные элементы преобразуются в пустые строки (`""`). Затем *join( )* конкатенирует все строки в одну длинную строку, разделяя их символом (или последовательностью символов), называемым *разделителем* (*delimiter*). Наконец, *join( )* возвращает созданную им строку. Синтаксис *join( )* такой:

```
arrayName.join(delimiter)
```

где *delimiter* – это строка, служащая для разделения преобразованных элементов массива *arrayName*. Если *delimiter* не задан, то по умолчанию его функцию выполняет запятая. Действие инструкции *join( )* легче всего понять на примере:

```
var siteSections = ["animation", "short films", "games"];
// Присваивает siteTitle значение "animation>> short films>> games"
var siteTitle = siteSections.join(">> ");
// Присваивает siteTitle значение "animation:short films:games"
var siteTitle = siteSections.join(":");
```

Обратите внимание, что инструкция *join( )* не изменяет массив, к которому она применена, а возвращает строку, получающуюся из этого массива.

При вызове без аргумента *join( )* ведет себя в точности, как *toString( )*. Поскольку *toString( )* не помещает пробел после запятой, разделяющей элементы, для красивого форматирования более удобным может оказаться применение *join( )* с запятой в качестве разделителя:

```
var x = [1, 2, 3];
trace(x.join(", "));  // Выводит: "1, 2, 3", а не "1,2,3"
```

Метод *join( )* может дать неожиданный результат, если применить его к массиву, содержащему элементы, которые *сами* являются массивами. Поскольку *join( )* преобразует элементы в строки, а массивы преобразуются в строки с помощью метода *toString( )*, то вложенные массивы преобразуются в строки с разделителем-запятой, а не с тем разделителем, который был передан *join( )* в виде аргумента. Иными словами, разделитель, переданный *join( )*, не действует во вложенных массивах. Например:

```
var x = [1, [2, 3, 4], 5];
x.join("|");  // Возвращает "1|2,3,4|5", а не "1|2|3|4|5"
```

Упражнение для читателей: напишите собственную функцию, использующую заданный разделитель для объединения вложенных массивов. Совет: обойдите в цикле все элементы массива и найдите те из них, которые сами являются массивами. С помощью оператора *instanceof* (описанного в главе 5) проверьте, является ли элемент данных массивом. Сможете ли вы создать рекурсивную версию, которая будет работать с массивами, вложенными на любую глубину?

Метод toString()

Как будет показано в главе 12, *toString( )* представляет собой универсальный метод объектов, возвращающий строковое представление того объекта, с которым он вызывается. В случае объекта *Array* метод *toString( )* возвращает список элементов массива, преобразованных в строки и разделенных запятыми. Метод *toString( )* может быть вызван явно:

```
arrayName.toString()
```

Обычно метод *toString( )* не используется явным образом, а вызывается автоматически, когда *arrayName* применяется в строковом контексте. Напри-

мер, когда мы пишем *trace(arrayName)*, в окне Output появляется список значений, разделенных запятыми; запись *trace(arrayName)* эквивалентна *trace(arrayName.toString( ))*. Метод *toString( )* часто удобно применять во время отладки, когда требуется быстро взглянуть на элементы массива, не форматируя их. Например:

```
var sites = ["www.moock.org", "www.macromedia.com", "www.oreilly.com"];
// Вывести массив в текстовом поле
debugOutput_txt.text = "The sites array is " + sites.toString();

trace("The sites array is " + sites.toString());   // Явное применение toString()
trace("The sites array is " + sites);              // Неявное применение toString()
```

Метод *join( )*, описанный в предыдущем разделе, предоставляет больше возможностей форматирования, чем *toString( )*.

Массивы как объекты

После прочтения главы 12 вы должны рассматривать массивы как тип объекта, а именно экземпляры класса *Array*. Как и к другим объектам, к объектам *Array* можно добавлять свойства. При добавлении к массиву именованных элементов в действительности происходит лишь добавление свойств к объекту *Array*. Однако, как было отмечено выше, для применения встроенных методов класса *Array* необходимо работать с нумерованными элементами.

В главе 9 был рассмотрен полезный гибрид «массив-объект» – объект *arguments*, в котором нумерованные элементы сочетались со свойствами. (Напоминаем, что у объекта *arguments* есть свойства `callee` и `caller`, а также в нем хранятся параметры в виде массива элементов.) Теперь можно понять, что все массивы хранят пронумерованный список элементов плюс встроенное свойство `length`, плюс любые свойства, добавленные пользователем.

Кроме того, поскольку массивы являются экземплярами класса *Array*, можно добавлять в *Array* новые методы, которые будут действовать со всеми массивами. Например, следующий код создает метод с именем *searchArray( )*, основанный на функции, которую мы создали в примере 11.2.

```
// Добавить во все массивы новый метод searchArray()
Array.prototype.searchArray = function (searchElement) {
  // Проверить каждый элемент на наличие в нем searchElement
  for (var i = 0; i < this.length; i++) {
    if (this[i] == searchElement) {
      return i;
    }
  }
  return null;
};

// Создать массив
var soundtracks = ["electronic", "hip hop", "pop", "alternative", "classical"];

// С помощью метода searchArray найти индекс слова "classical"
trace(soundtracks.searchArray("classical"));  // Выводит: 4
```

Для того чтобы определить, является ли объект массивом, используйте оператор *instanceof*. Например:

```
var colors = ["red", "blue", "green"];
trace(colors instanceof Array);  // Выводит: true
```

Многомерные массивы

До сих пор изложение было ограничено *одномерными массивами*, которые похожи на одну строку (или один столбец) таблицы. Но что делать, если требуется создать эквивалент таблицы, содержащий и строки, и столбцы? Необходимо второе измерение. ActionScript естественным образом поддерживает только одномерные массивы, но можно моделировать многомерные массивы, создавая массивы внутри массивов. Это означает, что можно создавать массивы, элементы которых сами являются массивами (иногда называемыми *вложенными* массивами).

Простейшим типом многомерного массива является двумерный массив, в котором элементы мысленно организованы в сетку строк и столбцов: строки представляют первое измерение массива, а столбцы – второе.

Посмотрим, как действуют двумерные массивы, на практическом примере. Допустим, что обрабатывается заказ, содержащий три наименования товаров, их количество и цену. Требуется моделировать таблицу из трех строк (по одной для каждого наименования) и двух столбцов (один для количества, а другой для цены). Создадим для каждой строки отдельный массив, рассматривая элементы как столбцы:

```
var row1 = [6, 2.99];   // Количество 6, цена 2.99
var row2 = [4, 9.99];   // Количество 4, цена 9.99
var row3 = [1, 59.99];  // Количество 1, цена 59.99
```

Затем поместим строки в массив-контейнер с именем `spreadsheet`:

```
var spreadsheet = [row1, row2, row3];
```

Теперь можно подсчитать суммарную стоимость заказа, перемножив в каждой строке количество и цену и сложив произведения вместе. Обращение к элементам двумерного массива производится с помощью двух индексов (одного для строки, а другого для столбца). Выражение `spreadsheet[0]`, к примеру, представляет массив из двух столбцов первой строки. Поэтому, чтобы получить доступ ко второму столбцу первой строки таблицы `spreadsheet`, мы используем `spreadsheet[0][1]`:

```
// Создадим переменную для хранения суммарной стоимости заказа
var total;

// Теперь найдем стоимость заказа. Для каждой строки перемножим
// столбцы и прибавим результат к total.
for (var i = 0; i < spreadsheet.length; i++) {
  total += spreadsheet[i][0] * spreadsheet[i][1];
}

trace(total);  // Выводит: 117.89
```

За исключением хранения и доступа, многомерные массивы ведут себя как обычные массивы. Для работы с данными, хранимыми в многомерных массивах, можно использовать все методы класса *Array*.

Вопросник с вариантами выбора: подход № 3

В примере 9.13 мы модифицировали вопросник с вариантами выбора из главы 1 (примеры 1.1–1.4). Тогда мы усовершенствовали код, создав многократно используемые централизованные функции. После изучения массивов можно вернуться к вопроснику и внести еще несколько усовершенствований. На этот раз организуем хранение ответов пользователя и правильных ответов на каждый вопрос в массивах, что сделает код более кратким, понятным и легко расширяемым, если понадобится добавить новые вопросы.

Изменения в вопроснике будут касаться слоя *scripts* кадра 1, где расположен основной код вопросника. Как предыдущую, так и данную обновленную версию вопросника можно найти в сетевом хранилище кода. Изменения, внесенные на этот раз, можно обнаружить, прочитав код примера 11.7 и уделив особое внимание комментариям.

Пример 11.7. Вопросник с вариантами выбора, версия 3

```
//  Остановить фильм на первом вопросе.
stop();

// ===================
// ИНИЦИАЛИЗИРОВАТЬ ПЕРЕМЕННЫЕ ВОПРОСНИКА
// ===================
// Сохранить ссылку на временную диаграмму с интерфейсом пользователя вопросника.
var quizTimeline = this;
// Создать массив для хранения ответов пользователя.
var userAnswers = new Array();
// Создать массив для хранения правильных ответов на каждый вопрос.
var correctAnswers = [3, 2];
// Вспомогательная переменная для хранения количества вопросов.
var numQuestions = correctAnswers.length;

// =====================
// СОЗДАТЬ ФУНКЦИИ ВОПРОСНИКА
// =====================
//   Функция: answer()
//   Описание: регистрирует ответы пользователя
//             и переходит к новому вопросу.
//  Параметры: choice           ответ пользователя на текущий вопрос
function answer (choice) {
  // Добавить текущий ответ в список ответов пользователя.
  userAnswers.push(choice);

  // Вспомогательная переменная для хранения номера текущего вопроса.
  var currentQuestion = userAnswers.length;

  // Показать вопрос и ответ в окне Output для отладки.
```

```
  trace("Question " + currentQuestion
        + ". The user answered: " + userAnswers[currentQuestion-1] + ".");

  // Если вопрос последний...
  if (currentQuestion == numQuestions) {
    // ... перейти в конечный кадр вопросника.
    quizTimeline.gotoAndStop("quizEnd");
    // Оценить ответы пользователя.
    gradeUser();
  } else {
    // ... в ином случае перейти к кадру со следующим вопросом.
    quizTimeline.gotoAndStop("q"+ (currentQuestion + 1));
  }
}

//   Функция: gradeUser()
//  Описание: подсчитывает результаты пользователя в конце опроса.
function gradeUser () {
  // Сообщить в окне Output, что мы собираемся оценить ответы.
  trace("Quiz complete. Now grading...");

  // Создать локальную переменную для хранения
  // количества правильных ответов.
  var totalCorrect = 0;

  // Подсчитать количество правильных ответов.
  // Для каждого вопроса...
  for (var i=0; i < numQuestions; i++) {
    // Если ответ пользователя совпадает с правильным.
    if (userAnswers[i] == correctAnswers[i]) {
      // Засчитать очко.
      totalCorrect++;
    }
    // Вывести правильный ответ и ответ пользователя
    // в окне Output для отладки.
    trace("Question " + (i + 1)
         + ". Correct answer: " + correctAnswers[i]
         + ". User's answer: "  + userAnswers[i]);
  }

  // Показать окончательный результат в окне Output для отладки.
  trace("User's score: " + totalCorrect + "/" + numQuestions);

  // Создать на экране текстовое поле для показа результата пользователя.
  quizTimeline.createTextField("totalOutput_txt", 1, 150, 200, 200, 20);

  // Вывести результат пользователя в текстовом поле на экране.
  quizTimeline.totalOutput_txt.text = "Your final score is: "
                                    + totalCorrect + "/" + numQuestions;

  // Настроить гарнитуру шрифта, высоту и цвет знаков для текстового окна.
  var formatObj = new TextFormat();
  formatObj.size = 16;
  formatObj.color = 0xC7FF9C;
```

```
  formatObj.font = "_sans";
  formatObj.bold = true;
  quizTimeline.totalOutput_txt.setTextFormat(formatObj);
}

// ================================
// ОПРЕДЕЛИТЬ ОБРАБОТЧИКИ КНОПОК ВОПРОСНИКА
// ================================
// Код, выполняемый при нажатии кнопки 1.
choice1_btn.onRelease = function () {
  // Вызвать answer() для записи ответа пользователя
  // и перехода к следующему вопросу.
  this._parent.answer(1);
};

// Код, выполняемый при нажатии кнопки 2.
choice2_btn.onRelease = function () {
  this._parent.answer(2);
};

// Код, выполняемый при нажатии кнопки 3.
choice3_btn.onRelease = function () {
  this._parent.answer(3);
};
```

Вперед!

В этой главе состоялось ваше первое знакомство с составным типом данных – массивами, и вы изрядно попрактиковались в работе с ними. Теперь вы знаете, как несколько элементов данных можно представить в одном контейнере, и готовы приступить к главе 12. В ней мы исследуем еще более мощный тип составных данных и фундаментальную составляющую ActionScript – *объекты*.

12

Объекты и классы

В этой главе рассказывается об *объектно-ориентированном программировании* (ООП), для многих читателей представляющем собой неизведанную область. Мы расскажем о методологии и технологии ООП и приведем для наглядности несколько практических примеров. В помощь опытным и начинающим программистам, стремящимся расширить свой кругозор, мы сравним практику ООП в ActionScript с методологией, которая существует в Java и других языках. Разработчики, имеющие опыт объектно-ориентированного программирования, желающие лишь получить представление о синтаксисе ООП в ActionScript, могут сразу перейти к разделу «Краткий справочник по ООП» в конце этой главы.

Тем же, кто такого опыта не имеет, возможно, приходилось слышать о таинственности ООП и о том, что его трудно понять. Совсем наоборот, идеи ООП весьма наглядны, а процесс ООП значительно легче, чем вас пытались убедить. По сути, ООП означает, что некоторые части программы рассматриваются как независимые модули, называемые «объектами». Под «независимостью» мы понимаем не изолированность объектов, а то, что они не обязательно зависят один от другого, и вы можете работать с объектом, не думая о его внутреннем устройстве. Концепции ООП будут понятны, если вы осознаете, что в реальном мире мы всегда имеем дело с независимыми объектами. Ваша собака, родители, машина и компьютер – все это независимые объекты. Что-то они делают самостоятельно, а что-то – по вашей просьбе, даже если вам неизвестен внутренний механизм их работы.

Не надо быть биологом, чтобы заставить свою собаку принести палку. Не надо быть механиком, чтобы вести машину. Не надо быть психоаналитиком, чтобы общаться с родителями, и, вопреки слухам, не надо быть специалистом в вычислительной технике, чтобы получать электронную почту. Достаточно знать, каким командам (они называются *методами*) подчиняется объект и каковы результаты выполнения этих команд. Например, нажимая на педаль газа, вы предполагаете, что машина увеличит скорость. Если вы даете собаке команду «сидеть!», то предполагаете, что собака сядет. Вооружившись такими житейскими представлениями об объектах реального мира, посмотрим, как эти представления связаны с ActionScript.

Классическим примером программного *объекта* служит прыгающий шарик. Как и у настоящего шарика, у объекта `ball` могут быть *свойства*, соответствующие его атрибутам, таким как радиус, цвет, масса, местонахождение и прыгучесть (упругость). Для представления прыгающего шарика в программе создадим объект `ball` со свойствами `radius` (радиус), `color` (цвет) и другими. Свойства объекта представляют его состояние в любой заданный момент, а некоторые из свойств со временем меняются. Например, чтобы заставить шарик двигаться, надо смоделировать законы движения Ньютона. Это означает, что необходимо на языке компьютера описать перемещение шарика с течением времени. В простейшем случае, вспомнив, что скорость, помноженная на время, равна расстоянию, можно определить будущее положение шарика по горизонтали с помощью такого уравнения:

```
Position = Speed * Time
```

Мы можем перевести это уравнение на язык ActionScript, чтобы рассчитать новое положение мячика:

```
ball.xPosition += ball.xVelocity * elapsedTime;
```

Это уравнение описывает движение шарика из текущего положения и добавляет пройденное расстояние (исходя из его скорости и прошедшего времени), в результате чего получается новое положение.

Поведение объекта – это действия, которые он может совершать, и услуги, которые он может предоставлять (одно из действий, на которые способен наш шарик, – перемещение). Обычно для описания поведения объекта в коде программы создаются так называемые *методы*, которые представляют собой функции, реализующие поведение объекта.

Например, можно создать метод *move( )* для нашего объекта `ball`, который реализует предыдущее уравнение. Поэтому методы можно рассматривать как команды, которым подчиняется объект. Конечно, у объекта может быть несколько методов. Допустим, требуется заставить шарик отскакивать. Можно создать метод *bounce( )*, меняющий направление движения шарика на противоположное и уменьшающий его скорость (шарик не обладает абсолютной упругостью). Метод *bounce( )* может реализовывать такое уравнение:

```
ball.xVelocity =  ((ball.xVelocity) * 0.95   // Упругость шарика равна 95%
```

Прежде чем проникнуть в тайны создания объектов, добавления свойств и реализации методов, формализуем некоторые из определений. *Объект* – это структура данных, объединяющая связанные между собой *свойства* (переменные) и *методы* (функции). Обычно объект *инкапсулирует* свое поведение, другими словами, внутренние детали выполнения им своих функций могут быть не видны снаружи. Программа взаимодействует с объектом посредством так называемых *интерфейсов* (то есть методов, доступ к которым открыт извне объекта). Другим частям программы совершенно не интересно, каким образом объект выполняет свои задачи. Программа лишь поставляет объекту *входные данные* и проверяет *выходные* (результаты), если это требуется.

Вы также услышите о *классах* и *экземплярах* (*instances*). *Класс* – это общая категория объектов, а экземпляр – лишь конкретный случай (то есть образец) класса объекта. Например, ваша домашняя собака представляет собой экземпляр общего класса *Dog*. Все собаки класса *Dog* лают и у них по четыре ноги, но у вашей собаки собственные значения свойств роста, веса и масти, которые применяются для описания собак. Если вам встречалась трехногая собака и вы возражаете против утверждения «у всех собак четыре ноги», значит, вы уже начали понимать ООП, – класс должен допускать всевозможные варианты своих экземпляров. Поэтому, чтобы удовлетворить ваше возражение против конструкции класса *Dog*, можно добавить в него свойство `numberOfLegs`, которое будет устанавливаться независимо для каждого отдельного объекта *Dog*.

Объектно-ориентированное программирование (ООП) – это всего лишь название технологии, в которой применяются объекты. Объекты и ООП настолько присущи ActionScript, что мы уже применяли их, возможно, не понимая этого. Нам уже знакомы такие объекты Flash, как клипы (MovieClip), реализованные как совокупность свойств и методов. Когда мы определяем высоту клипа с помощью `someClip._height`, то обращаемся к свойству `_height` объекта клипа. А когда мы воспроизводим клип с помощью `someClip`.*play( )*, то вызываем метод *play( )* этого объекта клипа.

Обычно все экземпляры класса объектов пользуются общими методами и именами свойств и отличаются один от другого *значениями* этих свойств.

Независимо от происхождения объектов – созданы они в нашей программе или встроенны в ActionScript – ООП четко отделяет компоненты программы один от другого (*инкапсулирует*) и позволяет им взаимодействовать между собой, не зная о деталях устройства друг друга. Это позволяет изменять внутреннее функционирование объекта без отрицательного воздействия на те части программы, которые работают с объектом, если только не меняются открытые методы объекта (то есть его интерфейс с внешним миром). Если вернуться, например, к нашему объекту `ball`, то нас не интересует, что законы физики изменяют характер движения нашего шарика. Мы лишь вызываем метод шарика *move( )*, а о деталях пусть заботится сам объект.

Другая приятная особенность ООП состоит в возможности обращаться с *разными* объектами, обладающими *разным* поведением, единообразно, если только в них реализованы методы с одинаковыми именами. Допустим, например, что есть объекты `circle` (круг) и `square` (квадрат). Если в обоих объектах реализован метод *getArea( )*, возвращающий площадь фигуры, то можно вызывать их методы *getArea( )*, не беспокоясь о том, как каждый из объектов вычисляет свою площадь.

В этой главе мы научимся создавать базовые объекты и определять категории объектов (то есть классы). Освоив основы, мы узнаем, как с помощью наследования использовать в классах и объектах общие характеристики. Например, можно реализовать класс *Horse* (лошадь), который наряду с классом

Dog (собака) является потомком класса *Mammal* (млекопитающее). В классе *Mammal* можно реализовать методы и свойства, общие для всех млекопитающих, например наличие шерсти, способность давать молоко, теплокровность и наличие четырех (не более!) ног. Таким образом, с помощью наследования можно создать «родословное дерево» классов. Позднее мы также узнаем, как с помощью ООП управлять средой Flash посредством встроенных объектов и классов ActionScript.

Не все объекты представляют собой какие-либо физические аспекты фильмов Flash. В конце главы мы покажем, как с помощью ООП реализовать наш вопросник с вариантами ответов. Мы воспользуемся объектами *Quiz*, *Question* и *QuizGUI*, из которых первые два представляют собой абстрактные понятия, а не физические сущности фильма. Наш класс *Quiz* не управляет клипами на сцене, а инкапсулирует все поведение типичного вопросника: добавление вопросов, вывод заголовка, задание ответов пользователя на конкретный вопрос и т. д. Даже сам ActionScript представляет такие неосязаемые сущности, как время, даты или форматирование, применяемое к текстовому полю, с помощью объектов.

Наконец, следует помнить, что ActionScript обладает значительной гибкостью и не принуждает к применению ООП. ООП можно использовать в части программы либо организовать с его помощью выполнение всех действий. Такая гибкость может смутить как опытных программистов ООП, так и новичков. Например, разработчики Java привыкли к тому, что программа состоит исключительно из классов, что допускается во Flash, но не обязательно в нем. Однако новичкам в ООП может быть неясно, в каком объеме они должны использовать ООП для создания нужного им кода. В этой главе мы будем рассматривать программу с точки зрения Java, согласно которой любой элемент программы представляет собой объект. Даже если в дальнейшем ваши разработки на Flash не будут основываться на таком подходе, демонстрация строгого ООП обогатит ваше понимание объектов и Flash на всех уровнях. Если вы новичок в ООП, попробуйте применять объекты везде, где вам потребуется создать более одного экземпляра какой-то структуры; например, пятнадцать шариков, десять противников или два вида вопросников (с вариантами ответов или с заполнением формы).

Надеюсь, что это введение пролило некоторый свет на объекты и ООП. Теперь займемся конкретными вопросами.

Анатомия объекта

Как и массив, объект служит контейнером для контейнеров. Типичный массив содержит несколько значений данных в отдельных нумерованных *элементах*. Аналогично объект содержит несколько значений данных в отдельных именованных *свойствах*. По сути, свойство – это переменная, определенная в объекте. На практике в языке программирования Java свойства объекта называют *переменными экземпляра*, или просто *переменными* (в Java объектами является все, поэтому там нет необходимости отличать пере-

менную от свойства объекта). При этом метод является функцией, определенной в объекте. Фактически объект есть не что иное, как совокупность связанных между собой переменных и функций.

На рис. 12.1 изображен пример свойств объекта с именем `ball`. У объекта `ball` есть два свойства: `radius` и `color`. Эти свойства имеют значения 50 и 0xFF0000 (шестнадцатеричное значение красного цвета). Имена свойств представляют собой уникальные идентификаторы, так же как названия переменных. Хотя у каждого свойства есть собственное имя, оба они хранятся в едином охватывающем их объекте, `ball`.

Рис. 12.1. *Пример структуры объекта*

Обычно в объекте определяются свойства, тесно связанные друг с другом. Говоря более конкретно, свойства объекта надо выбирать так, чтобы они отражали характеристики, позволяющие отличить один объект от другого. Объекты класса *MovieClip*, к примеру, обладают специфическими для отдельных клипов свойствами, такими как количество кадров (`_totalframes`) и положение на экране (`_x` и `_y`).

Свойства объектов принято устанавливать с помощью методов объекта, чтобы не нарушать принцип инкапсуляции. Это означает, что в строгом ООП объект должен устанавливать свои собственные свойства. Если извне требуется установить свойство объекта, это должно быть сделано путем вызова соответствующего метода объекта. Например, пуристу не понравилось бы непосредственное присваивание значения свойству клипа `_width`. Этот пурист стал бы доказывать, что лучше всего дать возможность объекту клипа самому устанавливать свойство `_width`, а код, находящийся вне объекта, должен делать это только косвенно путем вызова одного из методов объекта клипа, например *setWidth( )*. Благодаря ограничению доступа к свойствам с помощью методов разработчик класса *MovieClip* мог бы изменить имя свойства `_width` на `w`, не вызвав при этом неприятностей у других пользователей этого класса.

Создание экземпляров объектов

Допустим, что класс объектов уже создан. Это предположение не так уж нелепо, поскольку ActionScript предоставляет много встроенных классов,

и пользовательские классы, которые мы будем создавать, будут вести себя аналогично им. Предположив, что класс объектов уже есть, мы должны создать конкретный экземпляр объекта на базе этого класса. Например, ActionScript реализует класс *TextField*, но создание конкретных текстовых полей предоставляется пользователю.

Для создания экземпляра объекта обычно применяется оператор *new* и *функция конструктора*, с помощью которой инициализируется объект. Общий синтаксис при этом следующий:

```
new ConstructorFunction()
```

Создадим экземпляр нашего первого объекта с помощью функции конструктора, которая уже встроена в ActionScript: конструктор *Date( )*. Объект *Date* представляет определенный момент времени. При создании нового экземпляра объекта полученный экземпляр обычно сохраняется в переменной, элементе массива или свойстве другого объекта, чтобы в дальнейшем к нему можно было обращаться. Например, следующий код создает новый объект *Date* и сохраняет ссылку на него в переменной с именем `now`:

```
var now = new Date();
```

Конкретный момент времени представляет собой сложную группу чисел, включающую в себя секунды, минуты, час, день, месяц и год. С помощью методов объекта *Date* легко узнать или установить любую составляющую времени, представленного объектом. Например, можно вызвать метод *getHours( )* и получить час суток (в 24-часовом формате):

```
trace("The current hour is: " + now.getHours());
```

Прежде чем научиться добавлять методы в собственные классы, посмотрим внимательнее, как с ними можно работать.

Свойства объектов

Как вы уже знаете, свойства представляют собой именованные контейнеры данных, связанные с объектом. Они определяются классом объекта, но устанавливаются индивидуально для каждого экземпляра. Как и переменные, свойства объектов могут содержать любые строки, числа, булевы величины, `null`, `undefined`, функции, массивы, ссылки на клипы и даже другие объекты.

Обращение к свойствам

Знакомый оператор «точка» предоставляет доступ к свойствам объекта. Точка отделяет имя свойства от объекта, которому оно принадлежит:

```
objectName.propertyName
```

где *objectName* является именем объекта, а *propertyName* должно быть допустимым идентификатором, соответствующим имени некоторого свойства *objectName*.

Например, если есть экземпляр объекта `Ball` со свойством `radius`, то получить значение `radius` и присвоить его переменной `r` можно так:

```
var r = redBall.radius;
```

Присвоить свойству `radius` объекта `redBall` значение 50 можно так:

```
redBall.radius = 50;
```

Альтернативный способ состоит в том, чтобы обратиться к свойству с помощью оператора `[]`:

```
objectName[propertyNameString]
```

Оператор `[]` позволяет образовать имя свойства с помощью любого выражения, значением которого является строка. Например, сначала получим, а затем присвоим значение `radius`, получив такие же результаты, как в предыдущем примере:

```
var r = redBall["radius"];  // Получить
redBall["radius"] = 50;     // Присвоить
```

В данном случае оператор `[]` не так краток, как оператор «точка», и не предоставляет заметных преимуществ. Оператор `[]` полезен, когда имя свойства является сложным выражением. Кроме того, вместо строкового литерала мы задаем имя свойства с помощью выражения, значением которого является строка «radius»:

```
var prop = "radius";        // Присвоить переменной имя свойства.
trace(redBall[prop]);       // Переменная prop содержит имя свойства "radius".
```

Доступ к встроенным свойствам ActionScript также осуществляется с помощью операторов «точка» и `[]`. Здесь мы обращаемся к значению свойства `PI` объекта *Math* сначала с помощью оператора «точка», а затем с помощью оператора `[]`:

```
trace(Math.PI);
trace(Math["PI"]);
```

Однако в чистом ООП к свойству объекта почти никогда не обращаются напрямую, это делается с помощью методов. Например, для получения свойства `volume` экземпляра встроенного класса *Sound* пишут:

```
trace(mySound.getVolume());
```

а не:

```
trace(mySound.volume);  // НЕПРАВИЛЬНО!
```

Запомните, что оператор «точка» требует указать свойство в виде идентификатора, а не строки, тогда как оператор `[]` требует строку, а не идентификатор.

Обход свойств объекта в цикле for–in

В главе 8 мы узнали, что цикл *for-in* позволяет перечислять свойства объекта. Сейчас об объектах нам стало известно несколько больше, и стоит ненадолго вернуться к инструкции *for-in*, чтобы посмотреть, как с ее помощью осуществляется обход свойств объекта, то есть обращение к каждому из них по очереди.

Подобно всем циклам, инструкция *for-in* состоит из заголовка и тела. Тело инструкции *for-in* автоматически выполняется один раз для каждого свойства указанного объекта. Не надо заранее знать количество свойств или их имена, поскольку при каждой итерации цикла переменная «итератора» автоматически становится именем очередного свойства. Поэтому можно выполнить обход свойств объекта так:

```
// Перечислить свойства объекта ball
for (var prop in ball) {
  trace("Property " + prop + " has the value " + ball[prop]);
}
```

Обратите внимание, что переменная итератора `prop` в этом примере не является целым числом, каким она была бы в цикле *for*. Поэтому не путайте стандартный цикл *for*, обычно применяемый для доступа к нумерованным элементам массива, с циклом *for-in*, предназначенным для доступа к свойствам объекта. Дополнительные сведения о циклах *for-in*, включая способ предотвратить показ свойств объекта в *for-in*, можно найти в главе 8.

Методы объекта

Методы – это функции, связанные с объектом и обычно применяемые для выполнения задач, связанных с объектом, или доступа к данным объекта (то есть для получения или установки его свойств). Методы вызываются с помощью оператора вызова функций `()`, при этом имя метода отделяется от имени его объекта с помощью оператора «точка»:

```
objectName.methodName()
```

Например:

```
ball.getArea();  // Вызвать метод getArea() объекта ball
```

Как и свойства, методы определяются для класса и используются с отдельными экземплярами объектов. Например, в классе *MovieClip* определен метод *play( )*, который мы применяем к конкретным экземплярам клипов (например, к главной временной диаграмме фильма):

```
_root.play();
```

Методы могут принимать параметры, возвращать значения и в других отношениях вести себя точно так же, как функции, обладающие дополнительной возможностью обращаться к объекту, через который они вызваны, и воздействовать на него. Например, здесь мы вызываем *gotoAndPlay( )* как

метод объекта клипа `_root`, передавая ему в качестве аргумента число 5 – номер целевого кадра:

```
_root.gotoAndPlay(5);
```

Здесь мы вызываем метод *getDepth( )*, который возвращает целое число – уровень глубины:

```
trace(_root.getDepth());
```

Теперь, когда мы умеем использовать свойства и методы объектов с объектами существующих классов, посмотрим, как создавать свойства и методы собственных классов.

Классы и объектно-ориентированное программирование

Не так уж редко приходится создавать десятки сложных объектов, хранящих массу информации обо всем – от товаров в системе электронного магазина до персонажей видеоигр – нехороших субъектов, обладающих искусственным интеллектом. Для облегчения создания объектов и задания иерархий объектов (связей между ними) используются *классы* объектов. *Класс* служит определением в виде шаблона для целой категории объектов. Как говорилось во введении к этой главе, классы описывают родовые особенности определенных пород объектов, например «все собаки лают».

Классы объектов

Готовы ли вы поработать руками? Создадим пользовательский класс *Ball*. Первым шагом в создании класса является проектирование свойств и методов, которые он поддерживает. У нашего класса *Ball* будут такие свойства:

`radius`

Половина диаметра шарика.

`color`

Цвет шарика.

`xPosition`

Положение шарика по горизонтали.

`yPosition`

Положение шарика по вертикали.

А вот методы класса *Ball*:

moveClipTo( )

Служит для перемещения шарика.

getArea( )

Предназначен для определения размера площади, занимаемой отдельным шариком.

На обычном языке класс *Ball* можно описать интерпретатору так:

> *Ball* – это тип объекта. У всех объектов этого типа есть свойства `radius`, `color`, `xPosition`, `yPosition`, которые устанавливаются отдельно для каждого экземпляра. Все такие объекты совместно используют методы *moveClipTo( )* и *getArea( )*, одинаковые для всех членов класса *Ball*.

Экземпляры класса *Ball* не будут присутствовать на экране; пока они лишь теоретически представляют шарики в нашей программе. Ниже в разделе «Отображение шариков на экране» мы рассмотрим практический способ вывода изображения шарика на монитор. Затем, в главе 14, мы покажем, как реализовать класс *Ball* в виде клипа. Следите за событиями!

Создание класса

В ActionScript нет специального средства для «объявления класса»; нет инструкции «class», которая создавала бы новый класс, подобно инструкции *var*, создающей новые переменные. Вместо этого мы определим особый тип функции, называемой *конструктором*, которая генерирует новые экземпляры класса. В результате определения функции конструктора фактически создается шаблон класса, или его определение.

Синтаксически функции конструкторов (или просто *конструкторы*) имеют вид обычных функций. Например:

```
function Constructor () {
  statements
}
```

Именем функции конструктора класса может быть любое допустимое имя функции, но для того, чтобы было понятно, что это конструктор класса, принято писать его с прописной буквы. Имя конструктора должно описывать класс создаваемых им объектов, например *Ball*, *Product* или *Vector2d*. *Инструкции*, содержащиеся в конструкторе, инициализируют создаваемые им объекты.

Класс ActionScript и конструктор ActionScript нераздельны. Все переменные экземпляра создаются внутри конструктора, однако для методов это обычно не так, о чем вы вскоре узнаете. Тем, кто программирует на Java, следует обратить внимание, что в ActionScript у каждого класса может быть только один конструктор (то есть функции конструктора не могут *перегружаться*).

Для начала сделаем функцию конструктора *Ball( )* как можно более простой. Пока нам надо только, чтобы она создавала пустые объекты. Для создания пустого объекта нашему конструктору *Ball( )* не требуются никакие инструкции.

```
// Создать конструктор Ball()
function Ball () {

}
```

Это было нетрудно. Вскоре мы увидим, как создавать свойства внутри нашего конструктора и инициализировать создаваемые объекты с помощью функции конструктора *Ball( )*.

Создание экземпляров класса

Функция конструктора определяет класс и применяется для создания новых экземпляров класса. Как вы уже знаете из раздела «Создание экземпляров объектов» этой главы, при вызове вместе с оператором *new* функция конструктора создает, а затем возвращает экземпляр объекта.

Вспомним общий синтаксис создания нового объекта с помощью функции конструктора:

```
new Constructor();    // Возвращает экземпляр класса Constructor
```

Поэтому для создания объекта (*экземпляра*) с помощью класса *Ball* пишут:

```
// Сохраняет экземпляр класса Ball в переменной bouncyBall
var bouncyBall = new Ball();
```

Обратите внимание, что хотя наша функция конструктора *Ball( )* в настоящее время пуста и явным образом не возвращает экземпляры класса *Ball*, при вызове конструктора вместе с оператором *new* ActionScript автоматически создает экземпляр класса. Теперь снабдим конструктор нашего класса *Ball* возможностью присваивать значения свойствам создаваемых им объектов `ball`.

Создание свойств класса

Для создания свойств объекта на этапе его создания употребляется специальное ключевое слово `this`. Внутри функции конструктора ключевое слово `this` служит ссылкой на создаваемый в данный момент объект. С помощью этой ссылки можно присваивать зарождающемуся объекту любые свойства. Общая технология этого процесса описывается следующим синтаксисом:

```
function Constructor () {
  this.propertyName = value;
}
```

Здесь `this` указывает на создаваемый объект, *propertyName* — это свойство, которое следует прикрепить к объекту, а *value* представляет собой значение данных, присваиваемое этому свойству.

Применим эту технику в нашем примере с *Ball*. Вот как конструктор класса *Ball* может присваивать начальные значения свойств `radius`, `color`, `xPosition` и `yPosition` своим экземплярам. Обратите внимание на ключевое слово `this` и запомните, что внутри нашего конструктора класса *Ball* оно означает «новый создаваемый объект».

```
function Ball () {
  this.radius = 10;
  this.color = 0xFF0000;
```

```
  this.xPosition = 59;
  this.yPosition = 15;
}
```

При создании свойств объекта не следует применять ключевое слово *var*, употребляемое при создании переменных. Следующий код генерирует ошибку этапа компиляции:

```
var this.radius = 10;
```

С помощью такого конструктора *Ball( )* можно создавать экземпляры *Ball* (то есть объекты класса *Ball*) с определенными нами свойствами (`radius`, `color`, `xPosition` и `yPosition`) путем вызова *Ball( )* с оператором *new* так же, как и ранее. Например:

```
// Создать новый экземпляр Ball
var bouncyBall = new Ball();
```

Теперь мы можем обращаться к свойствам `bouncyBall`, установленным конструктором *Ball( )* при создании этого объекта, следующим образом:

```
trace(bouncyBall.radius);     // Выводит: 10
trace(bouncyBall.color);      // Выводит: 16711680 (0xFF0000 в десятичном виде)
trace(bouncyBall.xPosition);  // Выводит: 59
trace(bouncyBall.yPosition);  // Выводит: 15
```

Ну, не здорово ли?

К сожалению, пока наш конструктор *Ball( )* работает с фиксированными величинами, присваивая значения свойствам создаваемых им объектов (например, начальный радиус всегда равен 10). Поэтому все объекты класса *Ball* будут иметь одинаковые начальные значения свойств, что лишает код всякой гибкости. Добавим в наш конструктор параметры, которые позволят изменять начальные значения свойств и сделать каждый шарик уникальным с самого начала.

Вот общий синтаксис определения параметров для функции конструктора:

```
function Constructor (value1, value2, value3) {
  this.property 1 = value1;
  this.property2 = value2;
  this.property3 = value3;
}
```

Внутри функции *Constructor* ссылка на создаваемый объект осуществляется, как и раньше, с помощью ключевого слова `this`. Однако на этот раз значения свойств объекта не кодируются жестко. Вместо этого свойствам объекта присваиваются значения аргументов *value1*, *value2* и *value3*. Когда требуется создать новый уникальный экземпляр класса, в функцию конструктора передаются начальные значения его свойств:

```
someObject = new Constructor (value1, value2, value3);
```

Посмотрим на примере 12.1, как применить это к классу *Ball*.

Пример 12.1. Класс Ball со свойствами

```
// Пусть конструктор Ball() принимает значения свойств как аргументы
function Ball (radius, color, xPosition, yPosition) {
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Вызовем наш конструктор дважды, передавая ему аргументы, используемые
// в качестве значений свойств новых объектов
var greenBall = new Ball(42, 0x00FF00, 59, 15);
var whiteBall = new Ball(14, 0xFFFFFF, 50, 12);

// Посмотрим, получилось ли...
trace(greenBall.radius);  // Выводит: 42,  :) Круто...
trace(whiteBall.radius);  // Выводит: 14
                          // У каждого экземпляра уникальные значения свойств
```

Принято давать передаваемым параметрам (`radius`, `color` и т. д.) имена, аналогичные соответствующим внутренним свойствам (`this.radius`, `this.color` и т. д.), но не следует смешивать те и другие друг с другом. Первые – это временные переменные параметров, через которые осуществляется передача аргументов, которые применялись при вызове функции конструктора; вторые (начинающиеся с `this`) – это свойства объекта, в которых эти значения хранятся постоянно.

Обычно свойства класса объявляются в начале его конструктора. Как правило, методы класса не должны добавлять в него новые свойства. В отличие от Java, ActionScript настолько гибок, что позволяет методам создавать новые свойства объектов, но при таком подходе структура класса становится трудно контролируемой. Методы должны только *модифицировать* свойства, объявленные в конструкторе. Однако с целью сокращения расхода памяти класс, в котором есть, скажем, 100 свойств, может поручить создание свойств методам, если создание экземпляров оказывает заметное влияние на производительность Flash Player.

Создание класса *Ball* почти завершено. Но можно заметить, что все еще отсутствуют методы *moveClipTo( )* и *getArea( )*, о которых говорилось ранее. Ими мы теперь и займемся.

Создание методов класса

Методы добавляются в класс путем присваивания функций особому свойству `prototype` конструктора класса (о технической роли `prototype` будет рассказано далее в этой главе в разделе «Всемогущая цепочка прототипов»). Общий синтаксис таков:

```
Constructor.prototype.methodName = function;
```

Здесь *`Constructor`* – это функция конструктора класса, *`methodName`* представляет собой имя метода, а *`function`* – это функция, выполняемая при обраще-

нии к методу. Можно передавать *function* как ссылку на функцию или как литерал функции. Например:

```
// Ссылка на функцию
Ball.prototype.getArea = calcArea;
function calcArea () {
  // ...здесь находится тело метода...
}

// Литерал функции
Ball.prototype.getArea = function () {
  // ...здесь находится тело метода...
};
```

Создание метода посредством литерала функции, как правило, предпочтительнее, поскольку при этом после назначения метода в памяти не остается ненужной ссылки на функцию.

Java-разработчикам следует обратить внимание на то, что в ActionScript не обязательно задавать тип данных для значений, возвращаемых методом, и его параметров. Аналогично невозможно установить уровень доступа к методу, а также нет синтаксических средств для создания статических методов (хотя позже мы узнаем, как их можно моделировать).

Как вы уже знаете, метод объекта часто извлекает или устанавливает свойства объекта, с которым он вызывается. Поэтому возникает вопрос, каким образом метод ссылается на объект для получения доступа к его свойствам? Ответ: с помощью волшебного ключевого слова `this`. Вспомните, что мы уже работали с ним в функции конструктора для ссылки на новые объекты *Ball*, создаваемые в данный момент конструктором. Внутри метода ключевое слово `this` также указывает на объект, с которым в настоящий момент выполняются действия. Например, для доступа к свойству `radius` отдельного экземпляра *Ball* внутри метода *getArea( )* применяется `this.radius`, как показано в следующем листинге метода *Ball.getArea( )*:

```
Ball.prototype.getArea = function () {
  // Ключевое слово this указывает на объект, с которым вызывается getArea().
  return Math.PI * this.radius * this.radius;
};
```

Аналогично с помощью `this` метод может устанавливать значение свойства, как показывает наш метод *moveClipTo( )* класса *Ball*. Метод *moveClipTo( )* устанавливает новые значения для свойств `xPosition` и `yPosition`. Обратите внимание, что метод *moveClipTo( )* принимает два параметра, `x` и `y`, которые указывают на новое положение шарика:

```
Ball.prototype.moveClipTo = function (x, y) {
  // Обратите внимание на ключевое слово this, указывающее на перемещаемый шарик.
  this.xPosition = x;
  this.yPosition = y;
};
```

Теперь мы создали все свойства и методы класса *Ball*, так что наша работа завершена! Окончательный код показан в примере 12.2. Прикрепите код примера к кадру 1 нового фильма; затем выберите Control → Test Movie и посмотрите на код в действии.

Пример 12.2. Полный код класса Ball

```
// Конструктор класса Ball().
function Ball (radius, color, xPosition, yPosition) {
  // Создать свойства Ball.
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}
// Метод Ball.getArea().
Ball.prototype.getArea = function () {
  // Ключевое слово this указывает на шарик, метод getArea() которого был вызван
  return Math.PI * this.radius * this.radius;
};

// Метод Ball.moveClipTo().
Ball.prototype.moveClipTo = function (x, y) {
  // Обратите внимание, что this употребляется для ссылки на перемещаемый шарик.
  this.xPosition = x;
  this.yPosition = y;
};
// Создать большой красный шарик с начальной позицией (0, 0).
var bigBall = new Ball(600, 0xFF0000, 0, 0);

// Проверить, какую площадь занимает шарик, исходя из его радиуса.
trace(bigBall.getArea());  // Выводит: 1130973.35529233

// Переместим шарик с помощью метода moveClipTo().
bigBall.moveClipTo(100, 200);

// Проверим новое значение xPosition.
trace(bigBall.xPosition);  // Выводит: 100
```

У вызова метода и его определения схожий синтаксис, но весьма разные результаты. Очень важно понимать различие между ними. При определении метода круглые скобки должны быть опущены:

```
SomeClass.prototype.someMethod = someFunction;
```

При вызове метода круглые скобки должны присутствовать:

```
someObj.someMethod();
```

Различие в синтаксисе тонкое. Следите за правильностью кода.

Методы доступа

Выше отмечалось, что в строгом ООП непосредственный доступ к свойствам объекта не одобряется. А для доступа к свойствам следует обращаться к так называемым методам получения/записи. Метод *получения* (*getter*) извлекает

значение свойства, а метод *записи* (*setter*) устанавливает его. Например, наш класс *Ball* определяет свойство `radius`. Непосредственное обращение к этому свойству не приветствуется, а правильным стилем считается обращение к нему посредством методов *getRadius( )* и *setRadius( )*. Рассмотрим, например, такую упрощенную версию нашего класса *Ball*:

```
// Класс Ball
function Ball (radius) {
  // Свойства экземпляра
  this.radius = 0;

  // Инициализация
  this.setRadius(radius);
}

Ball.prototype.getRadius = function () {
  return this.radius;
}

Ball.prototype.setRadius = function (newRadius) {
  this.radius = newRadius;
}

Ball.prototype.getArea = function () {
  return Math.PI * this.getRadius() * this.getRadius();
};

// Экземпляр ball
var largeBall = new Ball(400);
largeBall.setRadius(500);
trace(largeBall.getArea());  // Выводит: 78.5398163397448
```

Конструктор класса объявляет свойство `radius` и предлагает для него тип данных, присваивая ему число (0). Однако при любом обращении к `radius` посредниками служат методы *getRadius( )* и *setRadius( )*. Даже сам конструктор присваивает начальное значение *Radius* с помощью *setRadius( )*. Внутри метода *getArea( )* свойство `radius` извлекается путем обращения к *getRadius( )*. А за его пределами с помощью *setRadius( )* мы присваиваем радиусу `largeBall` новое значение. Методы доступа позволяют нам указать, что мы хотим изменять некоторый атрибут шарика, но предоставляем объекту самому позаботиться о том, как реализовать такое изменение. При изменении внутренней реализации, например, начав измерять свойство `radius` в сантиметрах вместо пикселов, мы не нарушим работы какого-либо кода, использующего эти методы доступа; потребуется изменить только сами методы доступа. Если бы код вне объекта обращался к свойству `radius` непосредственно, то у программиста, работающего с объектом, были бы более ограниченные возможности его модификации.

Методы получения/записи надежно инкапсулируют механизм действия объекта, но надо признать, что они более многословны, чем присваивание свойствам, и часто затрудняют чтение кода. Поэтому на практике многие программисты не утруждают себя реализацией методов доступа, а некоторые

разработчики не используют эти методы, даже когда они имеются. Чтобы решить данную проблему, во Flash MX во все объекты, производные от класса *Object*, был введен метод *addProperty( )*. Метод *addProperty( )* создает так называемое *свойство доступа* (*getter/setter property*), при обращении к которому автоматически вызывается соответствующий метод доступа. Это позволяет программистам устанавливать и получать свойства объекта более кратким способом, в то же время согласуясь с требованиями чистого ООП. Общую технику демонстрирует следующий пример. Значительно более подробно о ней рассказывается в статье *Object.addProperty( )* «Справочника по языку».

```
// Класс Ball
function Ball (radius) {
  // Свойства экземпляра.
  this._radius = 0;

  // Инициализация
  this.setRadius(radius);
}

Ball.prototype.getRadius = function () {
  return this._radius;
}

Ball.prototype.setRadius = function (newRadius) {
  this._radius = newRadius;
}

Ball.prototype.getArea = function () {
  return Math.PI * this.radius * this.radius;
};

// Свойство доступа radius, определенное в прототипе
Ball.prototype.addProperty("radius",
                           Ball.prototype.getRadius,
                           Ball.prototype.setRadius);

// Экземпляр ball
var largeBall = new Ball(400);
largeBall.radius = 5;        // Автоматически вызывает метод setRadius().
trace(largeBall.getArea());  // Выводит: 78.5398163397448
```

Обратите внимание, что класс использует внутреннее свойство `_radius` для хранения значения свойства доступа с именем `radius`. Имя внутреннего свойства не должно совпадать с именем видимого снаружи свойства доступа, поэтому принято помещать в его начале символ подчеркивания. Свойства доступа можно использовать для создания свойств, доступных только для чтения, о чем говорится в статье *Object.addProperty( )* «Справочника по языку».

Упражнение для читателя: попробуйте добавить методы *getColor( )* и *setColor( )*, которые получают и устанавливают свойство `color` в классе *Ball*. Когда это получится, попробуйте с помощью *Object.addProperty( )* превратить `color` в свойство доступа. Не забудьте использовать внутреннее свойство `_color` для хранения свойства доступа с именем `color`!

Альтернативные способы назначения методов

Существует возможность добавить в каждый экземпляр метод, присвоив свойству экземпляра функцию в конструкторе класса, но этот способ не рекомендуется. Вот общий синтаксис:

```
function Constructor () {
  this.methodName = function;
}
```

В ActionScript присваивать каждому экземпляру метод в конструкторе класса можно потому, что технически методы ActionScript представляют собой функции, хранящиеся в свойствах объектов. Функцию, присваиваемую в качестве нового метода, можно передать несколькими способами, что показывают следующие примеры. *function* может быть литералом функции, например:

```
function Ball (radius) {
  this.radius = radius;
  // Добавить метод getArea()...
  this.getArea = function () { return Math.PI * this.radius * this.radius; };
}
```

В другом варианте *function* можно объявить *внутри* конструктора:

```
function Ball (radius) {
  this.radius = radius;
  // Добавить метод getArea()...
  this.getArea = calcArea;
  function calcArea () {
    return Math.PI * this.radius * this.radius;
  }
}
```

Наконец, *function* можно объявить *вне* конструктора, но присвоить свойству *внутри* конструктора:

```
// Объявить функцию getArea()
function calcArea () {
  return Math.PI * this.radius * this.radius;
}

function Ball (radius) {
  this.radius = radius;
  // Добавить метод getArea()...
  this.getArea = calcArea;
}
```

Особой разницы между этими тремя подходами нет. Обычно литералы функций удобнее всего, но они не так способствуют многократному использованию кода, как функции, определяемые вне конструктора, благодаря чему с ними могут работать другие конструкторы.

Тем не менее все три способа недостаточно эффективны и не могут быть рекомендованы. Если с помощью описанной технологии присваивать метод

объектам класса, возникает ненужное дублирование метода в *каждом экземпляре* класса. Формула *getArea( )* одна и та же для каждого шарика, поэтому код, выполняющий данную задачу, следует хранить в центральном месте – свойстве конструктора `prototype`. Плохо это или хорошо, но ActionScript не относится к строгим языкам и часто предоставляет несколько способов решения аналогичных задач.

Статические методы и свойства

В Java *статическими методами*, или *методами класса*, называют методы, вызываемые непосредственно через класс, а не через объект, то есть экземпляр класса. Аналогично *статическая переменная* Java соответствует свойству, обращение к которому происходит через класс независимо от экземпляра объекта. Статические методы и свойства Java предназначены для хранения поведения и информации, логически связанной с классом, но доступной всюду, а не только через конкретный экземпляр объекта. Например, класс *Square* может определять статическое свойство `numSides`, задающее количество сторон для всех его экземпляров. Методы доступа для статических свойств, например *getNumSides( )* и *setNumSides( )*, принято реализовывать в виде статических методов. Встроенный класс ActionScript *TextField* определяет статический метод *getFontList( )*, возвращающий список шрифтов, установленных в данной системе. Метод *getFontList( )* по смыслу связан с текстовыми полями в целом, а не с конкретным текстовым полем, поэтому он реализован в виде статического метода.

В ActionScript нет синтаксического механизма для непосредственного объявления методов или свойств как статических, реализованного в Java. Статические методы и свойства создаются путем прикрепления свойств и методов к функции конструктора класса, например:

```
// Определение класса
function SomeClass () {
  statements
}

// Статическое свойство
SomeClass.someStaticProperty = value;

// Статический метод
SomeClass.someStaticMethod = function;
```

Обратите внимание, что в отличие от Java у статических методов ActionScript нет ограничений на переменные, свойства или методы, к которым они могут обращаться (в Java статический метод может обращаться только к статическим переменным и методам). Кроме того, в статическом методе ActionScript ключевое слово `this` указывает на функцию конструктора класса, тогда как в статических методах Java это ключевое слово употреблять нельзя.

При определении свойства, которое будет хранить константу, также допускается использование свойства `prototype` класса, например:

```
function Square () {
```

```
    statements
  }
  Square.prototype.numSides = 4;
```

Однако при таком подходе требуется, чтобы обращение к `numSides` осуществлялось через экземпляр *Square*, а не непосредственно через класс *Square*. Для того чтобы можно было работать со свойством, не создавая при этом экземпляра класса, данное свойство следует сделать статическим.

Иногда статические свойства и методы применяются для объединения ряда взаимосвязанных констант и функций в один класс, экземпляры которого создаваться не будут. В ActionScript для этого обычно помещают свойства и методы в общий экземпляр класса *Object*. Таким способом, например, реализованы объекты *Math*, *Key*, *Mouse* и *Stage*.

Для удобства программистов на Java в табл. 12.1 приведены примерные соответствия методов и свойств Java и ActionScript.

Таблица 12.1. Эквивалентные свойства в Java и ActionScript

Название в Java	**Описание для Java**	**Описание для ActionScript**	**Пример для ActionScript**
Переменные экземпляра	Локальные переменные экземпляра объекта	Свойства, определенные в функции конструктора класса и копируемые в объекты	`function Square (side) { this.side = side; }`
Методы экземпляра	Методы, вызываемые с экземпляром объекта	Методы, определенные в объекте `prototype` функции-конструктора класса, к которым автоматически происходит обращение через `prototype` при вызове с экземпляром	`Square.prototype.getArea = function () { ... };` `mySquare.getArea();`
Переменные класса	Переменные, имеющие одно и то же значение для всех экземпляров объектов класса	Свойства, определенные в функции-конструкторе класса	`Square.numSides = 4;`
Методы класса	Методы, вызываемые через класс	Методы, определенные в функции-конструкторе класса	`Square.findSmaller = function (square1, square2) { ... }`

Наследование: надклассы и подклассы

Одно из важнейших свойств классов в объектно-ориентированом программировании заключается в их способности наследовать свойства и методы. Это означает, что класс может получить все свойства и методы другого класса и добавить к ним свои специфические возможности. Класс, заимствующий свойства и методы другого класса, называют *подклассом*. Класс, у которого заимствуют свойства и методы, называют *надклассом*. Говорят, что

подкласс *расширяет* свой надкласс, поскольку он заимствует (*наследует*) свойства своего надкласса, но добавляет к ним свои собственные. Несколько подклассов могут иметь один общий надкласс. Например, может существовать надкласс *Car* с подклассами *Coupe* и *Sedan*. Способность классов принимать методы и свойства от других классов называют *наследованием*.

Допустим, например, что нам надо создать два типа программ-вопросников: вопросник с вариантами ответов и вопросник с заполнением формы. У этих двух программ много общего. Обе они загружают вопросы, показывают вопросы, принимают ответы и подсчитывают результат пользователя. Однако существуют некоторые различия в том, как они показывают вопросы, получают ответы и подсчитывают результаты. Вместо двух отдельных, очень похожих между собой программ мы создадим общий класс *Quiz*, в котором будут определены общие методы и свойства обоих вопросников. Затем для каждого из конкретных типов вопросников мы создадим два подкласса *Quiz*, а именно *MultipleChoiceQuiz* и *FillInBlankQuiz*. Эти подклассы наследуют все возможности *Quiz*, но вводят методы и свойства, специфические для задач выбора варианта или ввода ответа в форме. Эти подклассы могут даже замещать (*override*) поведение одного или нескольких методов *Quiz*.

На примере нашего простого класса *Ball* посмотрим, как реализуется наследование в ActionScript. Ранее мы создали для *Ball* метод *getArea( )*. Предположим теперь, что мы хотим создать несколько типов круглых объектов. Вместо того чтобы создавать метод *getArea( )* для каждого класса круглого объекта, мы создадим общий надкласс *Circle* и определим в нем общий метод *getArea( )* для расчета площади любого круглого объекта. После этого мы сделаем *Ball* (и все другие классы круглого объекта) подклассом *Circle*, благодаря чему *Ball* унаследует метод *getArea( )* у *Circle*. Обратите внимание на иерархичность – простейший класс, надкласс *Circle*, определяет самые общие методы и свойства. Другой класс, а именно подкласс *Ball*, строится на основе этого простого класса *Circle*, вводя новые функции, специфичные для экземпляров класса *Ball*, но получая от *Circle* основные атрибуты, общие для всех круглых объектов.

Создание надкласса

Ранее мы узнали, что методы определяются в особом свойстве `prototype` функции конструктора класса.

Чтобы указать *надкласс* данного класса, мы целиком заменяем свойство `prototype` этого класса новым экземпляром нужного надкласса.

Формальное рассмотрение свойства `prototype` мы на некоторое время отложим, а пока займемся общим синтаксисом описания надкласса:

```
SubclassConstructor.prototype = new SuperClassConstructor();
```

Здесь *SubclassConstructor* – это функция конструктора подкласса, а *SuperClassConstructor* – функция конструктора надкласса. В примере 12.3 этот прием применяется к нашим классам *Ball* и *Circle*.

Пример 12.3. Создание надкласса

```
// Создать конструктор Circle (надкласса)
function Circle () {
  // Здесь свойства экземпляра и инициализация.
}

// Создать метод Circle.getArea()
Circle.prototype.getArea = function () {
  return Math.PI * this.radius * this.radius;
};

// Создать конструктор обычного класса Ball
function Ball (radius, color, xPosition, yPosition) {
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Сделать Ball подклассом Circle
Ball.prototype = new Circle();

// Создадим экземпляр Ball и проверим его свойства
var bouncyBall = new Ball (16, 0x445599, 34, 5);
trace(bouncyBall.xPosition);   // 34
trace(bouncyBall.getArea());   // 804.247719318987
                               // Метод getArea() унаследован от Circle
```

По умолчанию новые объекты наследуют свойства и методы класса *Object*. Мы присваиваем значение свойству `prototype` функции конструктора, чтобы задать надкласс, отличный от общего класса *Object*. Далее, в разделе «Всемогущая цепочка прототипов» этой главы, мы выясним в точности, как действует данный механизм.

Иерархия классов примера 12.3 организована плохо – *Ball* определяет `radius`, но свойство `radius` является общим для всех круглых фигур, и по этой причине место ему – в классе *Circle*. То же самое относится к `xPosition` и `yPosition`. Чтобы исправить структуру, переместим `radius`, `xPosition` и `yPosition` в *Circle*, оставив в *Ball* только `color`. (В нашем примере будем рассматривать `color` как свойство, которым могут обладать только шарики.)

Вот принципиальная организация наших модифицированных конструкторов *Circle( )* и *Ball( )* (метод *Circle.getArea( )* не показан):

```
// Создать конструктор Circle (надкласса)
function Circle (radius, xPosition, yPosition) {
  this.radius = radius;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Создать конструктор класса Ball
function Ball (color) {
```

```
  this.color = color;
}
```

После перемещения свойств мы сталкиваемся с новой проблемой. Каким образом присваивать значения свойствам `radius`, `xPosition` и `yPosition`, если объекты создаются с помощью *Ball*, а не *Circle*? В код конструктора *Ball( )* надо внести еще одно изменение. Сначала мы заставим *Ball* принимать все необходимые свойства в виде параметров:

```
function Ball (color, radius, xPosition, yPosition) {
```

Затем в конструкторе *Ball( )* воспользуемся оператором *super* для вызова функции конструктора *Circle* с создаваемым объектом *Ball*. При этом мы передадим конструктору *Circle* аргументы, полученные *Ball*:

```
super(radius, xPosition, yPosition);
```

Полностью код класса/надкласса приведен в примере 12.4.

Пример 12.4. Класс и его надкласс

```
// Создать конструктор Circle (надкласса)
function Circle (radius, xPosition, yPosition) {
  this.radius = radius;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Создать метод Circle.getArea()
Circle.prototype.getArea = function () {
  return Math.PI * this.radius * this.radius;
};

// Создадим конструктор класса Ball
function Ball (color, radius, xPosition, yPosition) {
  // Вызовем конструктор Circle() с объектом ball, передав ему значения
  // переданные как аргументы конструктору Ball()
  super(radius, xPosition, yPosition);
  // Установить цвет шарика
  this.color = color;
}

// Установить Circle надклассом Ball
Ball.prototype = new Circle();

// Создадим экземпляр Ball и проверим его свойства
var bouncyBall = new Ball (0x445599, 16, 34, 5);
trace(bouncyBall.xPosition);   // 34
trace(bouncyBall.getArea());   // 804.247719318987
trace(bouncyBall.color);       // 447836
```

Во Flash 5 оператор *super* отсутствует, поэтому надо заменить строку

```
  super(radius, xPosition, yPosition);
```

следующими тремя строками, решающими ту же задачу:

```
this.superClass = Circle;
this.superClass(radius, xPosition, yPosition);
delete this.superClass;
```

При задании надкласса с помощью стандартного способа, приведенного в примере 12.4, все имеющиеся методы, определенные в свойстве `prototype` конструктора подкласса, будут уничтожены! Следите за тем, чтобы надкласс подкласса был задан *перед* созданием методов подкласса!

Доступ к замещенным методам

Если в подклассе реализован метод с таким же именем, как у метода его надкласса, то версия метода подкласса замещает версию метода надкласса. Это означает, что экземпляры подкласса будут использовать реализацию этого метода, имеющуюся в подклассе, а не в надклассе, если она существует. Однако в методе подкласса можно получить доступ к версии любого метода из надкласса с помощью оператора *super*. Допустим, например, что в *Circle* и *Ball* есть методы с одинаковым названием *toString( )*:

```
Circle.prototype.toString = function () {
  return "A circular object";
}

Ball.prototype.toString = function () {
  return "A ball object";
}
```

По умолчанию экземпляры *Ball* будут использовать метод *toString( )* класса *Ball* и игнорировать метод с таким же именем в *Circle*. Однако можно создать в *Ball* другой метод с именем *superToString( )*, который будет специально обращаться к *Circle.toString( )*:

```
Ball.prototype.superToString = function () {
  return super.toString();
}
```

Подробнее об операторе *super* рассказано в главе 5.

Полиморфизм

Наследование делает возможным существование другого ключевого понятия ООП, *полиморфизма*. Слово «полиморфизм» означает «много форм». Вы сообщаете объекту, что он должен сделать, детали реализации оставляете на его усмотрение. Лучше всего проиллюстрировать это примером. Допустим, разрабатывается игра, персонажами которой являются полицейские и грабители. Одновременно на экране отображается масса полицейских, грабителей и невинных прохожих. Полицейские преследуют грабителей, грабители убегают от полицейских, а пораженные и испуганные невинные прохожие перемещаются случайным образом. Допустим, что в коде игры для каждой категории персонажей создается класс объектов:

```
function Cop () { ... }
function Robber () { ... }
function Bystander () { ... }
```

Кроме того, создадим надкласс *Person*, наследниками которого являются все классы *Cop*, *Robber* и *Bystander*:

```
function Person () { ... }
Cop.prototype       = new Person();
Robber.prototype    = new Person();
Bystander.prototype = new Person();
```

В каждом кадре фильма Flash каждый персонаж на экране должен перемещаться согласно правилам своего класса. Для этого в каждом объекте определяется метод *move( )* (метод *move( )* настраивается для каждого класса):

```
Person.prototype.move = function () { ... перемещение по умолчанию ... }
Cop.prototype.move = function () { ... перемещение в погоне за грабителем ... }
Robber.prototype.move = function () { ... перемещение грабителя, убегающего
                                                          от полицейского ... }
Bystander.prototype.move = function ()  { ... беспорядочное перемещение ... }
```

Требуется, чтобы в каждом кадре Flash все персонажи перемещались по экрану. Для управления всеми действующими лицами создадим главный массив объектов *Person*. Вот пример заполнения элементов массива `persons`:

```
// Создать полицейских
var cop1 = new Cop();
var cop2 = new Cop();

// Создать грабителей
var robber1 = new Robber();
var robber2 = new Robber();
var robber3 = new Robber();

// Создать прохожих
var bystander1 = new Bystander();
var bystander2 = new Bystander();

// Создать массив, заполненный полицейскими, грабителями и прохожими
persons = [cop1, cop2, robber1, robber2, robber3, bystander1, bystander2];
```

В каждом кадре фильма Flash вызываем функцию *moveAllPersons( )*, определенную таким образом:

```
function moveAllPersons () {
  for (var i=0; i < persons.length; i++) {
    // Вызвать метод move() для каждого объекта лица в массиве
    persons[i].move();
  }
}
```

При вызове *moveAllPersons( )* каждый полицейский, грабитель и прохожий перемещаются согласно отдельным правилам, которые определены в их классах в методах *move( )*. Это и есть полиморфизм в действии: объекты с об-

щими характеристиками можно организовать вместе, сохранив отличительные индивидуальные особенности. У полицейских, грабителей и прохожих много общего, что воплощено в надклассе *Person*. У них есть общие операции, такие как перемещение. Однако общие операции могут быть по-разному реализованы, а классы могут поддерживать и другие специфические операции и данные.

Полиморфизм позволяет единообразно обращаться с разнородными объектами. С помощью функции *move( )* можно вызвать перемещение любого объекта, хотя в каждом классе функция *move( )* уникальна. На самом деле, даже если классы *Cop*, *Robber* и *Bystander* не являются наследниками *Person*, их метод *move( )* можно использовать единообразно.

Определение принадлежности объекта к надклассу

Допустим, что у нас есть класс *Rectangle* (прямоугольник), являющийся подклассом *Shape* (фигуры). Проверим, является ли объект потомком *Shape*, для чего воспользуемся оператором *instanceof*, который появился во Flash MX:

```
rect = new Rectangle(15, 35);
trace(rect instanceof Rectangle);  // Возвращает true
trace(rect instanceof Circle);     // Возвращает false
trace(rect instanceof Shape);      // Возвращает true
```

С помощью *instanceof* можно выполнять над объектом условные действия, зависящие от его класса. В следующем примере, взятом из вопросника в конце этой главы, из `quizClip` удаляются все клипы и текстовые поля. Мы выполняем обход свойств `quizClip` и вызываем для каждого из них соответствующий метод удаления в зависимости от того, является ли свойство экземпляром класса *TextField* или принадлежит к типу данных *MovieClip*.

```
for (var p in quizClip) {
  if (typeof quizClip[p] == "movieclip") {
    quizClip[p].removeMovieClip();
  } else if (quizClip[p] instanceof TextField) {
    quizClip[p].removeTextField();
  }
}
```

Во Flash 5 оператора *instanceof* нет, поэтому класс объекта следует определять с помощью цепочки объектов `prototype` (называемой *цепочкой прототипов*), как показано в примере 12.5. Вскоре мы подробнее изучим объекты `prototype` и `__proto__`.

Пример 12.5. Проход по цепочке прототипов

```
// Эта функция проверяет, является ли theObj потомком theClass
function objectInClass (theObj, theClass) {
  while (theObj.__proto__ != null) {
    if (theObj.__proto__ == theClass.prototype) {
      return true;
```

```
      }
      theObj = theObj.__proto__;
    }
    return false;
  }

  // Создать новый экземпляр Rectangle
  myObj = new Rectangle();

  // Проверить, наследует ли myRect классу Shape
  trace (objectInClass(myRect, Shape)); // Выводит: true
```

Окончание цепочки наследования

Все объекты ведут свое происхождение от класса верхнего уровня *Object*. Таким образом, все объекты наследуют свойства, определенные в конструкторе *Object* (например, методы *toString( )* и *valueOf( )*). Поэтому можно добавить новые свойства к каждому объекту фильма, если добавить новые свойства в класс *Object*. Свойства, прикрепленные к `Object.prototype`, распространяются по всей иерархии классов, включая встроенные объекты *Array*, *TextField* и даже *MovieClip*. Такой подход позволяет создавать своего рода глобальные переменные или функции. Например, в следующем коде ширина и высота сцены (*stage*) жестко кодируются в прототипе класса *Object*. После этого данная информация становится доступной в любом клипе (такой прием часто применялся в фильмах Flash 5, но во Flash Player 6 размеры сцены можно получить, прочитав свойства `Stage.width` и `Stage.height`):

```
Object.prototype.mainstageWidth  = 550;
Object.prototype.mainstageHeight = 400;
trace(anyClip.mainstageWidth);              // Выводит: 550
```

Свойства `Object.prototype` наследуются не только клипами, но и всеми объектами, поэтому даже объекты, созданные пользователем, и экземпляры встроенных классов *XML*, *Date* и *Sound* наследуют свойства, установленные в `Object.prototype`. Этот побочный эффект вызывает появление лишних свойств в циклах *for-in*. Поэтому при экспорте во Flash Player 6 всегда создавайте глобальные свойства и методы с помощью объекта `_global`. Подробности представлены в описании `_global` в «Справочнике по языку».

Упражнение для читателя. Прикрепить новые свойства и методы можно к любому встроенному классу. Попытайтесь добавить функцию нечувствительной к регистру сортировки по алфавиту из примера 11.6 к классу *Array* в качестве пользовательского метода.

Вывод Ball на экран

С самого начала этой главы мы наблюдали за классом *Ball*, излагая попутно массу теоретических сведений. Однако в результате наших трудов на экране еще не появился ни один реальный шарик. Дело в том, что обновление экрана не обязательно должно входить в круг обязанностей класса *Ball*. Отрисовка текущего содержимого экрана часто возлагается на отдельный класс,

специально занимающийся экраном и графическим интерфейсом пользователя. Как рисовать на экране, когда и что на нем изображать, может существенно зависеть от данного конкретного приложения. Например, в вопроснике, приводимом в конце главы, экран будет обновляться классом *QuizGUI* при каждом показе вопроса. Однако в других приложениях возможно интенсивное перемещение объекта `ball`, поэтому его понадобится часто отображать с помощью *MovieClip.onEnterFrame( )* или *setInterval( )*.

Вот основные вопросы, которые следует учитывать, проектируя систему, отвечающую в приложении за вывод информации на экран.

- Сначала надо решить, будут ли выводимые на экран элементы отображать себя сами или этим будет заниматься специальный класс. Например, будет ли в классе *Ball* свой метод *drawBall( )*, а в каком-нибудь другом классе – метод *drawBalls( )* или *drawScreen( )*?
- Решить, как часто следует отображать элементы – при возникновении какого-либо события (например, щелчка мыши) или периодически (с частотой воспроизведения кадров Flash Player).
- Выбрать метод отображения. Каждый видимый элемент фильма Flash должен отображаться в клипе на сцене. Однако клипы можно размещать вручную на этапе разработки, прикреплять с помощью ActionScript на этапе выполнения или даже создавать с нуля с помощью MovieClip Drawing API.
- Выбрать политику обновления экрана. Будет ли каждый видимый элемент поддерживаться как один постоянный регулярно обновляемый клип или все содержимое сцены будет удаляться и создаваться заново при каждом обновлении экрана?

Принятые решения точно определят, каким образом приложение будет управлять своими видимыми объектами. Например, можно создать класс *BallSimulation*, который будет обновлять экран, рисуя круги, представляющие список объектов `ball`. Либо можно добавить в *Ball* свойство `mc`, в котором будет храниться ссылка на клип, представляющий каждый объект *Ball*. Метод *Ball.drawBall( )* мог бы при этом обновлять отображение соответствующего клипа для объекта *Ball*. Когда объект и представляющий его клип так тесно связаны, может оказаться разумным реализовать класс в виде подкласса *MovieClip*, как описано в главе 14.

Отдельные экземпляры Object как ассоциативные массивы

Как мы уже видели, ActionScript настолько гибок, что позволяет методам создавать новые свойства объектов. В действительности гибкость ActionScript еще выше – он позволяет любому коду создать новое свойство или метод в отдельном экземпляре объекта (не во всех экземплярах класса сразу). Например, в следующем коде создается `obj`, экземпляр общего класса *Object*, в котором сразу создаются новые свойства `x` (равное 10) и `y` (равное 20):

```
var obj = new Object();
obj.x = 10;
obj.y = 20;
```

Конечно, в ООП не поощряется назначение свойств и методов отдельным объектам. Но в ActionScript это обеспечивает совершенно законный способ хранения группы именованных величин (аналогично хешу или ассоциативному массиву). На практике многие встроенные методы ActionScript принимают общие объекты в качестве параметров. Например, *MovieClip.globalToLocal( )* принимает как параметр объект точки, через который передаются свойства x и y. Свойства x и y представляют глобальную точку, которая будет преобразована в локальную систему координат клипа.

```
pt = new Object();            // Создать общий объект для хранения точки
pt.x = 0;                     // Левая граница главной сцены
pt.y = 0;                     // Верхняя граница главной сцены
this.globalToLocal(pt);       // Преобразовать pt в локальные координаты
trace("From the current clip, the top-left corner of the main Stage is at ");
trace("x: " + pt.x + "y: " + pt.y);
```

Аналогично *MovieClip.attachMovie( )* принимает параметр *initObj* для передачи свойств объекта в новый экземпляр клипа car_mc:

```
// Создать объект
var initObj = new Object();

// Создать свойства объекта
initObj.speed = 3;
initObj.onEnterFrame = function () {
  this._x += this.speed;
};

// Передать свойства объекта в car_mc
this.attachMovie("CarSymbol", "car_mc", 1, initObj);
```

Обратите внимание, что в этом коде *onEnterFrame( )* определяется как метод *initObj*, что для данной задачи совершенно допустимо (и удобно!).

Литералы объектов

Для того чтобы создать объект класса *Object* непосредственно в коде, не прибегая к конструктору *Object*, можно воспользоваться литералом объекта – точно так же, как литералом строки или литералом массива. Литерал объекта – это ряд разделенных запятыми пар свойство–значение, заключенных в фигурные скобки. Вот общий синтаксис:

```
{ property1: value1, property2: value2, property3: value3 }
```

Имена свойств в литерале объекта должны быть допустимыми идентификаторами, как описано в главе 14. В качестве значений могут выступать любые допустимые выражения. Например:

```
// Литерал объекта с двумя числовыми свойствами
myObject = { x: 30+10, y: 20*2 };
```

Литералы объектов применяются для краткости кода. Например, ниже показан более краткий способ передачи в car_mc свойств speed и onEnterFrame (возможно, ценой некоторого ухудшения читаемости):

```
this.attachMovie("CarSymbol", "car_mc", 1, {speed:3, onEnterFrame: function () {
                                                this._x += this.speed;}});
```

Всемогущая цепочка прототипов

В главе 2 мы видели, как интерпретатор использует цепочку областей видимости для разрешения (поиска значения) неквалифицированных ссылок на переменные. Для поиска значений свойств интерпретатор применяет *цепочку прототипов*. Мы уже сталкивались со свойством property при изучении методов и надклассов. Теперь заглянем «за кулисы» системы наследования ActionScript и посмотрим, как реально действует свойство prototype. В данном контексте термин *свойство* будет обозначать свойства, методы и обработчики событий объектов (в последних двух случаях формально речь идет о функциях, хранящихся в свойствах).

Как мы видели, каждая функция конструктора класса автоматически получает свойство с именем prototype. Например, при создании класса *Book* конструктор *Book* автоматически определяет Book.prototype:

```
function Book () {
}
trace(Book.prototype);  // Выводит: [object Object]
```

Изначально свойство prototype функции конструктора содержит стандартный объект (то есть экземпляр общего класса *Object*). Ранее мы добавляли в объект prototype методы, которые становились доступными всем экземплярам, создаваемым конструктором. Как это действует?

Допустим, что мы определяем свойство непосредственно в объекте – либо вне конструктора:

```
var importantDates = new Object();
// Определить свойство gwenBirthday.
var importantDates.gwenBirthday = "January 17";
```

либо внутри него:

```
function Book (author) {
  // Определить свойство author.
  this.author = author;
}
var asGuide = new Book("Colin Moock");
```

Если мы запросим у интерпретатора какое-либо свойство, он сначала проверит, определено ли свойство в самом объекте. Например, в следующем коде интерпретатор проверяет, есть ли в asGuide свойство author, находит его и выводит значение в окно Output:

```
trace(asGuide.author);  // Выводит: Colin Moock
```

Но если интерпретатор не может найти указанное свойство в экземпляре объекта, он ищет свойство `prototype` конструктора класса. Пусть, например, мы определили свойство `description` в `Book.prototype`:

```
Book.prototype.description = "Bound paper with writing on it.";
```

Тогда мы проверяем значение `asGuide.description`:

```
trace(asGuide.description);  // Выводит: Bound paper with writing on it.
```

Интерпретатор не может найти `description` непосредственно в `asGuide`, поэтому он ищет его в `Book.prototype`. `Book.prototype` определяет `description`, поэтому интерпретатор использует его в качестве значения `asGuide.description`. Но если мы добавим свойство `description` непосредственно в экземпляр *Book*, в конструкторе *Book* или вне его, свойство экземпляра замещает свойство `Book.prototype`. Например:

```
function Book (author, desc) {
  // Определить свойство author
  this.author = author;
  // Определить свойство description
  this.description = desc;
}
Book.prototype.description = "Bound paper with writing on it.";

var asGuide = new Book("Colin Moock", "A book about ActionScript");
trace(asGuide.description);  // Выводит: "A book about ActionScript"
                             // А не: "Bound paper with writing on it."
```

Посмотрим, как это действует в отношении методов, добавив в *Book* метод *getAuthor( )*:

```
Book.prototype.getAuthor = function () {
  return this.author;
}
```

При вызове *asGuide.getAuthor( )* интерпретатор проверяет, определено ли в `asGuide` свойство с именем `getAuthor`. Если нет, интерпретатор ищет `getAuthor` в `Book.prototype`. Там интерпретатор находит наш пользовательский метод и выполняет его.

Что бы произошло, если бы `Book.prototype` не определил `getAuthor`? Внимательный читатель вспомнит, что по умолчанию `Book.prototype` хранит экземпляр общего объекта. Когда интерпретатор не может найти свойство в этом общем объекте, он проверяет свойство `prototype` конструктора этого объекта, которым, как вы догадались, является `Object.prototype`. Так действует цепочка прототипов: если свойство не найдено в экземпляре объекта, поиск выполняется в `prototype` класса объекта. Если в нем нет этого свойства, поиск производится в `prototype` класса объекта `prototype` и так далее по цепочке, пока не будет достигнут `Object.prototype`. Если свойство не определено в `Object.prototype`, его значением становится `undefined`. Вот цепочка прототипов для нашего экземпляра `asGuide` (обратите внимание, что цепочки прототипов, как и цепочки областей видимости, читаются снизу вверх):

```
Object.prototype      // Если нет здесь, возвратить undefined
Book.prototype        // Если нет здесь, искать в Object.prototype
asGuide               // Если нет здесь, искать в Book.prototype
                      // ЗДЕСЬ ИНТЕПРЕТАТОР НАЧИНАЕТ ПОИСК СВОЙСТВ
```

Если имя свойства появляется в цепочке прототипов многократно, используется значение самого нижнего свойства в цепочке (как мы это видели раньше на примере свойства `description`). Более низкое в цепочке свойство *замещает* значение более высокого. Например, класс *Object* определяет метод *toString( )* в `Object.prototype`. Мы можем заместить эту версию *toString( )* для нашего класса *Book*, определив *toString( )* в `Book.prototype`:

```
Book.prototype.toString = function () {
  return this.description + ", by " + this.getAuthor();
}

trace(asGuide.toString());   // Выводит: "A book about ActionScript,
                             // by Colin Moock"
                             // Но если мы удалим Book.prototype.toString,
                            // Выводит: [object Object], что является результатом,
                             // возвращаемым Object.prototype.toString().
```

Теперь вспомним синтаксис для назначения надкласса некоторого класса:

```
// Сделать Book надклассом Product
Book.prototype = new Product();
```

Разберем, как эта инструкция устанавливает отношение наследования между *Book* и *Product*. Прежде всего, мы знаем, что у всех экземпляров *Book* есть все свойства, определенные в `Book.prototype`. Мы также знаем, что по умолчанию `Book.prototype` является экземпляром класса *Object*. Назначая таким способом *Product* в качестве надкласса *Book*, мы заменяем общий экземпляр в `Book.prototype` совершенно новым экземпляром *Product*. В результате все свойства, определенные в *Product*, становятся определенными в `Book.prototype` и потому доступными во всех экземплярах *Book*. При хранении экземпляра *Product* в `Book.prototype` цепочка прототипов для нашего экземпляра *Book* `asGuide` становится такой:

```
Object.prototype      // Если здесь не найдено, возвратить undefined
Product.prototype     // Если здесь не найдено, искать в Object.prototype
Book.prototype        // Если здесь не найдено, искать в Product.prototype
asGuide               // Если здесь не найдено, искать в Book.prototype
                      // ЗДЕСЬ ИНТЕПРЕТАТОР НАЧИНАЕТ ПОИСК СВОЙСТВ
```

Цепочка прототипов может содержать любое количество уровней. Класс *Product* может сам иметь надкласс, который передает свои свойства всем подклассам.

Если вам очень хочется узнать, определено ли свойство в самом объекте или унаследовано от одноименного в цепочке прототипов, можно использовать метод *object.hasOwnProperty( )*, описанный в «Справочнике по языку». Например:

```
trace(asGuide.hasOwnProperty("author"));      // true
trace(asGuide.hasOwnProperty("getAuthor"));   // false
                                              // (потому что getAuthor определен
                                              // в Book.prototype, а не в asGuide)
```

Наоборот, если нужно получить доступ к свойству в цепочке прототипов, которое замещено, то с помощью свойства `__proto__` (о котором рассказывается в следующем разделе) можно явно обратиться к свойству `prototype` класса объекта. Например, следующий код показывает, как обратиться к `Book.prototype.description` из конкретного экземпляра *Book*, `asGuide`:

```
trace(asGuide.__proto__.description);
```

Свойство __proto__

Ранее в этой главе мы видели, что оператор *instanceof* сообщает, принадлежит ли экземпляр конкретному классу. Но как интерпретатор узнает, к какому классу принадлежит отдельный объект? Или, более точно, как интерпретатор узнает, к `prototype` какого конструктора обращаться при поиске свойства объекта? Хотя в объектно-ориентированных языках внутренняя информация такого рода обычно скрыта, в ActionScript она доступна разработчику.

При создании любого объекта интерпретатор автоматически создает в нем особое свойство с именем `__proto__` (обратите внимание на удвоенный символ подчеркивания с каждой стороны слова `proto`). Затем интерпретатор копирует в `__proto__` ссылку на свойство `prototype` класса объекта. Например, при создании экземпляра `asGuide` класса *Book* свойство `asGuide.__proto__` получает значение `Book.prototype`:

```
var asGuide = new Book("Colin Moock", "A book about ActionScript");
trace(asGuide.__proto__ == Book.prototype);  // Выводит: true
```

Свойство `__proto__` применяется, главным образом, для внутренних нужд интерпретатором ActionScript. Однако мы можем с его помощью определить класс объекта (как показано выше в примере 12.5) или обратиться к замещенным свойствам. Хотя свойство `__proto__` доступно для записи, в большинстве случаев изменять его не следует. Тем не менее вскоре мы столкнемся с ситуацией, в которой `__proto__` используется как часть альтернативной системы наследования.

Свойство constructor

Когда создается объект конструктора `prototype`, интерпретатор автоматически назначает ему особое свойство с именем `constructor`. Свойство `constructor` выступает в качестве ссылки на функцию конструктора класса, в которой определен объект `prototype`. Так, следующее равенство проверяет, указывают ли `Book` и `Book.prototype.constructor` на одну и ту же функцию:

```
trace(Book.prototype.constructor == Book);  // Выводит: true
```

Обратите внимание, что свойство `constructor` содержит *ссылку* на функцию конструктора, а не на строку, представляющую имя функции. С помощью `constructor` можно вызвать функции конструктора объекта, чтобы повторно его инициализировать. В следующем примере конструктор *Book* вызывается с `asGuide`:

```
asGuide.constructor.call(asGuide,
                         "Colin Moock",
                         "A large book about ActionScript");
```

Подробности представлены в описании *Function.call( )* в «Справочнике по языку».

Свойство `constructor` обеспечивает также возможность обращения к статическим свойствам и методам экземпляра. Например:

```
function Box () {
  // Здесь находится код конструктора
}
Box.numSides = 4;

var bigBox = new Box();
trace(bigBox.constructor.numSides);  // Выводит: 4
```

В ECMA-262, JavaScript и ActionScript свойство `constructor` записывается в прототип класса (как *`Class`*`.prototype.constructor`), а экземпляры класса могут извлекать его по цепочке прототипов. Однако ActionScript также помещает свойство `constructor` непосредственно в экземпляры (как *`instanceName`*`.constructor`), закрывая *`Class`*`.prototype.constructor`. Нестандартная реализация ActionScript считается в Macromedia ошибкой: стандарт ECMA-262 требует, чтобы экземпляр извлекал свой конструктор через *`Class`*`.prototype.constructor`. Вероятно, в последующих версиях Flash Macromedia исправит эту ошибку (то есть в будущем `constructor` не будет записываться непосредственно в экземпляры). В следующем разделе мы увидим, как запись свойства `constructor` в отдельные экземпляры влияет на наследование.

Более сложные вопросы стандартного назначения надкласса

Как мы видели, стандартный способ задания надкласса для класса заключается в следующем:

```
SubClass.prototype = new SuperClass();
```

При использовании оператора *new* для организации наследования некоторые разработчики столкнулись с небольшими проблемами, которые мы сейчас детально обсудим. Чтобы обойти эти трудности, иногда применяют для назначения надкласса неофициальный прием – так называемое наследование, основанное на *__proto__*:

```
SubClass.prototype.__proto__ = SuperClass.prototype;
```

Механизм наследования на основе __proto__ таков. Мы знаем, что экземплярам *SubClass* доступны свойства и методы, определенные в SubClass.prototype. Кроме того нам известно, что SubClass.prototype сам представляет собой объект, обращающийся к свойствам и методам собственного класса через свойство __proto__ (помните цепочку наследования?). Поэтому наследование, основанное на __proto__, говорит: «Предоставить объекту SubClass.prototype доступ к свойствам и методам, определенным в SuperClass.prototype (через SubClass.prototype.__proto__)».

Macromedia не рекомендует использовать __proto__ при организации наследования вместо оператора new. Однако организация наследования с помощью __proto__ относительно часто применяется как средство, позволяющее обойти следующие проблемы.

Проблема 1: перезапись prototype.constructor

Если *SubClass* – функция конструктора, то свойство *SubClass*.prototype.constructor первоначально хранит ссылку на *SubClass*. Что произойдет при попытке обратиться к свойству constructor экземпляра *SubClass* как к *instanceName*.constructor? В ECMA-262 конструктор не определен в экземплярах, поэтому ссылка разрешается через цепочку прототипов, и возвращается значение *SubClass*.prototype.constructor. Однако в ActionScript конструктор определен в самом экземпляре, поэтому возвращается *instanceName*.constructor, и интерпретатор не продолжает поиск по цепочке прототипов. Рассмотрим пример:

```
function Ball () {
  // Здесь находится определение класса
}
b = new Ball();
trace(b.constructor);
```

В ECMA-262 b.constructor указывает на функцию конструктора *Ball( )*, хранящуюся в Ball.prototype.constructor. В ActionScript для Flash Player 5 и Flash Player 6 b.constructor тоже указывает на функцию конструктора *Ball( )*, но на этот раз значение берется из свойства constructor, прикрепленного непосредственно к b.

В этом простом примере b.constructor в конечном счете разрешается (resolves) в одно и то же значение как в ECMA-262, так и в ActionScript. Так важно ли, что ECMA-262 считывает свойство constructor из prototype-класса, а ActionScript читает его из каждого отдельного экземпляра? Ответ зависит от того, одинаковы ли значения этих двух свойств constructor при обращении к конструктору. В простейшем случае, когда экземпляр создается из класса, несоответствие между ECMA-262 на самом деле спорно, поскольку значение constructor экземпляра совпадает со свойством класса prototype.constructor. Однако если надкласс назначается классу с помощью стандартного синтаксиса наследования (*SubClass.prototype = new superclass( );*), то прежнее свойство prototype подкласса заменяется новым экземпляром *SuperClass*. Поэтому все имеющиеся свойства, прикрепленные к SubClass.prototype, включая constructor, будут утрачены; наоборот, будут добавлены все

свойства нового экземпляра *SuperClass*. Поэтому после присваивания *SuperClass* в качестве значения *SubClass*.prototype свойство *SubClass*.prototype.constructor будет ссылаться на функцию конструктора *SuperClass*, а не на конструктор *SubClass*, как прежде.

Это демонстрирует следующий код:

```
// Создать SubClass
function SubClass () {
   // Здесь находится определение класса
}

// Проверить конструктор подкласса перед установлением наследования
trace(SubClass.prototype.constructor == SubClass);    // Выводит: true

// Создать SuperClass
function SuperClass () {
   // Здесь находится определение класса
}

// Установить наследование с помощью стандартного синтаксиса (с оператором new)
SubClass.prototype = new SuperClass();

// Проверить SubClass.prototype.constructor после организации наследования
trace(SubClass.prototype.constructor == SubClass);    // Теперь выводит: false!
trace(SubClass.prototype.constructor == SuperClass);  // Теперь выводит: true!
```

Обратите внимание, что *SubClass*.prototype.constructor совпадает с функцией конструктора *SuperClass*, а не с функцией конструктора *SubClass*. Каковы последствия такого очевидного расхождения? И по-прежнему ли отход ActionScript от стандарта ECMA-262 является предметом обсуждения?

Вспомните, что в JavaScript и ECMA-262 экземпляры *SubClass* получают значение constructor из *SubClass*.prototype.constructor, поэтому экземпляры *SubClass* сообщают о конструкторе *SuperClass* вместо собственного конструктора *SubClass*. Стандарт ECMA-262 был реализован именно так умышленно, и это не считается ошибкой, хотя и вносит некоторую путаницу.

Вспомните также, что во Flash Player 5 и Flash Player 6 ActionScript отходит от стандарта ECMA-262 и записывает свойство непосредственно в каждый экземпляр. Вследствие этого во Flash Player 5 и Flash Player 6 *anyInstance*.constructor всегда указывает на функцию, создавшую *anyInstance*, в данном случае *SubClass*, даже после того, как установлено наследование надклассу. Например:

```
subInstance = new SubClass();
trace(subInstance.constructor == SuperClass);  // false в Flash 5 и 6
trace(subInstance.constructor == SubClass);    // true в Flash 5 и 6
// Обратите внимание, что эти проверки равенства во Flash Player 5
// всегда возвращают true из-за не относящейся к делу ошибки
// в операторе == (исправленной во Flash Player 6).
```

Поскольку subInstance.constructor закрывает свойство *SubClass*.prototype.constructor, программисты ActionScript обычно вообще не замечают обманчивость этого свойства. Поэтому данная проблема ActionScript для Flash

Player 5 и 6 является несколько спорной. Однако не следует полагаться на свойство `constructor`, которое Flash 5 и 6 прикрепляют непосредственно к экземплярам, поскольку можно ожидать, что будущие версии Flash Player не будут его поддерживать.

У некоторых читателей может возникнуть соблазн «исправить» странную реализацию ECMA-262 путем присваивания *`SubClass`*`.prototype.constructor` другого значения, указывающего на первоначальную функцию конструктора *SubClass*, например:

```
SubClass.prototype.constructor = SubClass;
```

Однако Macromedia не рекомендует напрямую модифицировать *`SubClass`*`.prototype.constructor`, поскольку в результате изменяется предопределенное (хотя, возможно, неожиданное) значение свойства, совместимое с ECMA. Если вам необходима точная ссылка на функцию конструктора экземпляра, попробуйте добавить пользовательское свойство для хранения ссылки на конструктор (добавлять свойство следует после назначения надкласса):

```
// Назначить SuperClass
SubClass.prototype = new SuperClass();
// Создать пользовательское свойство constructorFunc
SubClass.prototype.constructorFunc = SubClass;
```

Если свойство `constructor` применяется только для того, чтобы определить, принадлежит ли объект заданному классу, то имеет смысл предпочесть оператор *instanceof*. Последний возвращает `true`, если указанный экземпляр принадлежит заданному классу или какому-либо из его подклассов. И работает корректно независимо от того, установлено ли наследование с помощью `__proto__` или оператора *new*:

```
SubClass.prototype = new SuperClass();
var subInstance = new SubClass();
trace(subInstance instanceof SuperClass);  // Выводит: true
trace(subInstance instanceof SubClass);    // Выводит: true
```

Проблема 2: нежелательный вызов конструктора SuperClass

При выполнении следующей инструкции выполняется функция *SuperClass*:

```
SubClass.prototype = new SuperClass();
```

Этот избыточный вызов функции без нужды расходует ресурсы и может выполнять в конструкторе *SuperClass* нежелательные действия (такие, как прикрепление клипа фильма или инкрементирование счетчика экземпляров). Для предотвращения нежелательного выполнения кода используется параметр, указывающий на то, что конструктор вызывается только для организации наследования. Например:

```
function SuperClass () {
  // Если первым аргументом служит объект,
  // свойство noConstruct которого имеет значение true...
  if (arguments[0].noConstruct === true) {
```

```
      // ...прервать выполнение конструктора
      return;
    }

    // Выполнить обычные действия конструктора
  }
  SubClass.prototype = new SuperClass( {noConstruct:true} );
```

Иногда бывает достаточно проверить количество аргументов, переданных функции конструктора *SuperClass*. Здесь функция ожидает получения двух аргументов и завершает работу, если не было передано никаких аргументов, как бывает при использовании *new* для организации наследования:

```
function SuperClass (param1, param2) {
  // Если не передано аргументов...
  if (arguments.length == 0) {
    // ...прервать выполнение конструктора
    return;
  }

  // Обычное выполнение конструктора
}
SubClass.prototype = new SuperClass();
```

При альтернативном способе организации наследования с помощью `__proto__`:

```
SubClass.prototype.__proto__ = SuperClass.prototype;
```

объект `SuperClass.prototype` присваивается непосредственно свойству `SubClass.prototype.__proto__`, и конструктор *SuperClass* не выполняется.

Проблема 3: в SubClass.prototype появляются неопределенные свойства

Обычно конструктор надкласса присваивает значения свойствам экземпляров, которые он создает. В данном примере конструктор *SuperClass* присваивает в экземплярах значение свойству `p`:

```
function SuperClass (p) {
  this.p = p;
}
```

Когда наследование организуется с помощью стандартного синтаксиса *new*, экземпляр *SuperClass* присваивается свойству `SubClass.prototype`, а свойство `p` создается в `SubClass.prototype` со значением `undefined` (поскольку значение `p` не было передано конструктору *SuperClass*). В результате в циклах *for-in*, выполняемых со всеми экземплярами *SubClass*, появляется свойство `p` со значением `undefined`. Например:

```
// Конструктор SubClass
function SubClass (p) {
  // Вызвать конструктор надкласса
  super(p);
}
// Конструктор SuperClass
```

```
function SuperClass (p) {
  this.p = p;
}

// Организовать наследование
SubClass.prototype = new SuperClass();

// Создать экземпляр SubClass, не указывая значения p
subInstance = new SubClass();

// Проверить свойства subInstance. Появляется свойство p,
// хотя при вызове конструктора ему не присваивалось значение.
for (var prop in subInstance) {
  // Выводит: property p is: undefined
  trace("Property " + prop + " is: " + typeof subInstance[prop]);
}
```

И хотя некоторых разработчиков появление p во всех экземплярах *SubClass* может раздражать, это нормальное и предполагаемое поведение. Если класс или надкласс определяет свойство, то ожидается, что оно будет доступно всем экземплярам, независимо от того, содержатся ли в нем полезные данные. Однако при желании можно предотвратить назначение экземплярам неопределенных свойств следующим образом:

```
function SuperClass (p) {
  // Создать p, только если оно не undefined
  if (p != undefined) this.p = p;
}
```

Не стоит считать этот прием хорошей практикой ООП, но он решает задачу сокрытия свойства p, когда у него нет значения (как это бывает при вызове конструктора для назначения надкласса).

Пользовательская функция setSuperClass()

Тот, кто решит организовать наследование с помощью __proto__ (несмотря на отрицательные рекомендации Macromedia), может создать пользовательскую функцию, которая скроет несколько сложный синтаксис этого свойства. В примере 12.6 приведена пользовательская функция *setSuperClass( )*, назначающая надкласс для класса (то есть организует наследование) с помощью __proto__.

Пример 12.6. Функция setSuperClass( )

```
// Добавить setSuperClass() в Function.prototype, чтобы
// сделать ее доступной всем объектам функции
Function.prototype.setSuperClass = function (theSuperClass) {
  // Установить связь между надклассом и подклассом.
  this.prototype.__proto__ = theSuperClass.prototype;
}

// Создать надкласс
function Circle (radius) {
  this.r = radius;
}
```

```
  // Создать подкласс
function Ball (radius) {
  // Нельзя использовать super при организации наследования с помощью __proto__
  this.base = Circle;
  this.base(radius);
  delete this.base;
}

// Теперь организовать наследование с помощью нашей
// пользовательской функции setSuperClass()
Ball.setSuperClass(Circle);

// Создать экземпляр Ball
var theBall = new Ball(15);

// Проверить свойство, назначенное классом Circle
trace(theBall.r);  // Выводит: 15
```

Обратите внимание, что вместо «setSuperClass» естественно было бы использовать слово «extends» (в соответствии с терминологией Java), но «extends» зарезервировано в качестве возможного ключевого слова.

Предупреждение для тех, кто применяет `__proto__`: если наследование организовано через `prototype.__proto__`, то оператор *super* во Flash Player 6 нельзя использовать для вызова конструктора надкласса. В этом случае конструктор надкласса надо вызывать с помощью следующего синтаксиса в старом стиле:

```
function SubClass (someArg) {
  this.base = SuperClass;
  this.base(someArg);
  delete this.base;
}
```

Однако после организации наследования через `__proto__` оператор *super* все же можно применять для вызова методов надкласса. Например:

```
SubClass.prototype.hello = function () {
  super.hello();  // Действует даже тогда, когда наследование
                  // установлено с помощью prototype.__proto__
}
```

Как же это проверить?

Изложенные сведения о `prototype`, `constructor` и `__proto__` могут представлять интерес для энтузиастов языков программирования, но все же возникает вопрос, действительно ли необходимо все это знать? Влияет ли все это на то, как надо применять классы и объекты?

Вполне искренне можно сказать, что знать эту информацию не обязательно. Внутренний механизм реализации ООП в языке программирования обычно не интересует пользователей языка. Пользователь может соорудить собственную систему ООП, но эта задача не должна возлагаться на него. Однако ECMAScript (спецификация, на которой основаны ActionScript и JavaScript) первоначально была разработана как облегченный язык сценариев. Требова-

ния, предъявляемые к этой спецификации в настоящее время, в некоторых отношениях выходят за рамки задач, которые она должна была решить. Поэтому разработчики, применяющие ECMAScript, вынуждены были взять дело в собственные руки и попытаться решить проблемы, связанные с реализацией языка, чтобы достичь своих целей. Тем временем идет работа по полному пересмотру системы наследования в ECMAScript 4 и JavaScript 2.0. См.:

http://www.mozilla.org/js/language/js20
http://www.mozilla.org/js/language/es4

На момент написания этой книги спецификация ECMAScript 4 была еще весьма недоработанной, поэтому позиция Macromedia в отношении будущего ECMAScript остается неопределенной.

В любом случае о применении `prototype` и `__proto__` имеет смысл задумываться, только столкнувшись с реальной необходимостью в своей работе. Если обычная система наследования ActionScript не вызывает проблем в ваших приложениях, не стоит прибегать к использованию `__proto__`; однако при возникновении проблем, обусловленных стандартной системой наследования, рассмотрите возможные альтернативы. К счастью, дорога в этом направлении уже частично проложена другими разработчиками. Один из возможных подходов можно найти в статье Дейва Янга (Dave Yang):

http://www.quantumwave.com/flash/inheritance.html

Брэнден Холл (Branden Hall) и Роберт Пеннер (Robert Penner) тоже пишут о наследовании (их книги перечислены в приложении А). Однако по большей части можно работать со стандартными средствами организации наследования (присваивание нового экземпляра надкласса свойству `prototype` подкласса) и не думать о технических тонкостях системы прототипов.

Встроенные классы и объекты ActionScript

Начав с главы 1, мы проделали большой путь. Наше первое знакомство с ActionScript представляло собой изучение элементов панели Actions. После этого мы изучили данные и выражения, операторы, инструкции и функции, а теперь мы рассмотрели понятия классов и объектов. Настало время нанести последние мазки на создаваемую картину языка ActionScript.

ActionScript предоставляет целый ряд синтаксических средств: *выражения* содержат данные; *операторы* обрабатывают данные; *инструкции* содержат предписания; *функции* объединяют предписания в переносимые команды. Все это составляет грамматику ActionScript, но пока наш словарный запас весьма ограничен. Этот пробел восполняется встроенными классами и объектами ActionScript, которые дают массу интересных тем для разговора.

Встроенные классы

Точно так же, как мы определяем собственные классы для описания объектов и действий с ними, ActionScript определяет свои классы данных. Мно-

жество готовых классов, включая класс *Object*, встроены непосредственно в язык ActionScript. Встроенные классы могут управлять физической средой фильма Flash.

Например, один из встроенных классов, *Color*, предоставляет методы, способные определять или задавать цвет клипа. Чтобы использовать эти методы, сначала надо создать объект *Color* с помощью конструктора класса *Color( )*:

```
clipColor = new Color(target);
```

где `clipColor` представляет собой переменную, элемент массива или свойство объекта, в которых хранится объект *Color*. Конструктор *Color( )* принимает один аргумент, `target`, который задает имя клипа, цвет которого надо узнать или установить.

Допустим, что есть клип с именем `square` и требуется изменить его цвет. Создадим новый объект *Color*:

```
squareColor = new Color(square);
```

Для того чтобы задать цвет клипа `square`, вызовем один из методов *Color* с объектом `squareColor`:

```
squareColor.setRGB(0x999999);
```

Метод *setRGB( )* устанавливает RGB-значение цвета для `target` нашего объекта *Color*, которым в данном случае служит `square`. Поэтому приведенный метод делает цвет `square` серым.

Поскольку объект *Color* – это встроенный класс, он может устанавливать цвет клипа, показываемого на экране. Другие классы позволяют управлять звуком, текстовыми полями, кнопками, датами и документами XML. Встроенные классы подробно описаны в «Справочнике по языку».

Встроенные объекты

Подобно встроенным классам, встроенные объекты позволяют управлять окружением фильма. Например, объект *Key* определяет ряд свойств и методов, которые сообщают о состоянии клавиатуры компьютера. Для применения этих свойств и методов объект *Key* создавать не надо, а можно работать с ним непосредственно. Встроенные объекты автоматически создаются интерпретатором при запуске Flash Player и доступны глобально во всем фильме.

Выведем, например, код нажатой в данный момент клавиши с помощью метода *getCode( )* объекта *Key*:

```
trace(Key.getCode());
```

А сейчас проверим, нажата ли клавиша пробела, с помощью метода *isDown( )*, которому передадим в качестве аргумента код клавиши пробела:

```
trace(Key.isDown(Key.SPACE));
```

В «Справочнике по языку» вы найдете данные о других встроенных объектах.

Знание всех ходов и выходов

Умение написать правильный код на ActionScript составляет лишь половину освоения программирования для Flash. Вторая половина состоит в изучении имеющихся встроенных классов и объектов с их многочисленными замечательными свойствами и методами. Этой задачей мы займемся в «Справочнике по языку». Однако не обязательно изучить все классы и объекты в один присест. Начните изучение с клипов, а по мере необходимости переходите к другим темам. Со временем вы узнаете, что является существенным для решения вашей конкретной задачи. Важно понять общую структуру объектно-ориентированного программирования. Если понять правила системы, то изучение нового объекта или класса сводится просто к просмотру имеющихся у него методов и свойств.

Добавление пользовательских методов и свойств во встроенные классы и объекты

В ActionScript уже определены встроенные классы и объекты, но можно придать им новые возможности путем добавления пользовательских методов и свойств точно так же, как мы делаем это с собственными классами и объектами. Можно также строить совершенно новые классы, производные от встроенных.

Добавление метода *randomInt( )* в статический объект *Math*:

```
// Возвращает число в диапазоне от minVal до maxVal включительно
Math.randomInt = function (minVal, maxVal) {
  return minVal + Math.floor(Math.random() * (maxVal + 1 - minVal));
};
```

Добавление метода *scrollToBottom( )* в класс *TextField*:

```
TextField.prototype.scrollToBottom = function () {
  this.scroll = this.maxscroll;
};
```

Создание подкласса класса *XML*:

```
function CustomXML (src) {
  super(src);
}
CustomXML.prototype = new XML();
```

Некоторые встроенные классы требуют выполнить их функцию-конструктор при создании новых объектов. Если возникают сомнения, вызывайте конструктор надкласса, как показано в предыдущих примерах. Учтите, что подклассы можно создавать не для всех встроенных объектов, поскольку некоторые встроенные классы (например, *TextField*, *Button*) привязаны к физическим объектам на экране. Подробные описания классов можно найти в «Справочнике по языку».

Гораздо больше примеров расширения встроенных классов и объектов можно найти на сайте сообщества разработчиков [PROTO]type:

http://www.layer51.com/proto/

Краткий справочник по ООП

В этой главе мы привели много материала. Для повторения и справки ниже приводится беглый обзор технологии ООП, используемой в ActionScript. В конце обзора можно найти шаблон для создания нового ООП-приложения.

Создадим класс:

```
function ClassName (param1, param2) {
  // ...здесь находится код конструктора...
}
```

Создадим свойства экземпляра для класса:

```
function ClassName (param1, param2) {
  // Свойства экземпляра
  this.propName1 = param1;
  this.propName2 = param2;
}
```

Лучше всего создать все свойства, используемые методами класса, в его конструкторе, даже если первоначально у них нет значений:

```
function ClassName (param1, param2) {
  // Объявить все свойства экзкмпляра
  this.propName1 = param1;
  this.propName2 = param2;
  this.propName3 = null;  // Используется в method1();
  this.propName4 = null;  // Используется в method2();
}
```

После создания свойств инициализируем объект:

```
function ClassName (param1, param2) {
  // Объявить все свойства экземпляра
  this.propName1 = param1;
  this.propName2 = param2;

  // Инициализировать объект.
  // ...здесь находится код инициализации...
}
```

Код инициализации можно разместить не в конструкторе, а в методе *init( )* и вызывать его для возврата объекта в состояние по умолчанию:

```
function ClassName (param1, param2) {
  // Объявить все свойства экземпляра
  this.propName1 = null;
  this.propName2 = null;
```

```
    // Инициализировать объект
    this.init.apply(this, arguments);
  }

  ClassName.prototype.init = function (param1, param2) {
    this.prop1 = param1;
    this.prop2 = param2;
  };
```

Создадим метод экземпляра:

```
ClassName.prototype.methodName = function () {
  // ...здесь находится тело метода...
};
```

Для ссылки на экземпляр внутри метода экземпляра применяется ключевое слово `this`:

```
ClassName.prototype.methodName = function () {
  this.propName1 = value;
  this.doSomething();
};
```

Для того чтобы создать статическое свойство, прикрепим его к функции конструктора класса:

```
ClassName.staticPropName = value;
```

Создадим статический метод, прикрепив его к функции конструктора класса:

```
ClassName.staticMethodName = function () {
  // ...здесь находится тело метода...
};
```

Создадим подкласс, присвоив значение свойству `prototype` функции конструктора класса:

```
function SubClassName (param1, param2) {
  // ...здесь находится код конструктора...
}
// СДЕЛАЙТЕ ЭТО ПЕРЕД ОПРЕДЕЛЕНИЕМ МЕТОДОВ SubClassName!
SubClassName.prototype = new SuperClassName();
```

Некоторые советы по ООП

Мы осветили большой теоретический материал, поэтому подытожим некоторые из приведенных соображений, что поможет вам начать применение ООП на практике.

- Определяйте все в виде классов. Все переменные и функции должны быть свойствами и методами класса. Это позволит объединить свойства и методы логически и ссылаться на них через некоторый объект (благодаря чему можно избежать проблем с областями видимости).
- С целью структурирования кода и его повторного использования в разных проектах записывайте каждый класс в отдельный текстовый файл *.as*.

- Определите все классы в `_global`.
- Пользуйтесь моделированием пространства имен. Создайте в `_global` объект, в котором будут храниться все ваши классы, как показано в следующем разделе «Шаблон приложения ООП».
- Создайте функцию *main( )*, которая будет запускать ваше приложение.
- Создавайте все методы экземпляров в объекте `prototype` конструктора, а не в конструкторе.
- Всегда документируйте свои классы, свойства и методы. Пример можно найти в следующем разделе «Шаблон приложения ООП». В документации следует указать, какие методы и свойства предназначены для использования в качестве открытых, а какие – в качестве закрытых внутренних методов класса.

Шаблон приложения ООП

Иногда гибкость ActionScript из достоинства превращается в недостаток, оборачиваясь ненадежностью. Способов создать допустимое объектно-ориентированное приложение так много, что поиск стандартного подхода становится тяжелой задачей. В приводимом ниже образце шаблона объектно-ориентированного приложения предпринимается попытка дать некоторое начальное направление. Этот образец читатели могут адаптировать к своим потребностям, а электронную версию шаблона можно найти в сетевом хранилище кода.

Шаблон основного приложения

Следующий код надо прикрепить к главной временной диаграмме *yourApp.fla*, где *yourApp.fla* – главный файл исходного кода вашего приложения Flash. Кадр, к которому вы прикрепите этот код, должен быть предварительно загружен (о предзагрузке см. *MovieClip*.`_framesloaded` в «Справочнике по языку»).

```
// ========================================
// ЗАГРУЗКА КЛАССОВ
// ========================================

// Создать пространство имен для хранения классов
if (_global.com == undefined) {
  _global.com = new Object();
}
if (_global.com.yourDomain == undefined) {
  _global.com.yourDomain = new Object();
}
// Загрузить внешние файлы .as, по одному для каждого класса
#include "YourClass.as"
#include "YourSubClass.as"
```

```
// ========================================
// ЗАПУСК ПРИЛОЖЕНИЯ
// ========================================
main();

// Функция: main()
//     Описание: Единственная точка входа вашего приложения
function main () {
  // Создание и использование экземпляров ваших классов
  var yourInstance = new com.yourDomain.YourClass(arg1, arg2);
  var yourSubInstance = new com.yourDomain.YourSubClass(arg1, arg2, arg3, arg4);
  yourInstance.yourMethod();
  yourSubInstance.yourSubMethod();
  yourSubInstance.yourMethod();
}
```

Шаблон класса

Следующий код надо сохранить в файле с именем *YourClass.as*. В этом примере *YourClass.as* и *yourApp.fla* располагаются в одном и том же каталоге.

```
/*
 * YourClass Класс
 *   Версия: 1.0.0
 *   Описание: Здесь ваше описание.
 *
 * Параметры конструктора:
 *   param1             -Краткое описание
 *   param2             -Краткое описание
 *
 * Методы:
 *   yourMethod()       -Краткое описание
 *
 * Статические свойства
 *   staticProp1        -Краткое описание
 *   staticProp2        -Краткое описание
 *
 * Статические методы
 *   staticMeth()       -Краткое описание
 */

/*
 * Конструктор класса
 */
_global.com.yourDomain.YourClass = function (param1, param2) {
  // Создать свойства экземпляров
  this.prop1 = param1;  // Краткое описание. Тип: состояние тип_данных
  this.prop2 = param2;  // Краткое описание. Тип: состояние тип_данных
  this.prop3 = null;    // Краткое описание. Тип: состояние тип_данных

  // Инициализировать объект
  // ...здесь находится код инициализации...
  // Постарайтесь поместить код инициализации в метод init()
};
```

```
/*
 * Статические свойства
 */
com.yourDomain.YourClass.staticProp1 = value;
com.yourDomain.YourClass.staticProp2 = value;

/*
 * Статические методы
 */
com.yourDomain.YourClass.staticMeth = function () {

};

/*
 * Методы экземпляра
 */

/*
 * Метод: YourClass.yourMethod()
 *   Описание: Краткое описание.
 *
 * Параметры:
 *   param1             -Краткое описание
 */
com.yourDomain.YourClass.prototype.yourMethod = function (param1) {
  // Тело метода...
  trace("yourMethod invoked");
};
```

Шаблон подкласса

Следующий код надо сохранить в файле с именем *YourSubClass.as*. В этом примере *YourSubClass.as* и *yourApp.fla* располагаются в одном и том же каталоге.

```
/*
 * YourSubClass класс. Расширяет YourClass.
 *   Версия: 1.0.0
 *   Описание: здесь ваше описание.
 *
 * Параметры конструктора:
 *   param1             -Параметр надкласса
 *   param2             -Параметр надкласса
 *   param3             -Краткое описание
 *   param4             -Краткое описание
 *
 * Методы:
 *   yourSubMethod()  -Краткое описание
 */

/*
 * Конструктор класса
 */
_global.com.yourDomain.YourSubClass = function (param1, param2, param3, param4) {
  // Вызвать конструктор надкласса.
  super(param1, param2);
```

```
  // Создать свойства экземпляра.
  this.subProp1 = param3;  // Краткое описание. Тип: состояние тип_данных
  this.subProp2 = param4;  // Краткое описание. Тип: состояние тип_данных
  this.subProp3 = null;    // Краткое описание. Тип: состояние тип_данных

  // Инициализировать объект.
  // ...здесь находится код инициализации...
  // Постарайтесь поместить код инициализации в метод init()
};

/*
 * Установить наследование. ПЕРЕД ОПРЕДЕЛЕНИЕМ МЕТОДОВ!
 */
com.yourDomain.YourSubClass.prototype = new com.yourDomain.YourClass();

/*
 * Методы экземпляра
 */

/*
 * Метод: YourSubClass.yourSubMethod()
 *   Описание: Краткое описание.
 *
 * Параметры:
 *   param1             -Краткое описание
 */
com.yourDomain.YourSubClass.prototype.yourSubMethod = function (param1) {
  // Тело метода...
  trace("yourSubMethod invoked");
};
```

Дополнительные темы

Объем книги не позволяет досконально обсудить следующие темы. Рекомендуется рассмотреть их в дополнительных источниках информации, которые здесь приводятся.

Пользовательские события

Вы могли обратить внимание, что мы изучили создание методов и свойств, но не обработчиков и приемников событий. В ActionScript нет документированных способов определения пользовательских событий для класса. Однако недокументированный метод *ASBroadcaster* предоставляет внутреннюю поддержку пользовательских событий. Подробности можно найти здесь:

http://chattyfig.figleaf.com/flashcoders-wiki/index.php?AsBroadcaster

Кроме того, Брэнден Холл (Branden Hall) разработал машину пользовательских событий под названием FLEM. См.:

http://chattyfig.figleaf.com/flem/
http://chattyfig.figleaf.com/flem/extend.php

Паттерны проектирования и UML

Освоившись с синтаксисом и методологией объектно-ориентированного программирования, вы в конечном счете почувствуете, что сложнее всего не технически реализовать, а спроектировать классы, представляющие то или иное приложение. В области ООП возникли устойчивые конструкции для однотипных задач, получившие название «шаблонов», или «паттернов» проектирования (design patterns). Документации по паттернам проектирования, относящимся конкретно к ActionScript, немного, но паттерны проектирования можно применять в любом языке ООП. Отважным разработчикам ActionScript, готовым потратить некоторое время на изучение Java и/или C++, рекомендуются книги:

«Design Patterns Explained: A New Perspective on Object-Oriented Design», Shalloway and Trott, Addison Wesley, 2001.[1]

«Design Patterns: Elements of Reusable Object-Oriented Software», Gamma et al, Addison Wesley, 1995.[2]

Обычно паттерны проектирования изображаются графически с помощью особого языка диаграмм, носящего название UML (Unified Modeling Language, унифицированный язык моделирования). UML является стандартом графического отображения иерархий классов. Информацию о нем можно найти здесь:

http://www.omg.org/gettingstarted/what_is_uml.htm
http://www.devx.com/uml/

«UML Distilled: A Brief Guide to the Standard Object Modeling Language», Fowler and Scott, Addison Wesley, 1999.[3]

Моделирование пространств имен

Пространство имен – это исключительная область программы, в которой гарантируется уникальность всех идентификаторов. Благодаря пространству имен в программе можно работать, скажем, с уникальными классами *Ball* как фирмы X, так и фирмы Y, избегая конфликтов между этими классами (то есть не допуская, чтобы объявление одного класса перекрыло другое). В ActionScript для Flash MX нет формальной поддержки пространств имен, но их можно моделировать, создав уникальный глобальный объект и определив в нем все идентификаторы программы (как было показано в разделе «Шаблон програм-

---

[1] Шаллоуей А., Тротт Д. «Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию». – Пер. с англ. – СПб: Питер, 2002.

[2] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб: Питер, 2001.

[3] Фаулер М., Скотт К. «UML. Основы», 2-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2001.

мы ООП»). Обычно уникальному глобальному объекту дается имя домена, поскольку уникальность имен доменов гарантирована. Например, пространство имен для своих классов я создаю так (домен моего сайта – *moock.org*):

```
if (_global.org == undefined) {
  _global.org = new Object();
}
if (_global.org.moock == undefined) {
  _global.org.moock = new Object();
}
```

Этот код есть во всех моих классах (каждый класс обязан создать пространство имен, если оно еще не существует). Вот удобная функция *AsSetupPackage( )*, с помощью которой можно создать пространство имен для своих классов:

```
_global.AsSetupPackage = function (path) {
  var a = path.split('.');
  var o = _global;
  for (var i = 0; i < a.length; i++) {
    var name = a[i];
    if (o[name] == undefined) {
      o[name] = new Object();
    }
    o = o[name];
  }
}
```

С помощью функции *AsSetupPackage( )* в каждом классе можно создать пространство имен класса::

```
AsSetupPackage("com.yourdomain");
```

Например, я делаю так:

```
AsSetupPackage("org.moock");
```

Вопросник с вариантами ответов в стиле ООП

Итак, вы познакомились с принципами ООП, и мы можем вернуться к нашему примеру вопросника с вариантами ответов и посмотреть, как можно применить эти принципы. Исходный код для нашего ООП-вопросника слишком велик, чтобы привести его здесь, но его можно загрузить из хранилища кода. Важной частью приложения является его структура объектов, приводимая ниже. Даже не видя кода вопросника, по следующим таблицам, в которых описаны классы, методы, свойства и события вопросника, можно получить полную картину работы приложения. На самом деле на основе предлагаемой структуры можно попытаться построить вопросник самостоятельно, не загружая его исходный код. Обратите внимание, что некоторые свойства не приведены в таблицах, поскольку они применяются только в методах класса и потому считаются внутренними («закрытыми») свойствами.

Наш вопросник состоит из трех классов, показанных в табл. 12.2. Весь вопросник полностью генерируется в ActionScript – в библиотеке фильма нет

ни одного объекта. Все кнопки изображаются средствами Drawing API, а весь текст создается с помощью *MovieClip.createTextField( )*, что обеспечивает вопроснику гибкость: внешнй вид адаптируется к текущим размерам Player, длине вопроса и длине ответа. Кнопки навигации – Next, Back, Grade и Try Again – дают пользователю возможность проходить вопросник в нелинейном направлении, изменять уже данный ответ, получать оценку и предпринимать повторные попытки пройти вопросник.

Таблица 12.2. Классы вопросника с вариантами ответов

Имя класса	Описание класса
Quiz	Управляет данными вопросника (вопросами и ответами) Выполняет загрузку вопросов из внешнего файла XML Запускает вопросник и следит за текущим вопросом Оценивает ответы пользователя
QuizGUI	Показывает вопросник на экране и принимает данные, вводимые пользователем
Question	Управляет данными отдельных вопросов и ответами на них

Класс *Quiz* определяет методы и свойства, перечисленные в табл. 12.3.

Таблица 12.3. Класс Quiz

Имя	Тип	Описание
loadQuizData( )	Метод	Загружает файл XML и на его основе генерирует вопросник
startNewQuiz( )	Метод	Приводит вопросник в начальное состояние и сообщает *QuizGUI*, что пора начать вопросник
setQuizGUI	Метод	Сохраняет ссылку на экземпляр *QuizGUI*, который будет показывать вопросник на экране
setTitle( )	Метод	Сохраняет заголовок вопросника
getTitle( )	Метод	Возвращает заголовок вопросника
setDescription( )	Метод	Сохраняет краткий текст с описанием вопросника
getDescription( )	Метод	Возвращает описание вопросника
getScore( )	Метод	Рассчитывает и возвращает текущий результат пользователя, исходя из вопросов, на которые уже дан ответ
setCurrent-Question( )	Метод	Устанавливает текущий вопрос
getCurrent-Question( )	Метод	Возвращает номер текущего вопроса
getNumUser-Answers( )	Метод	Возвращает количество вопросов, на которые уже дан ответ

Имя	Тип	Описание
setNumUser-Answers( )	Метод	Устанавливает количество вопросов, на которые уже дан ответ
setUserAnswer( )	Метод	Сохраняет ответ пользователя на конкретный вопрос
addQuestion( )	Метод	Вводит в вопросник новый объект *Question*
getQuestionAt( )	Метод	Возвращает конкретный объект *Question* из перечня вопросов
getNumQuestions( )	Метод	Возвращает количество вопросов в вопроснике
isComplete( )	Метод	Возвращает `true`, если вопросник считается завершенным; в противном случае возвращает `false`. В простейшем случае вопросник считается завершенным, если пользователь ответил на все вопросы
`quizGUI`	Свойство	Ссылка на экземпляр `quizGUI`, который будет показывать вопросник

Класс *QuizGUI* определяет методы, свойства и события, перечисленные в табл. 12.4. Обратите внимание, что обработчики событий в классе *QuizGUI* правильнее было бы определить как события приемника *Quiz*. Однако поскольку в ActionScript нет встроенных средств для создания событий приемника, мы упростили код, поместив обработчики непосредственно в *QuizGUI*.

Таблица 12.4. Класс QuizGUI

Имя	Тип	Описание
displaySplash-Scr een( )	Метод	Выводит экран приветствия вопросника
displayQuestion( )	Метод	Полностью показывает вопрос, включая ответ и кнопки навигации
displayQuizEnd( )	Метод	Показывает окончательный отчет о выполнении вопросника, включая результат пользователя
displayTitle( )	Метод	Выводит на экран заголовок вопросника
createNavButton ( )	Метод	Вспомогательная функция, которая создает и возвращает кнопку навигации
clearScreen( )	Метод	Удаляет с экрана все отображаемые элементы
onQuizStart( )	Обработчик события	Вызывается в начале вопросника
onQuestionUp-date( )	Обработчик события	Вызывается, когда изменяется содержимое вопроса, пользователь вводит ответ или меняется номер текущего вопроса
onQuizComplete( )	Обработчик события	Вызывается, когда пользователь ответил на все вопросы

Таблица 12.4 (продолжение)

Имя	Тип	Описание
quizObj	Свойство	Ссылка на экземпляр *Quiz*, отображением которого занимается данный *QuizGUI*
quizClip	Свойство	Ссылка на клип, в котором будут созданы отображаемые элементы вопросника

Класс *Question* определяет методы, перечисленные в табл. 12.5. Конструктор *Question* может получать данные вопроса в виде ряда значений или одного фрагмента XML.

Таблица 12.5. Класс Question

Имя	Тип	Описание
setQuestionTex t( )	Метод	Устанавливает текст вопроса
getQuestionTe xt( )	Метод	Возвращает текст вопроса
setAnswers( )	Метод	Записывает варианты ответов в массив
getAnswers( )	Метод	Возвращает массив с вариантами ответов
setCorrectAnswer( )	Метод	Записывает правильный ответ на вопрос в виде индекса в массиве ответов
getCorrectAnswer( )	Метод	Возвращает правильный ответ на вопрос в виде индекса в массиве ответов
setUserAnswer( )	Метод	Записывает ответ пользователя на вопрос в виде индекса в массиве ответов. Может присваиваться несколько раз
getUserAnswer( )	Метод	Возвращает текущий ответ пользователя на вопрос в виде индекса в массиве ответов

Вперед!

Классы и наследование открывают целый мир возможностей для совместного использования информации многими объектами. Объектно-ориентированное программирование представляет собой основу всего кода в ActionScript. Независимо от того, в какой мере вы используете возможности классов и объектов в своих сценариях, понимание их общих принципов составляет важную часть понимания программной среды Flash. Уверенное объектно-ориентированное программирование в значительной мере влияет на уверенность работы с ActionScript в целом.

Дополнительные сведения об объектно-ориентированном программировании в языках на основе ECMA-262 представлены в документации Netscape по JavaScript, «Details of the Object Model», по адресу:

http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm

В классической книге Дэвида Флэнагана (David Flanagan) «JavaScript: The Definitive Guide» (O'Reilly)[1] также содержатся ценные сведения об ООП в JavaScript.

Для общего знакомства с ООП с точки зрения Java обратитесь к руководству «Object-Oriented Programming Concepts» фирмы Sun (из «The Java™ Tutorial»):

http://java.sun.com/docs/books/tutorial/java/concepts

Полное электронное руководство по ООП в ActionScript представлено в учебнике Робин Дебройль (Robin Debreuil):

http://www.debreuil.com/docs/

Для интересующихся приводим ссылку на большую группу статей, в которых делается попытка развенчать сильно разрекламированную систему ООП. Независимо от степени вашего согласия с автором эта дискуссия может оказаться полезной.

http://www.geocities.com/tablizer/oopbad.htm

Конечно, мы прошли большой путь, и я надеюсь, что вы, как и я, не утратили энтузиазма. Теперь мы займемся изучением самого важного класса объектов, имеющегося в ActionScript, – класса *MovieClip*.

[1] Д. Флэнаган «JavaScript. Подробное руководство», 4-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2004.

13

Клипы

Каждый документ Flash содержит *сцену* (*stage*), на которой помещаются фигуры, текст и другие видимые элементы, а также временная диаграмма, с помощью которой задается изменение содержимого сцены в зависимости от времени. Основная сцена (то есть *главный фильм*, *main movie*) может содержать независимые вложенные фильмы, называемые *клипами* (*clips*). У каждого клипа есть собственная независимая временная диаграмма и *холст* (*canvas*); он даже может содержать другие клипы. (Сцена является холстом для главного фильма.) Клип, который содержится в другом клипе, называется *вложенным* (*nested*), а клип, содержащий в себе другой клип, называется *клипом-хозяином* (*host clip*), или *родительским клипом* (*parent clip*), для вложенного клипа.

Документ Flash может содержать иерархию взаимосвязанных клипов. Например, главный фильм представляет горный ландшафт. В отдельном клипе находится анимация персонажа, который можно перемещать по ландшафту, создавая впечатление ходьбы. Еще один независимый клип внутри клипа персонажа может быть использован для анимации моргающих глаз. Когда независимые элементы персонажа мультипликации воспроизводятся вместе, они кажутся единым элементом контента. Кроме того, компоненты могут «осмысленно» взаимодействовать друг с другом: можно сделать так, чтобы глаза моргали, когда персонаж останавливается, или ноги изображали движение, когда персонаж начинает перемещаться.

ActionScript обеспечивает детальный контроль над клипами: можно воспроизводить клип, останавливать его, перемещать головку воспроизведения по его временной диаграмме, программно устанавливать его свойства (такие как размер, угол поворота, степень прозрачности и положение на сцене) и работать с ним, как с настоящим программным объектом. В качестве формальной составляющей языка ActionScript клипы можно рассматривать как исходный материал для создания программно порождаемого содержимого Flash. Например, клип может выступать в роли шарика или ракетки в игре в теннис, в роли бланка заказа в каталоге веб-сайта или просто как контейнер для фонового звука мультипликации.

«Объектность» клипов

Начиная с Flash 5 с клипами можно работать как с объектами, о которых рассказывалось в главе 12. Можно получать и устанавливать свойства клипа, а также вызывать встроенные или пользовательские методы клипа. Производимые с клипами операции могут иметь в проигрывателе Flash видимые или слышимые результаты.

Фактически клипы – не совсем объекты, но похожи на них. Хотя клипы нельзя создавать с помощью конструктора класса или использовать для создания клипа литерал объекта, можно создавать экземпляры клипов с помощью методов *attachMovie( )*, *duplicateMovieClip( )* и *createEmptyMovieClip( )* (появившегося во Flash Player 6). Так что же такое клипы, если не объекты? Это члены самостоятельного, похожего на объекты типа данных под названием movieclip (в этом можно убедиться, выполнив над клипом *typeof* и получив в результате строку «movieclip»). Главное отличие между клипами и другими объектами заключается в том, каким образом для них выделяется (при создании) и освобождается (при удалении) память. Подробности см. по адресу:

http://www.moock.org/asdg/technotes/movieclipDatatype

Однако если не считать этих технических особенностей, работа с клипами почти всегда происходит точно так же, как с объектами.

Какое же влияние «объектность» клипов оказывает на работу с ними в ActionScript? Прежде всего, она позволяет вызывать методы клипов и использовать их свойства, как мы это делаем с другими объектами. Клипами можно непосредственно управлять с помощью встроенных методов, например:

```
eyes_mc.play();
```

Свойства клипов можно извлекать и устанавливать с помощью оператора точки, как это делается для любого объекта:

```
ball_mc._xscale = 90;
var radius = ball_mc._width / 2;
```

Переменная в клипе является просто свойством этого клипа, и с помощью оператора точки значения переменных можно устанавливать и извлекать:

```
theClip_mc.someVariable = 14;
x = theClip_mc.someVariable;
```

Вложенные клипы можно рассматривать как свойства родительского клипа, имеющие значением объект. Поэтому для доступа к вложенным клипам используется оператор точки:

```
clipA.clipB.clipC.play();
```

Для ссылки на клип, содержащий текущий клип, используется зарезервированное свойство `_parent`:

```
_parent.clipC.play();
```

Обращение с клипами как с объектами предоставляет все преимущества удобного синтаксиса и гибкого управления воспроизведением. Но использование клипов как объектов не препятствует также работе с ними как с данными: можно сохранить ссылку на клип в элементе массива или переменной и даже передать ссылку на клип в функцию в виде аргумента. Вот, например, функция, которая перемещает клип в определенную точку экрана:

```
function moveClipTo (clip, x, y) {
  clip._x = x;
  clip._y = y;
}
moveClipTo(ball_mc, 14, 399);
```

В этой главе будет подробно рассказано, как создавать ссылки на клипы, управлять ими и обрабатывать их как объекты данных.

Класс MovieClip

Все отдельные клипы являются экземплярами класса *MovieClip*, который определяет свойства, методы и обработчики событий, поддерживаемые клипами. С классом *MovieClip* можно обращаться так же, как с другими встроенными классами. Мы можем замещать его методы или добавлять новые с помощью приемов, рассмотренных в главе 12. Например, следующий код добавляет новый метод *getArea( )* во все клипы:

```
MovieClip.prototype.getArea = function () {
  return this._width * this._height;
};
```

Как мы увидим в главе 14, во Flash MX класс *MovieClip* можно даже использовать в качестве надкласса для новых классов (то есть новые классы можно делать производными от класса *MovieClip*). Полное описание всех свойств, методов и обработчиков событий, поддерживаемых классом *MovieClip*, см. в «Справочнике по языку».

Типы клипов

Не все клипы одинаковы. Фактически во Flash есть три разных типа клипов:

- Главные фильмы
- Обычные клипы
- Компоненты (во Flash 5 они назывались *«умными» (интеллектуальными) клипами – Smart Clips*)

Помимо этих трех официальных разновидностей можно выделить еще четыре подкатегории обычных клипов:

- Клипы обработки (process clips)
- Клипы сценариев

- Связанные клипы
- Порождающие клипы (seed clips)

Эти неофициальные категории не входят в число формальных терминов, употребляемых в ActionScript, но ими удобно пользоваться при программировании с помощью клипов. Познакомимся поближе с каждым типом клипа.

Главные фильмы

Главный фильм (*main movie*) документа Flash – это главная временная диаграмма и сцена, присутствующие в каждом документе *.swf*. Главный фильм служит фундаментом для всего содержимого документа, включая все остальные клипы. Иногда главный фильм называют *главной временной диаграммой* (*main timeline*), *временной диаграммой главного фильма* (*main movie timeline*), *главной сценой* (*main Stage*), или просто *корнем* (*root*).

Обратите внимание, что хотя файл *.swf* содержит только один главный фильм, во Flash Player может одновременно находиться несколько файлов *.swf* – мы можем загрузить несколько документов *.swf* (и, следовательно, несколько главных фильмов) в стек *уровней*, воспользовавшись функциями *loadMovie( )* и *unloadMovie( )*, которые мы рассмотрим ниже.

С главными фильмами можно обращаться по большей части как с обычными клипами, за следующими исключениями:

- Главный фильм нельзя удалить из файла *.swf* (хотя сам файл *.swf* можно удалить из проигрывателя Flash).
- Следующие методы клипов не работают при вызове с главным фильмом: *duplicateMovieClip( )*, *removeMovieClip( )* и *swapDepths( )*.
- Главные фильмы не поддерживают следующие свойства: `enabled`, `focusEnabled`, `hitArea`, `_name`, `_parent`, `tabChildren`, `tabEnabled`, `tabIndex`, `trackAsMenu` и `useHandCursor`.
- Во Flash 5 к главному фильму нельзя прикреплять обработчики событий. Во Flash Player 6 к главному фильму можно прикреплять следующие обработчики событий: *onData( )*, *onEnterFrame( )*, *onKeyDown( )*, *onKeyUp( )*, *onMouseDown( )*, *onMouseMove( )* и *onMouseUp( )*.
- Главные фильмы не могут получать фокус для ввода с клавиатуры.
- Ссылаться на главный фильм можно посредством встроенных глобальных свойств `_root` и `_level`*n*.

Обычные клипы

Обычные клипы (*regular movie clips*) служат самыми распространенными и базовыми контейнерами содержимого: они хранят видимые элементы и звуки и даже могут реагировать на действия пользователя и воспроизведение фильма посредством обработчиков событий. Для программистов JavaScript, привыкших к работе с DHTML, удобным может оказаться представление главного фильма как аналогии объекта документа HTML, а обычных клипов – как аналогий объектов слоев этого документа.

Компоненты

Компоненты Flash MX (components) представляют собой усовершенствованные «интеллектуальные клипы» Flash 5. Это клипы, которые содержат графический интерфейс пользователя, применяемый для настройки свойств клипа в среде разработки. Обычно компоненты разрабатываются опытными программистами для распространения в качестве независимых модулей, таких как выпадающие меню или ползунки. Компоненты также предоставляют менее опытным Flash-разработчикам возможность настроить поведение клипа, не зная, как работает его код. Подробно о компонентах мы расскажем в главах 14 и 16. Flash MX поставляется с набором готовых компонентов интерфейса, который называется Flash UI Components. Для доступа к Flash UI Components выберите пункт меню Window → Components.

Клипы обработки

Клип обработки (*process clip*) – это клип, используемый не ради содержания, а только для многократного выполнения блока кода. Клипы обработки можно строить с обработчиком события *enterFrame( )* или с циклом временной диаграммы, как было показано в разделе «Циклы временной диаграммы и событий клипа» главы 8.

Во Flash MX функциональность клипов обработки можно частично заменить применением функции *setInterval( )*, которая периодически выполняет функцию или метод. Код, выполняемый по таймеру, можно реализовать с помощью *setInterval( )*, а код, синхронизируемый именно с частотой кадров, следует реализовывать с помощью клипа обработки. См. описание *setInterval( )* в «Справочнике по языку».

Клипы сценариев

Как и клип обработки, *клип сценария* (*script clip*) представляет собою пустой клип, используемый не ради содержания, а для слежения за некоторой переменной, определения некоторого класса или просто выполнения некоторого произвольного сценария. Например, во Flash 5 клип сценария позволяет организовать хранение обработчиков событий, реагирующих на нажатия клавиш или события, связанные с мышью. Во Flash MX с помощью клипов сценариев можно создавать так называемые «чисто программные» компоненты.

Связанные клипы

Связанный *клип* (*linked clip*) выполняет экспорт из библиотеки фильма или импорт в нее. Доступ к настройкам экспорта и импорта осуществляется через опцию Linkage каждого клипа, которая находится в библиотеке. Чаще всего связанные клипы применяются для динамического генерирования экземпляра клипа непосредственно из символа библиотеки с помощью метода *attachMovie( )*, что будет показано ниже.

Порождающие клипы

До того как во Flash 5 был введен метод *attachMovie( )*, для создания новых клипов на основе некоторого существующего клипа, называемого *порождающим клипом* (*seed clip*), использовалась функция *duplicateMovieClip( )*. *Порождающий клип* находится на сцене исключительно для того, чтобы копироваться посредством *duplicateMovieClip( )*. С введением метода *attachMovie( )*, который создает экземпляры из клипов в библиотеке, потребность в порождающих клипах уменьшилась. Однако порождающие клипы и *duplicateMovieClip( )* по-прежнему применяются, если в процессе копирования надо сохранить преобразования и обработчики событий *onClipEvent( )*.

В фильме, интенсивно использующем *duplicateMovieClip( )* для динамического генерирования содержимого, часто можно увидеть ряд порождающих клипов на краю холста фильма. Порождающие клипы используются только для производства клипов-дубликатов и потому хранятся за пределами видимой области сцены.

Создание клипов

Обычно с клипами обращаются точно так же, как с объектами данных: их свойства устанавливаются с помощью оператора точки, их методы вызываются с помощью оператора вызова функции (круглые скобки), и хранятся они в переменных, элементах массивов и свойствах объектов. Однако создание клипов происходит не так, как создание объектов данных. Нельзя описать клип в коде в виде литерала, как можно описать объект посредством литерала объекта. Нельзя также создать клип с помощью оператора *new*:

```
myClip = new MovieClip();  // Неплохо, неплохо, да только ничего не выйдет
```

Хотя Flash Player 6 поддерживает команду *new MovieClip( )*, последняя создает подкласс *MovieClip* и *не может быть использована для создания нового экземпляра клипа в фильме*. Вместо этого клипы обычно создаются прямо в среде разработки, вручную. Когда клип создан, можно создавать его новые независимые дубликаты с помощью таких команд, как *duplicateMovieClip( )* и *attachMovie( )*. Во Flash Player 6 можно также создавать совершенно новый клип на этапе выполнения с помощью метода *createEmptyMovieClip( )*.

Символы и экземпляры клипов

Так же как все экземпляры объектов основываются на классе, все экземпляры клипов основываются на клипе-шаблоне, называемом *символом* (*symbol*), или иногда *определением* (*definition*). Символ выступает в качестве модели содержимого и структуры клипа. За исключением клипов, создаваемых с помощью *createEmptyMovieClip( )*, конкретные экземпляры клипа генерируются из символов клипов, находящихся в библиотеке. С помощью символа можно как вручную, так и программным образом создавать клипы, которые должны выводиться в фильме.

Конкретная копия символа клипа называется *экземпляром* (*instance*). Экземпляры являются отдельными объектами клипов, которые можно обрабатывать с помощью ActionScript; *символ* служит формой, с помощью которой отливаются все экземпляры определенного клипа. Символы клипов создаются в среде разработки Flash. Чтобы создать новый пустой символ, выполните следующие действия:

1. Выберите пункт меню Insert → New Symbol. Появится диалоговое окно Create New Symbol.
2. В поле Name введите идентификатор символа.
3. Чтобы задать поведение, выберите переключатель Movie Clip.
4. Щелкните по кнопке OK.

Как правило, после этого холст и временная диаграмма символа заполняются содержимым клипа. После своего создания символ хранится в библиотеке, ожидая использования при создании экземпляра клипа. Однако имеется также возможность преобразовать в символ группу фигур и объектов, уже существующих на сцене. Для этого выполните следующие шаги:

1. Выделите нужные фигуры и объекты.
2. Выберите пункт меню Insert → Convert to Symbol.
3. В поле Name введите идентификатор символа.
4. Чтобы задать поведение, выберите переключатель Movie Clip.
5. Щелкните по кнопке OK.

Выделенные для создания нового символа клипа фигуры и объекты заменяются безымянным экземпляром этого нового клипа. Соответствующий символ клипа появляется в библиотеке, готовый к использованию для создания новых экземпляров.

Создание экземпляров

Существует четыре способа создания нового экземпляра клипа. Три из них программные, а последний является сугубо ручным и используется в среде разработки Flash. Все методы, кроме *createEmptyMovieClip( )*, требуют, чтобы существовал символ клипа, от которого должен быть порожден новый экземпляр.

Создание экземпляров вручную

Экземпляры клипов могут создаваться вручную с помощью библиотеки в редакторе Flash. При физическом перетаскивании символа клипа из библиотеки на сцену создается новый экземпляр. Созданному таким способом экземпляру надо вручную дать имя, задав его в панели Instance (Flash 5) или инспекторе свойств (Flash MX). Об именах экземпляров мы еще расскажем в этой главе. Если вам не доводилось работать во Flash с клипами, обратитесь к разделу «Using Symbols, Instances, and Library Assets» в справочной системе Macromedia Flash.

Создание экземпляров с помощью duplicateMovieClip()

Любой экземпляр, уже находящийся на сцене фильма Flash, можно копировать с помощью ActionScript. После этого с новым экземпляром можно обращаться как с отдельным, совершенно независимым клипом. Дублировать можно экземпляры клипов, созданные как вручную, так и программным образом. Иными словами, можно дублировать дубликаты.

Существует два способа дублирования экземпляров с помощью *duplicateMovieClip( )*:

- Можно вызвать *duplicateMovieClip( )* в качестве *глобальной функции*, используя синтаксис:

  ```
  duplicateMovieClip(target, newName, depth);
  ```

  Здесь *target* – это строка, задающая имя экземпляра, который требуется дублировать, *newName* – строка, задающей идентификатор нового экземпляра, а *depth* представляет собой целое число, определяющее место в стеке содержимого, куда должен быть помещен новый экземпляр (о стеке содержимого мы вскоре расскажем).

- Возможен также вызов *duplicateMovieClip( )* в качестве метода существующего экземпляра:

  ```
  theClip.duplicateMovieClip(newName, depth);
  ```

  Здесь *theClip* является именем клипа, дубликат которого мы желаем создать, а *newName* и *depth* означают то же, что в предыдущем примере.

Если клип создан с помощью функции *duplicateMovieClip( )*, он прикрепляется в качестве дочернего к родительскому объекту *target* или *theClip*. Таким образом, новый клип помещается в иерархии клипов на одном уровне с *target* или *myClip*.

Если экземпляр создан с помощью *duplicateMovieClip( )*, то первоначально он помещается прямо поверх своего порождающего клипа. Поэтому первой задачей после дуплицирования является перемещение дублированного клипа в новое положение. Например:

```
ball_mc.duplicateMovieClip("ball2_mc", 0);
ball2_mc._x += 100;
ball2_mc._y += 50;
```

Дублированные экземпляры, порождающие клипы которых преобразованы с помощью ActionScript или вручную в среде разработки Flash (например, изменены цвет, угол поворота или размер), наследуют преобразование своих порождающих клипов в момент дублирования. Последующие преобразования порождающего клипа не влияют на продублированные экземпляры. Аналогично каждый экземпляр можно изменить независимым образом. Например, если порождающий клип повернуть на 45°, а затем создать его дубликат, начальным углом поворота дубликата будет 45°:

```
seedClip_mc._rotation = 45;
```

```
seedClip_mc.duplicateMovieClip("newClip_mc", 0);
trace(newClip_mc._rotation);  // Выводит: 45
```

Если после этого повернуть экземпляр дубликата на 10°, угол поворота составит 55°, но угол поворота порождающего клипа по-прежнему останется равным 45°:

```
newClip_mc._rotation += 10;
trace(newClip_mc._rotation);      // Выводит: 55
trace(seedClip_mc._rotation);     // Выводит: 45
```

Путем дублирования нескольких экземпляров подряд и незначительного преобразования каждого дубликата можно достичь интересных сложных эффектов, таких как звездное небо, след и геометрические анимации (см. пример «Load Event Starfield» в сетевом хранилище кода).

Применение метода *duplicateMovieClip( )* дает и другие преимущества перед размещением клипов в фильме вручную, например следующие возможности:

- Точно управлять появлением клипа на сцене в ходе выполнения программы.
- Точно управлять удалением клипа со сцены в ходе выполнения программы.
- Копировать обработчики событий клипа.

Такие возможности предоставляют развитый программный контроль над содержимым фильма. Для клипа, созданного вручную, создание и исчезновение должны быть предопределены с помощью временной диаграммы.

Создание экземпляров с помощью attachMovie()

Подобно *duplicateMovieClip( )*, метод *attachMovie( )* позволяет создать экземпляр клипа на этапе выполнения; однако *attachMovie( )* создает экземпляр из символа библиотеки, а не из другого экземпляра клипа. «Прикрепить» (attach) клип – значит сделать клип дочерним для главной временной диаграммы или другого клипа в иерархии клипов. Чтобы использовать *attachMovie( )* для создания экземпляра символа, необходимо сперва *экспортировать* этот символ из библиотеки. Вот как это делается:

1. Выбрать в библиотеке символ.
2. Во всплывающем меню Options библиотеки выбрать Linkage. Появляется диалоговое окно Linkage Properties.
3. Выставить флажок Export For ActionScript.
4. В поле Identifier ввести уникальное имя символа клипа. Имя может быть любой строкой (часто оно совпадает с именем самого символа), но должно отличаться от всех других экспортированных символов.
5. Во Flash MX можно также задать кадр, в который клип связи будет экспортирован с фильмом. Подробности см. в описании *MovieClip.attachMovie( )* в «Справочнике по языку».
6. Щелкнуть по кнопке OK.

Новые экземпляры экспортированного символа можно прикреплять к существующему клипу путем вызова *attachMovie( )* со следующим синтаксисом:

```
theClip.attachMovie(symbolIdentifier, newName, depth, [initObject]);
```

Здесь *theClip* представляет собой имя клипа, к которому надо прикрепить новый экземпляр. Если опустить *theClip*, то *attachMovie( )* прикрепит новый экземпляр к текущему клипу (тому, в котором находится вызов *attachMovie( )*). Параметр *symbolIdentifier* – это строка, содержащая имя символа, используемого для генерирования экземпляра, как оно указано в поле Identifier параметров Linkage библиотеки (на шаге 4); *symbolIdentifier* не обязательно должен совпадать с именем самого фильма. Параметр *newName* является строкой, определяющей идентификатор вновь создаваемого экземпляра. Если строка *newName* не может быть преобразована в допустимый идентификатор, это может привести к достаточно удивительным результатам, как показывают примеры, приведенные далее в этом разделе. Наконец, *depth* является целым числом, обозначающим место в стеке содержимого родительского клипа, куда должен быть помещен новый экземпляр. Flash MX вводит поддержку необязательного четвертого параметра *initObject*, который позволяет скопировать свойства существующего объекта в новый клип. Детали см. в описании *MovieClip.attachMovie( )* в «Справочнике по языку».

Например, чтобы прикрепить новый клип, основанный на символе «square», к текущей временной диаграмме, выполните следующий код:

```
this.attachMovie("square", "square1_mc", 1);
square1_mc._x = 50;   // Поместить новый клип в точку с горизонтальной координатой 50
square1_mc._y = 200;  // ...и вертикальной координатой 200
```

Метод *attachMovie( )* прикрепляет новый клип в качестве дочернего для *theClip* (или текущего клипа, если не задан никакой клип). Не так ведет себя функция *duplicateMovieClip( )*, прикрепляющая клип на одном уровне с дублируемым.

Экземпляр, прикрепляемый к другому клипу, располагается в центре клипа среди содержимого стека контента клипа. Экземпляр, прикрепляемый к главному фильму документа, располагается в левом верхнем углу сцены в точке с координатами (0, 0).

Во Flash Player 6 функция *attachMovie( )* возвращает ссылку на вновь созданный клип, что удобно при отладке. Если функция возвращает `undefined`, это означает, что создание клипа было неудачным (в отличие от Flash 5, где *attachMovie( )* всегда возвращает `undefined`). В таком случае, скорее всего, было неверно задано имя *symbolIdentifier* (помните о необходимости заключить строку в кавычки, правильно ввести имя и указать для экспорта имя, заданное в параметрах Linkage, а не имя символа в библиотеке). Строка *symbolIdentifier* не чувствительна к регистру, поэтому если символ экспортирован как «Symbol 1», вы можете задать «symbol 1» в качестве значения *symbolIdentifier* (хотя, согласно правилам хорошего стиля, делать этого не следует).

Значение, возвращаемое функцией *attachMovie( )*, особенно полезно, когда операция успешна, но не удается обратиться к новому клипу через его пред-

полагаемый идентификатор. Формально не обязательно, чтобы *newName* было допустимым идентификатором, но в таком случае при обращении к новому клипу возникнут проблемы, если не записать возвращенное значение в переменную. Это означает, что к уже прикрепленному клипу можно обращаться с помощью возвращенного значения, если параметр *newName* был задан неправильно и не может быть превращен в допустимый идентификатор. Фактически параметр *newName* сначала преобразуется в строку, а затем снова преобразуется в идентификатор. Рассмотрим этот механизм подробнее.

В нашем первом примере имя *newName* задано идентификатором «newClip 1» и содержит пробел, поэтому преобразование в допустимый идентификатор невозможно. К счастью, обратиться к вновь созданному клипу можно посредством возвращенного значения, записанного в newClip_mc:

```
newClip_mc = _root.attachMovie("Symbol 1", "newClip 1", 0);
trace(newClip_mc);  // Выводит: _level0.newClip 1
                    // (не является допустимым идентификатором)
```

В следующем примере *newName* задано уже идентификатором newClip1, по виду вполне допустимым, но кавычек, необходимых для задания строки, в нем нет. Команда *attachMovie( )* сначала пытается преобразовать его в строку, но поскольку интерпретатор рассматривает newClip1 как переменную со значением undefined, преобразование возвращает пустую строку. К счастью, можно вновь обратиться к созданному клипу с помощью значения, сохраненного в newClip_mc:

```
newClip_mc = _root.attachMovie("Symbol 1", newClip1, 0);
trace(newClip_mc);  // Выводит _level0.
                    // (имя клипа - пустая строка!)
```

А если задать *newName* как допустимый идентификатор, например ball_mc, который ссылается на уже существующий клип? Тогда команда *attachMovie( )* сначала снова пытается преобразовать его в строку, которая (согласно табл. 3.2 для преобразования строк клипов) дает для *newName* промежуточное значение «_level0.ball_mc». Эта серия преобразований приводит к тому, что у вновь созданного клипа будет неверное имя «_level0._level0.ball_mc». К счастью, еще раз можно воспользоваться возвращенным значением, хранящимся в newClip_mc для правильной ссылки на вновь созданный клип:

```
//Предполагаем, что ball_mc - это  идентификатор существующего клипа
newClip_mc = _root.attachMovie("Symbol 1", ball_mc, 0);
trace(newClip_mc);        // Выводит "_level0._level0.ball_mc"
```

Как видите, жизнь значительно облегчается, если задать строку, которая преобразуется в допустимый идентификатор для *newName*.

Создание экземпляров с помощью createEmptyMovieClip()

Во Flash Player 6 можно создавать в существующем клипе совершенно пустые новые клипы, если обратиться к методу *createEmptyMovieClip( )*, имеющему такой синтаксис:

```
theClip.createEmptyMovieClip(newName, depth);
```

Здесь *theClip* представляет собой имя существующего клипа, к которому следует прикрепить новый пустой клип. Новый экземпляр клипа получает имя *newName* и помещается в стек содержимого *theClip* на указанную глубину *depth*.

Клипы, создаваемые с помощью *createEmptyMovieClip( )*, не являются производными от символов библиотеки; это просто пустые экземпляры клипов с одним кадром. Пустые клипы можно использовать в качестве холста для рисования, клипов сценариев или контейнеров для вложенных клипов (например, в качестве клипа формы, содержащего текстовые поля и кнопку Submit).

Имена экземпляров

Создаваемым экземплярам присваиваются идентификаторы, или *имена экземпляров*, что позволяет позже ссылаться на них. Идентификаторы присваиваются клипам не так, как обычным объектам. При создании обычного объекта данных (не клипа) надо присвоить ссылку на этот объект переменной или другому контейнеру данных, чтобы объект сохранился и в дальнейшем можно было обращаться к нему по имени. Например:

```
new Object();              // Объект умирает сразу после создания, и ссылки
                           // на него невозможны, поскольку мы его не сохранили
var thing = new Object();  // Ссылка на объект сохраняется в thing, и
                           // потом к нему можно обращаться как к thing
```

Ссылки на экземпляры клипов не обязательно сохранять в переменных, чтобы иметь возможность к ним обращаться. В отличие от обычных объектов экземпляры клипов доступны в ActionScript через имена экземпляров сразу, как только они созданы – программно или в среде разработки. Способ получения экземпляром своего первоначальноого имени зависит от способа его создания. Программно генерируемые экземпляры получают свои имена на этапе исполнения от функции, которая их создает. Экземплярам, создаваемым вручную, обычно явно присваиваются имена с помощью инспектора свойств в редакторе, для чего необходимо:

1. Выделить экземпляр на сцене.
2. В инспекторе свойств ввести в поле <Instance Name> имя экземпляра.

(Во Flash 5 имя экземпляра устанавливается через панель Instance.) Присвоив экземпляру имя в среде разработки, к нему можно обращаться в ActionScript по этому имени. (Полезно добавлять к именам идентификаторов клипов, создаваемых во время разработки или на этапе исполнения, суффикс «_mc».) Например, если на сцене существует экземпляр с именем `ball_mc`, то к его свойствам можно обращаться так:

```
ball_mc._y = 200;
```

Если созданному вручную клипу не присвоено имя экземпляра, его автоматически назначает Flash Player на этапе исполнения. Автоматически назначаемые имена выбираются из последовательности `instance1`, `instance2`, `instance3`, ... `instance`*n*, но они не описывают содержимое клипа осмысленно

(и необходимо гадать, какое автоматическое имя было сгенерировано). Например, к первому неименованному экземпляру клипа можно обратиться так:

```
instance1._y = 200;
```

Поскольку имена экземпляров, присваиваемые на этапе разработки или выполнения, используются в ActionScript как идентификаторы, всегда следует образовывать их согласно правилам создания допустимых идентификаторов, описанным в главе 15. Прежде всего, имена экземпляров не должны начинаться с цифр или содержать дефисы и пробелы. Принято задавать в именах экземпляров клипов суффикс «_mc», особенно если предполагается обращаться к ним через ActionScript на этапе выполнения.

Имя каждого экземпляра хранится в его встроенном свойстве `_name`, которое доступно как для чтения, так и для записи. Для клипов, определенных вручную на этапе разработки, значением `_name` по умолчанию служит строка с идентификатором исходного клипа:

```
trace(ball_mc._name);       // Выводит "ball_mc"
```

Для клипов, определенных программно, начальное значение `_name` задается параметром *newName*, который передается *duplicateMovieClip( )* или *attachMovie( )*.

Свойство `_name` полезно при отладке или выводе имени клипа (без полного пути). Обратите внимание, что значение, возвращаемое `_name`, отличается от других представлений идентификатора клипа (см. также старое свойство `_target`, обсуждаемое далее в этой главе):

```
trace(instance1._name);         // Выводит instance1
trace(instance1);               // Выводит _level0.instance1
trace(String(instance1));       // Выводит _level0.instance1
trace(targetPath(instance1));   // Выводит _level0.instance1
trace(instance1.toString());    // Выводит [object Object]
```

В примере 13.1 свойство `_name` применяется для поиска конкретного клипа. Пример предотвращения бесконечного цикла при помощи свойства `_name` можно найти в разделе «Свойство _name» этой главы.

Пример 13.1. Поиск клипов на временной диаграмме

```
// Находит в gameboard_mc все клипы, в имени которых есть слово "enemy".
for (var prop in gameboard_mc) {
  if (typeof gameboard_mc[prop] == "movieclip") {
    if (gameboard_mc[prop]._name.indexOf("enemy") != -1) {
      // Найдя "вражеский" клип, заставить его атаковать игрока.
      gameboard_mc[prop].attackPlayer();
    }
  }
}
```

Если изменить свойство _name экземпляра, то все последующие ссылки на этот экземпляр должны использовать новое имя. Так, если изменить значение ball_mc._name, то ссылка ball_mc перестанет существовать, и ссылаться на этот экземпляр надо по новому имени:

```
ball_mc._name = "circle_mc";  // Изменить имя ball_mc на circle_mc
trace(typeof ball_mc);        // Выводит "undefined", поскольку ball_mc
                              // больше не существует.
circle_mc._x = 59;            // После изменения имени надо пользоваться
                              // новым именем клипа.
```

Поэтому не следует изменять свойство _name на этапе выполнения, так как это может привести к сбою в работе кода.

Импорт внешних фильмов и графики

Мы рассказали о создании экземпляров клипов внутри одного документа, но Flash Player может одновременно отображать несколько документов *.swf*. С помощью *loadMovie( )* (в качестве глобальной функции или метода клипа) можно импортировать в проигрыватель Flash внешний файл *.swf* и поместить его в экземпляр клипа или на уровень с заданным номером над основным фильмом (то есть на передний план относительно основного фильма).

Во Flash Player 6 *loadMovie( )* может также загружать графические файлы JPEG в клип или на нужный уровень документа. Подробности см. в описании *loadMovie( )* и *MovieClip.loadMovie( )* в «Справочнике по языку».

Благодаря размещению содержимого в отдельных файлах достигается точный контроль над процессом загрузки и облегчается частичная модификация приложения. Допустим, например, что есть фильм, содержащий главное меню навигации и пять подразделов. Прежде чем пользователь сможет переместиться в пятый раздел, должна завершиться загрузка разделов с первого по четвертый. Но если поместить каждый раздел в отдельный файл *.swf*, то разделы можно загружать в произвольном порядке, и пользователь получает прямой доступ к каждому разделу. Чтобы модифицировать раздел, мы заменяем соответствующий файл *.swf* новым.

Когда внешний файл *.swf* загружается на указанный уровень, его главная временная диаграмма становится корневой временной диаграммой этого уровня и происходит замена фильма, который ранее был загружен на этот уровень. Аналогично, когда внешний фильм загружается в клип, его главная временная диаграмма заменяет временную диаграмму этого клипа, выгружая графические элементы, звуки и сценарии, находящиеся в этом клипе.

Как и *duplicateMovieClip( )*, *loadMovie( )* можно вызывать как самостоятельную функцию или как метод экземпляра. При самостоятельном использовании синтаксис *loadMovie( )* таков:

```
loadMovie(url, location)
```

Здесь *url* задает адрес внешнего файла *.swf*, который надо загрузить. Параметр *location* – это строка, указывающая путь к существующему клипу или уровню документа, куда должен быть помещен новый файл *.swf*, то есть загружаемый фильм. Например:

```
loadMovie("circle.swf", "_level1");
loadMovie("photos.swf", "viewClip");
```

Так как в строковом контексте ссылка на клип преобразуется в путь, можно задавать *location* в виде ссылки на клип, то есть _level1 , а не "_level1". Однако, работая со ссылками, будьте осторожны. Если передаваемая ссылка не указывает на реальный клип, функция *loadMovie( )* ведет себя неожиданно: она загружает файл *.swf* в *текущую* временную диаграмму. Дополнительную информацию по данной теме можно найти в разделе «Проблемы, связанные с перекрытием методов и глобальных функций» далее в этой главе.

Синтаксис *loadMovie( )* как метода клипа таков:

```
theClip.loadMovie(url);
```

Если *loadMovie( )* выступает в качестве метода клипа, то предполагается, что внешний *.swf* загружается в *theClip*, поэтому параметр *location*, требуемый для самостоятельной функции *loadMovie( )*, не нужен. Таким образом, задается только путь к загружаемому файлу *.swf* через параметр *url*. Разумеется, *url* может указывать как абсолютное, так и относительное имя файла, например:

```
viewClip.loadMovie("photos.swf");
```

Если загруженный фильм помещен в экземпляр клипа, он принимает свойства этого клипа (например, масштаб клипа, угол поворота, преобразование цвета и т. д.).

Обратите внимание, что для использования *loadMovie( )* в виде метода *theClip* должен существовать. Например, следующая попытка загрузки *circle.swf* окажется неудачной, если _level1 пуст:

```
_level1.loadMovie("circle.swf");
```

Очередность выполнения загрузки фильма

Функция *loadMovie( )* выполняется не сразу после появления в блоке инструкций. На самом деле она выполняется лишь после того, как будут выполнены *все* остальные инструкции в блоке.

Нельзя обращаться к свойствам или методам загружаемого извне фильма в том же самом блоке кода, где находится вызов *loadMovie( )*, загружающий этот фильм в проигрыватель.

Поскольку функция *loadMovie( )* загружает внешний файл (обычно из сети), она выполняется *асинхронно*. Это означает, что *loadMovie( )* может завершиться в любой момент в зависимости от скорости передачи файла. Поэтому перед обращением к загружаемому фильму необходимо проверить, закончена ли передача фильма в проигрыватель. Это делается посредством кода,

Использование loadMovie() с attachMovie()

Загрузка внешнего файла *.swf* в экземпляр клипа с помощью *loadMovie( )* приводит к неожиданному результату: становится невозможным прикреплять к этому клипу экземпляры с помощью *attachMovie( )*. Если в клип загружен внешний файл *.swf*, он больше не принимает прикрепляемых клипов из библиотеки, от которой он произошел. Например, если *movie1.swf* содержит экземпляр с именем `clipA`, и в `clipA` загрузить *movie2.swf*, то становится невозможным прикрепление к `clipA` экземпляров из библиотеки *movie1.swf*.

В чем тут дело? Метод *attachMovie( )* действует только внутри *одного* документа. Это означает, что нельзя прикреплять экземпляры из библиотеки одного документа к другому документу. При загрузке в клип файла *.swf* этот клип становится носителем нового документа и, следовательно, новой (другой) библиотеки. Последующие попытки прикрепления к клипу экземпляров из исходного документа оказываются безуспешными, поскольку библиотека клипа больше не соответствует библиотеке первоначального документа. Однако если выгрузить из клипа документ с помощью *unloadMovie( )*, то возможность прикрепления к клипу символов из библиотеки его документа восстанавливается.

Аналогично загрузка в клип файла *.swf* с помощью *loadMovie( )* лишает нас возможности копирования этого клипа с помощью *duplicateMovieClip( )*.

который часто называют *предзагрузчиком* (*preloader*). Он проверяет, какая часть файла загружена, прежде чем разрешить выполнение каких-либо действий. Предзагрузчики можно создавать с помощью свойств клипов `_totalframes` и `_framesloaded` и методов клипов *getBytesLoaded( )* и *getBytesTotal( )*. Примеры кода можно найти в соответствующих статьях для класса *MovieClip* в «Справочнике по языку».

Порядок расположения фильмов и экземпляров в стеке

Все экземпляры клипов и загружаемые извне фильмы, воспроизводимые в проигрывателе Flash, зрительно располагаются друг над другом подобно колоде карт. Если экземпляры или загруженные извне файлы *.swf* перекрывают друг друга в проигрывателе, один клип («верхний») закрывает другой клип («нижний»). В принципе, это выглядит достаточно просто, но при этом главный *стек содержимого*, в котором находятся *все* экземпляры и файлы *.swf*, фактически дробится на множество меньших вложенных стеков. Сначала мы рассмотрим эти вложенные стеки отдельно, а потом разберемся, как они объединяются и образуют главный стек. (В рамках данного обсуждения стек содержимого не имеет прямого отношения к стекам LIFO и FIFO, обсуждавшимся в главе 11.)

Глубина клипа

Каждый экземпляр клипа, включая главную временную диаграмму фильма, помещает различные элементы своего содержимого (клипы, текстовые поля и кнопки) в один из стеков: внутренний стек уровней (для элементов, создаваемых в среде разработки) или программно генерируемый стек содержимого (для элементов, создаваемых на этапе выполнения). Для элементов, находящихся в этих стеках (в совокупности называемых *стеком содержимого клипа*), устанавливается целочисленное значение *глубины* (*depth*), которое управляет их перекрытием на экране. Это значение находится в диапазоне от (16 384 до 1 048 575. Значения глубины от 0 до 1 048 575 отведены для динамически генерируемого содержимого, а значения от –16 383 до –1 выделены для содержимого, создаваемого в среде разработки; глубина –16 384 отведена для динамического содержимого, которое оказывается ниже любого содержимого среды разработки в каждом клипе. Значение глубины некоторого элемента выясняется при помощи метода *getDepth( )*, а для того чтобы поменять значения глубины двух клипов, следует вызвать метод *swapDepths( )*.

Внутренний стек слоев

Экземпляры, созданные вручную в среде разработки Flash, располагаются во *внутреннем стеке слоев* (*internal layer stack*). Порядок слоев в этом стеке управляется фактическими слоями временной диаграммы фильма: если перекрываются два созданных вручную экземпляра с различных слоев временной диаграммы, то экземпляр, находящийся в более высоком слое, закрывает экземпляр, находящийся в менее высоком слое. («Высокий» означает, что этот слой находится вверху окна временной диаграммы в среде разработки Flash).

Кроме того, поскольку на одном уровне временной диаграммы может находиться несколько клипов, каждый слой внутреннего стека слоев фактически поддерживает свой собственный министек. Перекрывающиеся клипы, расположенные на одном и том же слое временной диаграммы, упорядочиваются в стеке с помощью команд среды разработки Modify → Arrange.

Положения двух экземпляров во внутреннем стеке слоев можно поменять местами с помощью метода *swapDepths( )* при условии, что они находятся на одной и той же временной диаграмме (то есть значения свойства `_parent` должны быть одинаковы для обоих клипов). В версиях, предшествовавших Flash 5, нельзя было изменять внутренний стек слоев посредством ActionScript.

Значение глубины для элементов, создаваемых в среде разработки, непредсказуемо и предполагается зарезервированным для использования этой средой. Поэтому при обмене уровней глубины клипов, один из которых создан в среде разработки, а другой – программным образом, всегда задавайте в *swapDepths( )* параметр *target*, а не целочисленное значение уровня, как это описано в «Справочнике по языку» для *MovieClip.swapDepths( )*.

Стек содержимого, создаваемого программно

Программно создаваемые экземпляры обычно хранятся в стеке отдельно от экземпляров, создаваемых *вручную* и содержащихся во внутреннем стеке слоев. Каждый экземпляр обладает собственным *стеком программно генерируемого содержимого*, в котором находятся:

- Клипы, созданные с помощью *duplicateMovieClip( )*, *attachMovie( )* и *createEmptyMovieClip( )*
- Экземпляры текстовых полей, созданных с помощью *createTextField( )*

Порядок клипов в стеке программно генерируемого содержимого зависит от способа их создания.

Добавление в стек клипов, созданных с помощью attachMovie() и createEmptyMovieClip()

Новый экземпляр, созданный с помощью *attachMovie( )* или *createEmptyMovieClip( )*, всегда кладется сверху (то есть на передний план) по отношению к клипу, к которому он был прикреплен. Допустим, к примеру, что в стеке внутренних слоев фильма есть два клипа – `X` и `Y`, и что `X` располагается в слое, лежащем выше `Y`. Предположим также, что новый клип `A` прикрепляется к `X` и новый клип `B` прикрепляется к `Y`:

```
x.attachMovie("A", "A", 0);
y.attachMovie("B", "B", 0);
```

В этом сценарии клипы будут располагаться сверху вниз в порядке `A`, `X`, `B`, `Y`, как показано на рис. 13.1.

Рис. 13.1. *Пример стека экземпляров*

После того как клип сгенерирован, он также предоставляет отдельное пространство над своим внутренним стеком слоев для новых программно генерируемых клипов. Это значит, что можно прикреплять клипы к прикрепленным клипам.

Клипы, прикрепленные к фильму `_root` документа Flash, помещаются в программно генерируемый стек содержимого, который оказывается впереди *всех* экземпляров клипов `_root`, даже тех, которые имеют программно генерируемое содержание.

Разовьем предшествующий пример. Если прикрепить клип C к _root фильма, содержащего клипы X, Y, A и B, то клип C появится впереди всех остальных клипов. Расширенная структура показана на рис. 13.2.

Рис. 13.2. *Стек экземпляров, показывающий клип, прикрепленный к _root*

Добавление в стек клипов, созданных с помощью duplicateMovieClip()

Каждый экземпляр, копируемый с помощью *duplicateMovieClip( )*, попадает в программный стек родительского объекта породившего его клипа (клипа, на временной диаграмме которого находится порождающий клип). Рассмотрим это на нашем примере.

Если создать клип D путем дублирования клипа X (созданного вручную), то клип D помещается в стек над _root, вместе с клипом C. У порождающего клипа и его дубликата всегда один и тот же родительский объект (в данном случае _root). Аналогично, если создать клип E путем дублирования клипа D, то E также помещается в стек над _root, с клипами C и D. Но если создать клип F путем дублирования клипа A, созданного с помощью *attachMovie( )*, то F помещается в программный стек над X, с клипом A. И снова у F и породившего его клипа A оказывается один и тот же родительский объект: X. Лучше всяких слов это иллюстрирует рис. 13.3.

Назначение глубины для экземпляров в стеке программно генерируемого содержимого

Читатель может заинтересоваться, каким образом определяется порядок в стеке клипов C, D и E или клипов A и F, показанных на рис. 13.3. Порядок в стеке программно генерируемого клипа определяется аргументом *depth*, который передается методам *attachMovie( )*, *createEmptyMovieClip( )* или *duplicateMovieClip( )* и может в любой момент быть изменен с помощью функции *swapDepths( )*. Для каждого программно генерируемого клипа его *глубина* (*depth*), иногда называемая также его *z-индексом*, определяет положение клипа в конкретном стеке программно генерируемых клипов.

Глубина клипа может быть любым целым числом и измеряется снизу вверх, так что глубина 5 ниже, чем глубина 6; глубина 7 выше, чем глубина 6

Рис. 13.3. *Стек экземпляров, показывающий различные дублированные клипы*

(то есть находится впереди); глубина 8 еще выше и т. д. Если два программно сгенерированных клипа занимают на экране одну и ту же область, тот из них, который имеет большее значение глубины, выводится поверх другого.

Хотя на этапе разработки один слой временной диаграммы может быть занят несколькими клипами, на каждой позиции глубины обитает только один жилец. Только один клип может занимать слой в некоторый момент времени: если поместить клип в позицию, которая уже занята, то предыдущий клип, занимавший ее, удаляется.

Вполне допускаются разрывы в глубине клипов: один клип может находиться на глубине 0, другой – на глубине 500, а третий – на глубине 1000. Наличие разрывов не снижает производительность и не увеличивает расход памяти.

Стек «_level» документа .swf

Помимо внутреннего стека слоев и стека программно генерируемого содержимого, есть еще третий (и последний) вид стека, *стек документов* (*document stack*), или *стек уровней* (*level stack*), который управляет перекрытием не экземпляров, а целых файлов *.swf*, загружаемых в проигрыватель посредством *loadMovie( )*.

Файл *.swf*, загружаемый в проигрыватель Flash первым, помещается на самый нижний уровень стека документов (соответствующий глобальному свойству `_level0`). При загрузке в проигрыватель вслед за этим первым документом каких-либо дополнительных файлов *.swf* можно при желании поместить их перед первоначальным документом, назначив им в стеке документов уровень выше `_level0`. Все содержимое документов, имеющих более высокий уровень в стеке документов, появляется впереди документов с более низким уровнем независимо от порядка в стеках клипов внутри каждого документа.

Так же как стек программно генерируемого содержимого допускает размещение только одного клипа в каждом слое, стек документов разрешает иметь на каждом уровне только один документ. Если загрузить файл *.swf* на занятый уровень, ранее располагавшийся на нем документ будет заменен новым. Например, можно заменить исходный документ, загрузив новый файл *.swf* на `_level0`. Загрузка нового файла *.swf* на `_level1` приведет к закрытию изображения фильма на `_level0`, но не удалит его из проигрывателя.

Отношения между различными стеками, поддерживаемыми Flash-проигрывателем, представлены на рис. 13.4.

Рис. 13.4. *Полный стек содержимого Flash Player*

Стеки и порядок выполнения кода

Расположение слоев временных диаграмм и распределение клипов по слоям оказывают влияние на порядок выполнения кода. Действуют следующие правила:

- Код в кадрах различных слоев временной диаграммы всегда выполняется сверху вниз (относительно панели временной диаграммы).
- При начальной загрузке экземпляров, созданных вручную, код их временных диаграмм и обработчиков событий *onLoad( )* выполняется согласно порядку загрузки (Load Order), установленному в Publish Settings документа Flash: либо снизу вверх (Bottom Up), что является порядком по умолчанию, либо сверху вниз (Top Down).

  Допустим, что есть диаграмма с двумя слоями, *top* и *bottom*, причем *top* находится над *bottom* в стеке слоев. Клип `X` находится в слое *top*, а клип `Y` в слое *bottom*. Если порядок загрузки для документа определен как Bottom Up, то код в клипе `Y` выполняется раньше кода в клипе `X`. Если напротив, порядок загрузки установлен как Top Down, то код в клипе `X` будет выполнен раньше кода в клипе `Y`. Этот порядок выполнения применяется *только* к кадру, в котором `X` и `Y` появляются впервые.
- После загрузки для всех экземпляров фильма устанавливается порядок выполнения, противоположный порядку загрузки: код последнего добавленного в фильм экземпляра всегда будет выполняться первым.

Полагаясь на эти правила, следует проявлять осторожность. Слои могут изменяться, поэтому надо избегать кода, зависящего от их относительного положения. Следует стремиться к созданию такого кода, который можно безбоязненно выполнять независимо от порядка клипов в стеке. Уберечься от некоторых проблем, обусловленных стеком выполнения, можно, размещая весь код в слое *scripts*, лежащем наверху каждой содержащей код временной диаграммы.

Ссылки на экземпляры и главные фильмы

В предыдущих разделах этой главы мы рассказали, как создавать и распределять по слоям экземпляры клипов и внешние файлы *.swf* для Flash Player. Чтобы успешно управлять содержимым с помощью ActionScript, надо иметь возможность ссылаться на это содержимое.

Ссылки на экземпляры и фильмы нужны в четырех общих случаях, когда требуется:

- Получить или установить свойство клипа или фильма.
- Создать или вызвать метод клипа или фильма.
- Применить к клипу или фильму некоторую функцию.
- Обработать клип или фильм как данные, например при сохранении их в переменной или передаче в качестве аргумента функции.

Хотя ситуации, в которых требуется сослаться на клип или фильм, достаточно просты, способы создания ссылок многочисленны и разнообразны. В оставшейся части этого раздела будут рассмотрены средства ActionScript для создания ссылок на экземпляры и фильмы.

Использование имен экземпляров

Ранее мы видели, что ссылаться на клипы можно по *именам* их *экземпляров*, например:

```
trace(someVariable);     // Ссылка на переменную
trace(someClip_mc);      // Ссылка на клип
```

Для непосредственной ссылки на экземпляр (как в *trace( )* приведенного примера) экземпляр должен располагаться на временной диаграмме, к которой прикреплен код. Например, если экземпляр с именем `clouds_mc` помещен на главную временную диаграмму документа, можно обращаться к `clouds_mc` из кода, прикрепленного к главной временной диаграмме:

```
// Установить свойство экземпляра
clouds_mc._alpha = 60;
// Вызвать метод экземпляра
clouds_mc.play();
// Поместить экземпляр в массив с другими родственными экземплярами
var background = [clouds_mc, sky_mc, mountains_mc];
```

Если экземпляр, на который надо сослаться, находится на временной диаграмме, отличной от той, где расположен код, необходимо использовать более развитый синтаксис, описываемый ниже в разделе «Ссылки на вложенные экземпляры».

Ссылка на текущий экземпляр фильма

При ссылке на клип не всегда обязательно указывать имя экземпляра. Код, прикрепленный к кадру на временной диаграмме экземпляра, может ссылаться на свойства и методы этого экземпляра непосредственно, без имени экземпляра.

Например, чтобы установить свойство `_alpha` клипа `cloud_mc`, можно поместить в кадр на временной диаграмме `cloud_mc` такой код:

```
_alpha = 60;
```

Аналогично для вызова метода *play( )* клипа `cloud_mc` из кадра на временной диаграмме `cloud_mc` достаточно выполнить:

```
play();
```

Этот прием можно применять на любых временных диаграммах, в том числе временных диаграммах главных фильмов. Например, следующие две инструкции являются синонимами, если они прикреплены к кадрам на главной временной диаграмме документа Flash. Первая ссылается на главный фильм неявным образом, тогда как вторая делает это явно с помощью глобального свойства `_root`:

```
gotoAndStop(20);
_root.gotoAndStop(20);
```

Однако не все методы могут вызываться с неявной ссылкой на клип. Всякий метод клипа, имя которого совпадает с именем соответствующей глобальной функции (например, *duplicateMovieClip( )* или *unloadMovie( )*), должен вызываться с явной ссылкой на экземпляр. Поэтому при возникновении сомнений ссылка должна быть явной. Нам еще придется подробнее поговорить о конфликтах методов и глобальных функций далее в этой главе в разделе «Проблемы, связанные с перекрытием методов и глобальных функций».

Учтите, что явные ссылки на переменные или клипы всегда надежнее, чем неявные. Последние неоднозначны, что часто приводит к неожиданным результатам и вводит в заблуждение других разработчиков, которые читают ваш код.

Возвратные ссылки, использующие ключевое слово this

Для того чтобы явно сослаться на текущий экземпляр из кадра на его временной диаграмме или в одном из его обработчиков событий, можно воспользоваться ключевым словом `this`. Например, следующие инструкции будут равносильны, если они прикреплены к кадру на временной диаграмме экземпляра `cloud_mc`:

```
_alpha = 60;        // Неявная ссылка на текущую временную диаграмму
this._alpha = 60;   // Явная ссылка на текущую временную диаграмму
```

Есть три причины использовать `this` при ссылке на клип, даже когда допустимо ссылаться на свойства и методы клипа непосредственно.

- Во-первых, другим разработчикам легче читать явные ссылки, потому что они устраняют неоднозначность смысла предложения.
- Во-вторых, при отсутствии явной ссылки на экземпляр некоторые методы клипов могут быть ошибочно приняты интерпретатором за глобальные функции. Если опустить ссылку `this`, интерпретатор решает, что делается попытка вызвать глобальную функцию с таким же именем и сообщает об отсутствии параметра *target*. Для решения этой проблемы используется `this`, например:

```
this.duplicateMovieClip("newClouds_mc", 0);  // Вызвать метод экземпляра

// Если опустить ссылку this, возникнет ошибка
duplicateMovieClip("newClouds_mc", 0);  // Ой!
```

- В-третьих, с помощью `this` удобно передавать ссылки на текущую временную диаграмму функциям, действующим над клипами:

```
// Вот функция, обрабатывающая клипы
function moveClipTo (theClip, x, y) {
  theClip._x = x;
  theClip._y = y;
}
```

```
// Теперь вызовем ее с текущей временной диаграммой
moveTo(this, 150, 125);
```

Замечание как для новичков, так и для опытных объектно-ориентированных программистов: внутри метода `this` является ссылкой не на текущую временную диаграмму, а на объект, через который вызван метод. Если метод объекта должен обратиться к конкретному клипу, ссылку на этот клип надо передать методу в виде параметра. Подробности использования ключевого слова `this` внутри методов см. в главе 12.

Ссылки на вложенные экземпляры

Как отмечалось во введении в данную главу, экземпляры клипов часто оказываются вложенными один в другой. Это означает, что холст клипа может содержать экземпляр другого клипа, который, в свою очередь, может содержать экземпляры других клипов. Например, в некоторой игре клип `spaceship` (космический корабль) может содержать экземпляр клипа `blinkingLights` (мерцающие огни) или клипа `burningFuel` (горящее топливо). Либо клип персонажа `face` (лицо) может содержать отдельные клипы `eyes`, `nose` и `mouth` (глаза, нос и рот).

Ранее было кратко показано, что можно осуществлять перемещение вверх или вниз из любой точки иерархии экземпляров клипов, подобно тому как можно перемещаться вверх и вниз в последовательности каталогов на жестком диске. Изучим это подробнее и рассмотрим еще несколько примеров.

Сначала посмотрим, как сослаться на экземпляр клипа, расположенный *внутри* текущего экземпляра.

Когда клип помещается на временную диаграмму другого клипа, он становится свойством этого клипа, и обращаться к нему можно точно так же, как к любому другому свойству объекта (с помощью оператора «точка»).

Допустим, например, что `clipB` помещен на холст `clipA`. Для доступа к `clipB` из кадра на временной диаграмме `clipA` применяется непосредственная ссылка на `clipB`:

```
clipB._x = 30;
```

Можно воспользоваться и явной ссылкой:

```
this.clipB._x = 30;
```

Допустим теперь, что в `clipB` содержится еще один экземпляр, `clipC`. Для ссылки на `clipC` из кадра на временной диаграмме `clipA` обращение к `clipC` осуществляется как к свойству `clipB`:

```
clipB.clipC.play();
clipB.clipC._x = 20;
```

Красиво, не правда ли? И эту систему можно расширять до бесконечности. Так как каждый экземпляр одного клипа, помещенный на временную диаграмму другого, становится свойством приютившего его клипа, можно обходить иерархию, разделяя экземпляры операторами точки, например:

```
clipA.clipB.clipC.clipD.gotoAndStop(5);
```

Теперь, когда мы знаем, как перемещаться вниз по иерархии экземпляров, посмотрим, как перемещаться по ней вверх, чтобы ссылаться на экземпляр или фильм, которые содержат текущий экземпляр. Как отмечалось выше, у каждого экземпляра есть встроенное свойство _parent, ссылающееся на содержащие его клип или главный фильм. Свойством _parent можно пользоваться так:

```
theClip._parent
```

где *theClip* является ссылкой на экземпляр клипа. Вспоминая наш недавний пример с клипом clipA на главной временной диаграмме, клипом clipB внутри clipA и клипом clipC внутри clipB, посмотрим, как с помощью _parent и точки ссылаться на различные клипы в иерархии. Допустим, что в кадр временной диаграммы clipB помещен такой код:

```
_parent          // Явная относительная ссылка на clipA
this             // Явная относительная ссылка на clipB (текущий clip)
this._parent     // Еще одна ссылка на clipA
// Милая Шейла, я в восторге! Давай еще попробуем...
_parent._parent  // Ссылка на родителя clipA (дедушку клипа clipB), которым
                 // в данном случае является главная временная диаграмма
```

Заметьте, что нет необходимости в окольном пути для перемещения *вниз* по иерархии с помощью ссылки на свойство clipC клипа clipB только для того, чтобы перейти назад *вверх* по иерархии с помощью _parent, хотя это допускается. Необходимости в таких окружных ссылках нет, но они демонстрируют гибкость точечной нотации:

```
clipC._parent                  // Окружная ссылка на clipB
                               // (текущая временная диаграмма)
clipC._parent._parent._parent  // Окружная ссылка на главную временную диаграмму
```

Обратите внимание на применение оператора «точка» для спуска по иерархии клипов и свойства _parent для восхождения по ней.

Если для вас это в новинку, попробуйте построить иерархию clipA, clipB, clipC во Flash и проверить код нашего примера. Правильные ссылки на экземпляры представляют одну из основ мастерства программирования на ActionScript.

Обратите внимание на сходство иерархии клипов с родословным деревом. В отличие от обычного родословного дерева видов, размножающихся половым путем, в котором у каждого потомка есть два родителя, наше родословное дерево развивается неполовым образом. Это значит, что каждая семья возглавляется единственным родителем, который может усыновить любое

количество детей. У каждого клипа (то есть каждого *узла* дерева) может быть один и только один родитель (содержащий его клип), но несколько *дочерних элементов* (содержащихся в нем клипов). Конечно, родитель каждого клипа может, в свою очередь, иметь единственного родителя, и это значит, что у каждого клипа может быть только один дедушка (а не два дедушки и две бабушки, как у людей). На рис. 13.5 приведен пример иерархии клипов.

Рис. 13.5. *Пример иерархии клипов*

Как низко ни спускайся по родословному дереву, вернувшись назад на то же количество шагов, обязательно попадешь в исходную точку. Следовательно, бессмысленно опускаться вниз по иерархии, чтобы потом возвращаться назад. Однако *не* бессмысленно подняться по иерархии вверх, а затем спуститься по *другому* пути. Допустим, что главная временная диаграмма нашего примера содержит также `clipD`, который станет, таким образом, «братом» `clipA`, поскольку для обоих клипов родителем является главная временная диаграмма `_parent`. В этом случае ссылаться на `clipD` из сценария, прикрепленного к `clipB`, можно так:

```
_parent._parent.clipD   // Это ссылка на clipD, дочерний для главной временной
                        // диаграммы (родителя clipA) и потому одноуровневый с clipA
```

Обратите внимание, что у главной временной диаграммы нет свойства `_parent` (главный фильм находится в вершине любой иерархии клипов и не может содержаться в другой временной диаграмме): ссылки на `_root. _parent` возвращают `undefined`.

Ссылка на главный фильм с помощью _root и _leveln

Рассмотрев способы перемещения вверх и вниз по иерархии клипов *относительно* текущего клипа, займемся другими способами перемещения по *абсолютным путям* и даже между документами, хранящимися на других уровнях стека документов проигрывателя Flash. В предыдущих главах такие

технологии рассматривались применительно к переменным и функциям, теперь же посмотрим, как они используются для управления клипами.

Ссылка на главный фильм текущего уровня с помощью _root

Если некоторый экземпляр находится в глубине иерархии клипов, можно добраться до главной временной диаграммы фильма, последовательно применяя свойство _parent. Однако для облегчения ссылок на главную временную диаграмму из глубоко вложенных клипов можно воспользоваться встроенным глобальным свойством _root, дающим короткую ссылку на главную временную диаграмму. Вот, например, как воспроизвести главный фильм:

```
_root.play();
```

О свойстве _root говорят, что это *абсолютная ссылка* на некоторую точку в иерархии клипов, поскольку, в отличие от свойств _parent и this, являющихся относительными по отношению к текущему клипу, свойство _root указывает на главную временную диаграмму текущего уровня, независимо от того, из какого клипа к нему обращаются (об исключении из этого правила сказано ниже в предупреждении). Все следующие ссылки эквивалентны одна другой (исключение составляют сценарии, прикрепленные к главной временной диаграмме, где свойство _parent не действует):

```
_parent._root
this._root
_root
```

Поэтому можно использовать _root, когда неизвестно положение данного клипа в иерархии. Рассмотрим, к примеру, такую иерархию, в которой circle является дочерним для главной временной диаграммы фильма, а square — дочерним для circle:

```
main timeline
  circle
    square
```

Теперь допустим, что следующий сценарий прикреплен к кадру circle и кадру square:

```
_parent._x += 10  // Переместить родительский клип данного клипа
                  // на 10 пикселов вправо
```

В результате выполнения этого кода в circle главный фильм будет перемещен на 10 пикселов вправо. В результате выполнения этого кода в square клип circle (а не главный фильм) будет перемещен на 10 пикселов вправо. Для того чтобы сценарий перемещал главный фильм на 10 пикселов вне зависимости от места его выполнения, надо переписать его так:

```
_root._x += 10   // Переместить главный фильм на 10 пикселов вправо
```

Кроме того, свойство _parent не действует внутри главной временной диаграммы, а версия сценария с использованием _root будет действовать и в кадре главной временной диаграммы.

Свойство `_root` можно сочетать с обычными ссылками на экземпляры для спуска по иерархии вложенных клипов:

```
_root.clipA.clipB.play();
```

Ссылки, начинающиеся с `_root`, указывают на одну и ту же известную начальную точку из любого места документа. Несложно догадаться, на что они ссылаются.

После загрузки файла *.swf* в экземпляр клипа `_root` становится ссылкой не на главную временную диаграмму этого файла *.swf*, а на главную временную диаграмму фильма, в который был загружен файл *.swf*!

Если известно, что фильм будет загружен в клип, нельзя пользоваться `_root` для ссылки на главную временную диаграмму. Следует определить глобальную ссылку на главную временную диаграмму, поместив на главную временную диаграмму фильма такой код:

```
_global.myAppMain = this;
```

Здесь *myApp* — имя вашего приложения, а словом `Main` принято обозначать главную временную диаграмму. Затем используйте *myApp*`Main` вместо `_root`.

Ссылки на другие документы в проигрывателе Flash с помощью _leveln

Если в стек документов проигрывателя Flash с помощью *loadMovie( )* загружено несколько файлов *.swf*, то на главные временные диаграммы различных документов можно ссылаться с помощью последовательности встроенных глобальных свойств от `_level0` до `_level`*n*, где *n* задает слой документа, на который надо сослаться.

Поэтому `_level0` представляет документ на самом нижнем уровне стека документов (документы на более высоких уровнях будут выводиться на переднем плане). Если только какой-либо фильм не был загружен на `_level0` с помощью *loadMovie( )*, `_level0` занят фильмом, изначально загруженным при запуске проигрывателя.

В следующем примере воспроизводится главная временная диаграмма документа, находящегося на уровне 3 стека документов проигрывателя:

```
_level3.play();
```

Так же как и свойство `_root`, свойство `_level`*n* можно сочетать с обычными ссылками с помощью оператора «точка»:

```
_level1.clipA.stop();
```

Как и ссылки `_root`, ссылки на свойства `_level`*n* называются *абсолютными*, так как они приводят в одно и то же место из любой точки документа.

Обратите внимание, что `_level`*n* и `_root` не являются синонимами. Свойство `_root` всегда представляет главную временную диаграмму *текущего* документа независимо от уровня, на котором тот находится, в то время как свой-

ство `_level`*n* является ссылкой на главную временную диаграмму фильма на определенном уровне документов. Допустим, к примеру, что в *myMovie.swf* есть код `_root.play()`. Если загрузить *myMovie.swf* на уровень 5, то этот код будет воспроизводить главную временную диаграмму `_level5`. С другой стороны, если поместить в *myMovie.swf* код `_level2.play()` и загрузить *myMovie.swf* на уровень 5, то этот код будет воспроизводить главную временную диаграмму `_level2`, а не `_level5`. Разумеется, для уровня 2 `_root` и `_level2` будут эквивалентны.

Создание ссылок на экземпляры с помощью Insert Target Path

Если структура экземпляров в фильме становится очень сложной, составление ссылок на клипы и главные временные диаграммы может стать утомительным. Не всегда удается вспомнить точную иерархию последовательности клипов, в результате чего приходится часто выделять клипы и обращаться к среде разработки, чтобы определить структуру их вложенности. Инструмент Insert Target Path окна Actions создает ссылки на клип визуально, освобождая от ручного труда. Кнопка Insert Target Path показана на рис. 13.6.

Рис. 13.6. *Кнопка Insert Target Path*

Для того чтобы воспользоваться средством Insert Target Path, надо выполнить следующее:

1. Поместить курсор в то место кода, куда требуется вставить ссылку на клип.
2. Щелкнуть по кнопке Insert Target Path, показанной на рис. 13.6.
3. В диалоговом окне Insert Target Path выбрать клип, на который надо сослаться.
4. Выбрать тип ссылки: *absolute reference*, чтобы начать ссылку с `_root`, или *relative reference*, что обозначает ссылку на целевой клип относительно клипа, в котором содержится ваш код (`this`).

5. Если осуществляется экспорт в формат Flash 4, выбрать кнопку Slashes Notation для совместимости с Flash 4. (Выбираемая по умолчанию кнопка Dot Notation создает ссылки, не работающие во Flash 4.) См. табл. 2.1.

Инструмент Insert Target Path не позволяет генерировать относительные ссылки, содержащие восхождение по иерархии клипов. То есть его нельзя использовать для ссылки на клип, содержащий текущий (если только не начать путь с `_root` и не вести его вниз). Для создания ссылок, восходящих по иерархии клипов, необходимо использовать относительные ссылки, начинающиеся с `_root` (которые становятся поэтому нисходящими ссылками), либо вручную ввести относительные ссылки в код с использованием свойства `_parent`.

Динамические ссылки на объекты клипов

Обычно имя конкретного экземпляра или фильма, с которым производятся действия, известно, но иногда желательно управлять клипом, имя которого мы не знаем. Например, может потребоваться уменьшить размер целой группы клипов, применив цикл, или создать кнопку, которая при щелчке по ней ссылается на различные клипы. Для работы в таких ситуациях надо динамически создавать ссылки на клипы на этапе выполнения.

Использование оператора доступа к элементам массива

Как было показано в главах 5 и 12, свойства объектов можно извлекать с помощью оператора «точка» или оператора доступа к элементам массива `[]`. Например, следующие два утверждения эквивалентны:

```
someObject.myProperty = 10;
someObject["myProperty"] = 10;
```

У оператора доступа к элементам массива есть одно важное свойство, отсутствующее у оператора «точка»: он позволяет (и даже требует) ссылаться на свойство посредством *строкового выражения*, а не *идентификатора*. Вот, например, выражение конкатенации строк, которое дает действительную ссылку на свойство `propertyX`:

```
someObject["prop" + "ertyX"];
```

Этот же прием годится для динамического создания ссылок на экземпляры и фильмы. Выше отмечалось, что экземпляры клипов хранятся в виде свойств своих родительских клипов. Для ссылки на такие свойства экземпляров ранее применялся оператор «точка». Например, из кода на главной временной диаграмме можно сослаться на клип `clipB`, вложенный внутрь другого экземпляра, `clipA`, следующим образом:

```
clipA.clipB;          // Ссылка на clipB, находящийся внутри clipA
clipA.clipB.stop();   // Вызвать метод клипа clipB
```

Поскольку экземпляры являются свойствами, допустимо ссылаться на них с помощью оператора `[]`, например:

```
clipA["clipB"];                    // Ссылка на clipB, находящийся внутри clipA
clipA["clipB"].stop();             // Вызвать метод клипа clipB
```

Обратите внимание, что в случае ссылки на clipB с помощью оператора [] имя clipB передается как строка, а не как идентификатор. Ссылка в виде строки может быть любым выражением, возвращающим строку. Вот, например, ссылка на clipB с применением конкатенации строк:

```
var clipCount = "B";
clipA["clip" + clipCount];          // Ссылка на clipB, находящийся в clipA
clipA["clip" + clipCount].stop();   // Вызвать метод клипа clipB
```

Можно динамически создавать ссылки на клипы для доступа к ряду последовательно именованных клипов:

```
// Цикл для остановки clip1, clip2, clip3 и clip4
for (var i = 1; i <= 4; i++) {
  _root["clip" + i].stop();
}
```

Сила!

Сохранение ссылок на клипы в контейнерах данных

В начале главы было сказано, что хотя в ActionScript клипы формально являются особым типом данных, ActionScript работает с ними, как с объектами. Поэтому можно записать ссылку на экземпляр клипа в переменную, элемент массива или свойство объекта.

Вспомните пример иерархии вложенных клипов (clipC, вложенный в clipB, вложенный в clipA), помещенной на главную временную диаграмму документа. Если сохранить эти разные клипы в контейнерах данных, то можно динамически управлять ими с помощью контейнеров, а не явных ссылок на клипы. В примере 13.2, содержащем код, который помещен в кадр главной временной диаграммы, хранение экземпляров и управление ими организовано при помощи контейнеров данных.

Пример 13.2. Хранение ссылок на клипы в переменных и массивах

```
var x = clipA.clipB;     // Записать ссылку на clipB в переменную x
x.play();                // Воспроизвести clipB
// Теперь запишем клипы в элементы массива
var theClips = [clipA, clipA.clipB, clipA.clipB.clipC];
theClips[0].play();      // Воспроизвести clipA
theClips[1]._x = 200;    // Поместить clipB в 200 пикселах от точки регистрации clipA
// Остановить все клипы в массиве, используя цикл
for (var i = 0; i < theClips.length; i++) {
  theClips[i].stop();
}
```

Сохраняя ссылки на клипы в контейнерах данных, можно обрабатывать их (например, воспроизводить, поворачивать или останавливать), не зная иерархии клипов в документе и не касаясь ее. Хранение ссылок на клипы в переменных также облегчает чтение кода. Длинный абсолютный или относительный путь в иерархии клипов можно заменить более коротким и простым именем переменной.

Доступ к клипам с помощью цикла for–in

В главе 8 было показано, как перечислить свойства объекта с помощью цикла *for-in*. Напомним, что переменная итератора цикла *for-in* автоматически пробегает по всем свойствам объекта, и тело цикла выполняется по одному разу для каждого свойства:

```
for (var prop in someObject) {
  trace("the value of someObject." + prop + " is " + someObject[prop]);
}
```

В примере 13.3 показано, как с помощью цикла *for-in* перечислить все клипы, находящиеся на данной временной диаграмме.

Пример 13.3. Перечисление клипов, находящихся на временной диаграмме

```
for (var property in myClip) {
  // Проверить, является ли текущее свойство myClip клипом
  if (typeof myClip[property] == "movieclip") {
    trace("Found instance: " + myClip[property]._name);
    // Здесь с клипом выполняются какие-то действия
    myClip[property]._x = 300;
    myClip[property].play();
  }
}
```

С помощью цикла *for-in* удобно обращаться к клипам, содержащимся в конкретном экземпляре клипа или на главной временной диаграмме. Цикл *for-in* позволяет контролировать любые клипы на любых временных диаграммах независимо от того, известно ли имя клипа и был ли он создан вручную или программным образом.

Пример 13.4 представляет собой рекурсивную версию примера 13.3. Здесь отыскиваются все экземпляры клипов на временной диаграмме, а также экземпляры клипов на всех вложенных временных диаграммах.

Пример 13.4. Рекурсивный поиск всех клипов на временной диаграмме

```
function findClips (theClip, indentSpaces) {
  // С помощью пробелов выделить отступом дочерние клипы для каждого последующего уровня
  var indent = " ";
  for (var i = 0; i < indentSpaces; i++) {
    indent += " ";
  }
  for (var property in theClip) {
    // Проверить, является ли текущее свойство theClip клипом
    if (typeof theClip[property] == "movieclip") {
      trace(indent + theClip[property]._name);
      // Проверить, является ли данный клип родительским для каких-либо других клипов
      findClips(theClip[property], indentSpaces + 4);
    }
  }
}
findClips (_root, 0); // Найти все экземпляры клипов, являющиеся потомками
                      // главной временной диаграммы
```

Дополнительные сведения о рекурсивных функциях можно найти в разделе «Рекурсивные функции» главы 9.

Свойство _name

Как указывалось выше в разделе «Имена экземпляров клипов», имя каждого экземпляра хранится в виде строки во встроенном свойстве _name. Как было показано в примере 13.1, это свойство позволяет определить имя текущего клипа или какого-либо другого клипа в иерархии экземпляров:

```
this._name;          // Имя текущего экземпляра
_parent._name        // Имя клипа, содержащего текущий клип
```

Свойство _name удобно для выполнения условных операций, определенных для конкретных клипов. Например, следующий код создает копию клипа seedClip при его загрузке:

```
onClipEvent (load) {
  if (this._name == "seedClip") {
    this.duplicateMovieClip("clipCopy", 0);
  }
}
```

Непосредственно проверяя, не является ли seedClip именем клипа, удается избежать бесконечной рекурсии: без условного предложения обработчик события *load* каждого дублированного цикла заставил бы клип еще раз дублировать себя.

Свойство _target

У каждого экземпляра клипа есть встроенное свойство _target – строка, задающая абсолютный путь к клипу в устаревших обозначениях Flash 4 с косой чертой. Например, если clipB помещен внутрь clipA, а clipA помещен на главную временную диаграмму, свойства _target этих клипов имеют следующие значения:

```
_root._target                    // Содержит: "/"
_root.clipA._target              // Содержит: "/clipA"
_root.clipA.clipB._target        // Содержит: "/clipA/clipB"
```

Функция targetPath()

Функция *targetPath( )* возвращает строку, содержащую путь абсолютной ссылки на клип в точечной нотации. Функция *targetPath( )* служит современным объектно-ориентированным эквивалентом _target. Она имеет следующий формат:

```
targetPath(theClip)
```

где *theClip* – это идентификатор клипа, абсолютную ссылку на который требуется получить. Вот несколько примеров для той же иерархии клипов:

```
targetPath(_root);              // Содержит: "_level0"
targetPath(_root.clipA);        // Содержит: "_level0.clipA"
targetPath(_root.clipA.clipB);  // Содержит: "_level0.clipA.clipB"
```

Функция *targetPath( )* возвращает полный путь к клипу, тогда как свойство `_name` дает только имя этого клипа. (Это аналогично полному пути к файлу и просто имени файла.) Поэтому *targetPath( )* можно использовать в коде, управляющем клипами, исходя не только из их имен, но и их местонахождения. Например, можно создать общую кнопку для навигации, устанавливающую свой цвет, соответствующий тому разделу содержимого, в котором она находится, на основании значения, возвращаемого *targetPath( )*. Демонстрация практического применения *targetPath( )* есть в «Справочнике по языку» в примере для объекта *Selection*.

Удаление экземпляров клипов и главных фильмов

Мы научились создавать клипы и пользоваться ссылками на них; теперь посмотрим, как превращать их в изначальный хаос (иными словами, как от них избавляться).

Способ создания экземпляра или фильма определяет технологию последующего удаления этого экземпляра или фильма. Можно явным образом удалять фильмы и экземпляры с помощью *unloadMovie( )* и *removeMovieClip( )*. Кроме того, можно изгнать клип неявно, поместив на его место новый клип с помощью *loadMovie( )*, *attachMovie( )* или *duplicateMovieClip( )*. Рассмотрим эти приемы по отдельности.

Использование unloadMovie() с экземплярами и уровнями

Встроенная функция *unloadMovie( )* может удалить любой экземпляр клипа или главный фильм – как созданный вручную, так и созданный функциями *loadMovie( )*, *duplicateMovieClip( )* и *attachMovie( )*. Ее можно вызывать как глобальную функцию или как метод уровня экземпляра:

```
unloadMovie(clipOrLevel);   // Глобальная функция
clipOrLevel.unloadMovie();  // Метод
```

В формате *unloadMovie( )* как глобальной функции `clipOrLevel` представляет собой строку, указывающую путь к клипу или уровню, которые должны быть выгружены. Поскольку в строковом контексте клипы преобразуются в пути, `clipOrLevel` может также быть ссылкой на клип. В формате *unloadMovie( )* как метода `clipOrLevel` должен быть ссылкой на объект клипа. Конкретные действия, выполняемые *unloadMovie( )*, различаются в зависимости от того, применяется она к уровню или экземпляру.

Использование unloadMovie() с уровнями

Будучи применена к уровню в стеке документов (то есть `_level0`, `_level1` или `_level2`), *unloadMovie( )* полностью удаляет указанный уровень и содержащийся на нем фильм. Последующие ссылки на удаленный уровень возвращают значение `undefined`. Удаление уровней документов является самым распространенным применением функции *unloadMovie( )*:

```
unloadMovie("_level1");
_level1.unloadMovie();
```

Использование unloadMovie() с экземплярами

Будучи применена к экземпляру (созданному вручную или программно), *unloadMovie( )* удаляет *содержимое* клипа, *но не сам клип*! Временная диаграмма и холст клипа удаляются, но пустая оболочка остается на сцене. На эту оболочку можно ссылаться, пока экземпляр не будет удален окончательно посредством *removeMovieClip( )* (или пока не закончится диапазон кадров, на которых располагается экземпляр). Кроме того, любые обработчики событий клипа в этой оболочке продолжают действовать.

Такое частичное удаление экземпляров предоставляет интересные возможности, позволяя иметь общий клип-контейнер, содержимое которого можно многократно изменять с помощью *loadMovie( )* и *unloadMovie( )*. Например, единственный клип позволяет загружать разделы веб-сайта или графику для фотоальбома. Следующий отрывок кода демонстрирует такой прием для экземпляра по имени `clipA` (в реальном приложении к ним надо добавить соответствующий код предзагрузчика):

```
clipA.loadMovie("section1.swf");  // Загрузить документ в clipA
clipA.unloadMovie();              // Выгрузить документ, сохранив clipA
clipA.loadMovie("section2.swf");  // Загрузить в clipA другой документ
```

Одно предупреждение, связанное с такими действиями. В случае применения с экземпляром *unloadMovie( )* удаляет все пользовательские свойства клипа, содержащегося в экземпляре. Физические свойства, например, `_x` и `_alpha`, сохраняются, но пользовательские переменные и функции утрачиваются.

Если при использовании *unloadMovie( )* в формате глобальной функции передать ей в качестве аргумента несуществующий клип или уровень, будет выгружен тот клип, из которого вызвана функция *unloadMovie( )*.

Например, если `_level1` не определен, то при выполнении из главной временной диаграммы `_level0` представленного ниже кода будет выгружен `_level0`:

```
unloadMovie(_level1);
```

Такой образ действий по-своему логичен, но об этом будет сказано ниже в разделе «Проблемы, связанные с перекрытием методов и глобальных функций». Это препятствие можно обойти путем использования строки при задании аргумента *clipOrLevel* функции *unloadMovie( )* либо путем явной проверки существования *clipOrLevel* перед его выгрузкой. Вот пример того и другого способа:

```
unloadMovie("_level1");    // clipOrLevel задан в виде строки
if (_level1) {             // Явная проверка существования уровня
  unloadMovie(_level1);
}
```

Использование removeMovieClip() для удаления экземпляров

Для удаления из проигрывателя экземпляров, полученных с помощью *duplicateMovieClip( )*, *attachMovie( )* или *createEmptyMovieClip( )*, можно использовать функцию *removeMovieClip( )*. Экземпляр удаляется, когда он больше не нужен приложению, например, если вражеский космический корабль уничтожен или окно сообщения закрыто. Обратите внимание, что *removeMovieClip( )* действует только для прикрепленных или дублированных экземпляров. Удалить экземпляр, созданный вручную, или главный фильм она не может. Как и *unloadMovie( )*, *removeMovieClip( )* можно вызвать в формате метода или глобальной функции (несмотря на различие синтаксиса результат одинаков):

```
removeMovieClip(theClip)    // Глобальная функция
theClip.removeMovieClip()   // Метод
```

Когда *removeMovieClip( )* имеет формат глобальной функции, *theClip* представляет собой строку, указывающую путь к клипу, который надо удалить. Поскольку в строковом контексте клипы преобразуются в строки, *theClip* может также быть ссылкой на клип. Когда *removeMovieClip( )* имеет формат метода, *theClip* должен быть ссылкой на объект клипа.

В отличие от *unloadMovie( )*, удаление экземпляра с помощью *removeMovieClip( )* целиком уничтожает объект клипа, не оставляя оболочки и каких-либо следов клипа или его свойств. После выполнения *theClip*.removeMovieClip() все ссылки на *theClip* возвращают значение undefined.

Удаление экземпляров, созданных вручную

Экземпляры клипов, созданные вручную в редакторе Flash, также имеют ограниченную продолжительность жизни: они удаляются, когда головка воспроизведения входит в ключевой кадр, в котором их нет. Поэтому созданные вручную клипы живут в страхе перед всемогущим пустым ключевым кадром.

Запомните, что когда клип исчезает с временной диаграммы, он перестает существовать как объект данных. Все переменные, функции, методы и свойства, определенные внутри него, утрачиваются. Поэтому, если желательно сохранить информацию или функции клипа, необходимо следить, чтобы клип не был удален вручную, и диапазон кадров, в которых находится клип, охватывал те точки, где нужна информация клипа. (На практике, чтобы совершенно об этом не заботиться, надо прикрепить код, который требуется постоянно, к кадру на главной временной диаграмме фильма или объекту _global.) Чтобы спрятать клип, находящийся на временной диаграмме, достаточно установить его свойство _visible равным false. Установка свойства _x равным очень большому положительному числу или очень маленькому отрицательному числу также обеспечивает невидимость клипа для пользователя, но этот способ рекомендовать нельзя, поскольку вывод клипа продолжается и за пределами экрана, что расходует ресурсы.

Проблемы, связанные с перекрытием методов и глобальных функций

Как уже неоднократно говорилось в этой главе, имена некоторых методов клипов совпадают с именами глобальных функций. Вы сами можете убедиться в этом в среде разработки Flash. Откройте панель Actions, перейдите в режим Expert Mode и загляните в Movie Control и Movie Clip Control в папке Actions. Вы увидите список действий, в том числе *gotoAndPlay( )*, *gotoAndStop( )*, *nextFrame( )* и *unloadMovie( )*. Эти действия могут выполняться и как методы клипов. Такое дублирование означает не просто разделение на категории: действия являются глобальными функциями, совершенно отличными от методов клипов с теми же именами.

Поэтому при выполнении

```
theClip.gotoAndPlay(5);
```

происходит вызов *метода клипа* с именем *gotoAndPlay( )*. Но при выполнении

```
gotoAndPlay(5);
```

происходит обращение к *глобальной функции* с именем *gotoAndPlay( )*. У этих двух команд одинаковые имена, но сами они разные. Глобальная функция *gotoAndPlay( )* работает с текущим экземпляром или фильмом, а метод *gotoAndPlay( )* – с объектом клипа, через который он вызван. Чаще всего это тонкое различие не играет роли. Но для некоторых перекрывающихся пар метод/функция разница может быть весьма существенна.

Некоторые глобальные функции требуют параметра `target`, указывающего клип, над которым должна действовать функция. Этот параметр не требуется соответствующим методам клипа, потому что методы автоматически действуют над клипами, через которые вызываются. Например, *unloadMovie( )* в форме метода работает так:

```
theClip.unloadMovie();
```

В качестве метода *unloadMovie( )* вызывается без параметров и автоматически выполняет действия с `theClip`. Но в качестве глобальной функции *unloadMovie( )* действует так:

```
unloadMovie(target);
```

В формате глобальной функции *unloadMovie( )* требует параметр `target`, который задает фильм, подлежащий выгрузке. Почему при этом могут возникнуть проблемы? Прежде всего потому, что можно ошибочно рассчитывать, будто текущий документ будет выгружен при вызове глобальной версии *unloadMovie( )* без параметров, подобно тому как это происходит при вызове *gotoAndPlay( )* без параметров:

```
unloadMovie();
```

Такой формат *не* выгружает текущий клип. Он вызывает сообщение об ошибке «Wrong number of parameters» (неверное количество параметров). Вторая

причина возможных проблем с параметрами *target* в глобальных функциях несколько сложнее, и ее будет достаточно трудно проследить, если вы к этому не готовы. При передаче клипа *target* в глобальную функцию, которой необходим параметр *target*, можно использовать строку, описывающую путь к клипу, которым надо управлять, или ссылку на клип. Например:

```
unloadMovie(_level1);     // Целевой клип задан как ссылка
unloadMovie("_level1");   // Целевой клип задан как строка
```

Можно использовать ссылку просто потому, что ссылки на объекты клипов в строковом контексте преобразуются в пути к клипам. Достаточно просто, но если параметр *target* разрешается в пустую строку или значение undefined, то ***функция работает с объектами текущей временной диаграммы***!

Следующие примеры демонстрируют, как некорректная ссылка на клип может непреднамеренно выгрузить текущую временную диаграмму:

```
unloadMovie(x);    // Если x не существует, то x разрешается в undefined,
                   // и функция действует над текущей временной диаграммой
unloadMovie("");   // target является пустой строкой, поэтому функция
                   // действует над текущей временной диаграммой
```

В результате можно получить совершенно неожиданные результаты. Посмотрим, что получится, если сослаться на несуществующий уровень:

```
unloadMovie(_level1);
```

Если _level1 пуст, то интерпретатор разрешает ссылку, как если бы она была необъявленной переменной. В результате получаем значение undefined, поэтому функция действует над текущей временной диаграммой, а не над _level1! Как же справиться с таким поведением? Есть несколько вариантов. Можно проверить, существует ли цель, прежде чем выполнять над ней функцию:

```
if (_level1) {
  unloadMovie(_level1);
}
```

Можно взять за правило всегда указывать цель при помощи строки. Если путь, указанный в строке, не разрешается в реальный клип, вызов функции оказывается неудачным:

```
unloadMovie("_level1");
```

В некоторых случаях можно использовать для той же операции эквивалентную функцию с числовым аргументом:

```
unloadMovieNum(1);
```

Наконец, можно вообще избавиться от проблем, если всегда применять методы клипа:

```
_level1.unloadMovie();
```

Вот, для справки, перечень функций, с которыми могут быть проблемы (глобальных функций ActionScript для Flash 5, принимающих параметр *target*):

duplicateMovieClip( )
loadMovie( )
loadVariables( )
print( )
printAsBitmap( )
removeMovieClip( )
startDrag( )
unloadMovie( )

Если в фильме возникают необъяснимые ошибки, то, возможно, стоит взять этот список и проверить, не вызваны ли они неправильным применением глобальной функции. При передаче ссылки на клип в качестве параметра `target` не поленитесь лишний раз проверить правильность синтаксиса.

Рисование в клипе на этапе выполнения

Во Flash MX появилась возможность рисовать отрезки, кривые, фигуры и делать заливку в клипах с помощью *Drawing API*, набора методов для рисования на этапе выполнения. Методы Drawing API описаны в «Справочнике по языку» в следующих статьях:

MovieClip.beginFill( )
MovieClip.beginGradientFill( )
MovieClip.clear( )
MovieClip.curveTo( )
MovieClip.endFill( )
MovieClip.lineStyle( )
MovieClip.lineTo( )
MovieClip.moveTo( )

Чтобы указать текущую позицию для рисования в *Drawing API* используется понятие *пера* (*drawing pen*), подобного чертежному перу в старых линейных графопостроителях. Первоначально перо располагается в точке регистрации клипа (с внутренними координатами (0, 0)). Методы черчения позволяют:

- Перемещать перо без вычерчивания линий и заливки с помощью *MovieClip.moveTo( )*
- Начертить отрезок прямой от текущего положения пера до заданной точки с помощью *MovieClip.lineTo( )*
- Начертить кривую линию от текущего положения пера до заданной точки с помощью *MovieClip.curveTo( )*
- Начертить фигуру с помощью *MovieClip.beginFill( )* и *MovieClip.endFill( )* или *MovieClip.beginGradientFill( )* и *MovieClip.endFill( )*

До, во время и после рисования можно задать характеристики линии, воспользовавшись *MovieClip.lineStyle( )*. Удалить начерченное можно с помощью *MovieClip.clear( )*.

Обратите внимание, что в Drawing API нет методов для вычерчивания таких фигур, как треугольник, прямоугольник, окружность. Они создаются с помощью элементарных методов черчения, как показано в «Справочнике по языку» для *MovieClip.beginFill( )*.

Подробное описание методов черчения оставлено для «Справочника по языку». Здесь мы рассмотрим только пару примеров, демонстрирующих общее применение Drawing API.

В примере 13.5 вычерчивается прямая линия длиной 100 пикселов и толщиной 2 пиксела вправо от точки регистрации текущего клипа.

Пример 13.5. Вычерчивание прямой линии

```
// Установить толщину линии в 2 пиксела
this.lineStyle(2);
// Начертить линию
this.lineTo(100,0);
```

В примере 13.6 создается красный квадрат со стороной 200 пикселов с центром в точке регистрации текущего клипа.

Пример 13.6. Вычерчивание квадрата

```
// Установить толщину линии в 3 пиксела
this.lineStyle(3);
// Переместить перо на 100 пикселов влево вверх от точки регистрации
this.moveTo(-100,-100);
// Начать вычерчивание красного квадрата
this.beginFill(0xFF0000);
// Начертить стороны квадрата
this.lineTo(100, -100);
this.lineTo(100, 100);
this.lineTo(-100, 100);
this.lineTo(-100, -100);
// Закрыть нашу красную фигуру
this.endFill();
```

Ряд интересных применений Drawing API (включая черчение дуг, многоугольников, штриховых линий, звездочек и клиньев) можно найти по адресу:

http://www.formequalsfunction.com/downloads/drawmethods.html

Использование клипов в качестве кнопок

Во Flash Player 6 у клипов (но не у главных фильмов) есть все характеристики, которые раньше были зарезервированы за символами кнопок. Клип может динамически определять и переопределять обработчики событий кнопок, *активную область* (*hit region*), состояния *Up*, *Over* и *Down*. Кроме того, в отличие от кнопок, экземпляры клипов можно создавать динамически на этапе исполнения. Те, кому достаточно реализовать простое интерактивное поведение на этапе разработки, вполне могут обойтись символами кнопок. Но для создания сложного динамического поведения кнопок на этапе выполнения без клипов не обойтись.

Прежде всего при реализации клипа, ведущего себя аналогично кнопке, надо определить для клипа одно или несколько событий кнопки. Обычно для этого следует присвоить функцию обратного вызова готовому свойству события кнопки (показано ниже), но синтаксис кнопки *on(event)* можно применить к экземпляру клипа прямо в среде разработки. В следующем коде создается клип, в него помещается текстовое поле, а затем определяется обработчик события кнопки *onRelease( )*:

```
// Создать клип
this.createEmptyMovieClip("submit_mc", 1);

// Добавить текстовое поле
this.submit_mc.createTextField("submit_txt", 1, 0, 0, 50, 20);
this.submit_mc.submit_txt.text = "Submit";

// Определить обработчик кнопки
this.submit_mc.onRelease = function () {
  trace("You pressed the submit button.");
}
```

Такой клип отлично работает в качестве кнопки. Теперь попробуем сделать так, чтобы пользователю было легче щелкать по слову «Submit». Для увеличения размера активной (реагирующей на щелчок) области нашего клипа создадим отдельный *клип активной области* (*hit area movie clip*), выполнив следующие действия:

1. Создадим клип, размеры которого соответствуют желаемой области.
2. Скроем клип активной области, установив для его свойства `_visible` значение `false`.
3. Присвоим клип активной области свойству `hitArea` клипа с событиями кнопки.

Клип, выполняющий такую функцию активной области, может располагаться на любой временной диаграмме, но обычно его помещают внутри клипа с событиями кнопки, чтобы активная область перемещалась и меняла размеры вместе со своим родительским клипом. Например, в следующем коде с помощью Drawing API создается клип активной области с размерами, большими чем у слова «Submit»:

```
// Создать hit_mc внутри submit_mc
this.submit_mc.createEmptyMovieClip("hit_mc", 0);

// Нарисовать в hit_mc прямоугольник
this.submit_mc.hit_mc.moveTo(-30,-15);
this.submit_mc.hit_mc.beginFill(0xFF0000);
this.submit_mc.hit_mc.lineTo(80, -15);
this.submit_mc.hit_mc.lineTo(80, 35);
this.submit_mc.hit_mc.lineTo(-30, 35);
this.submit_mc.hit_mc.lineTo(-30, -15);
this.submit_mc.hit_mc.endFill();

// Скрыть клип активной области
this.submit_mc.hit_mc._visible = false;
```

```
// Задать hit_mc в качестве активной области submit_mc
this.submit_mc.hitArea = this.submit_mc.hit_mc;
```

Пока в нашем примере клип не меняет вида при нажатии кнопки мыши или перемещении указателя над кнопкой Submit. С помощью ActionScript изменение внешнего вида кнопки можно реализовать динамически внутри каждого обработчика событий кнопки. Например, можно выделить слово «Submit» жирным шрифтом в обработчике *onRollOver( )* клипа `submit_mc`:

```
// Жирный текст, если указатель мыши находится над кнопкой
this.submit_mc.onRollOver = function () {
  this.submit_txt.setTextFormat(new TextFormat(null, null, null, true));
}
// Обычный текст при выходе указателя за пределы кнопки
this.submit_mc.onRollOut = function () {
  this.submit_txt.setTextFormat(new TextFormat(null, null, null, false));
}
```

Другая возможность – создать клип в среде разработки и определить изменение видимого состояния кнопки с помощью ключевых кадров с особыми метками `_up`, `_over` и `_down`, соответствующих состояниям кнопки Up (мышь не над кнопкой), Over (мышь над кнопкой) и Down (кнопка нажата). (Чтобы присвоить метки ключевым кадрам, используйте инспектор свойств.) При взаимодействии мыши с клипом Flash автоматически перемещает головку воспроизведения фильма в кадр с соответствующей меткой. Например, при прохождении мыши над клипом Flash выполняет действие, эквивалентное следующему:

```
theClip.gotoAndStop("_over");
```

В каждом из помеченных кадров состояния кнопки можно выполнить команду *play( )* для создания эффекта анимированной кнопки. При этом, однако, надо также поместить команду *stop( )* в кадре 1 клипа, чтобы данный фильм не воспроизводился при всех состояниях кнопки. Обычно этот первый кадр имеет метку `_up` (неактивное состояние кнопки по умолчанию).

Клипы с событиями кнопок могут также использовать свойства кнопок: `enabled` (для перехода между состояниями включенной и выключенной кнопки), `useHandCursor` (для изменения или запрещения изменения курсора мыши при нахождении над кнопкой) и `trackAsMenu` (для модификации требований обработчика кнопки *onRelease( )*, включающей поведение в стиле меню). Например, для клипа `submit_mc` можно запретить отображение указателя мыши в виде руки:

```
submit_mc.useHandCursor = false;
```

Обычно веб-броузеры не показывают указатель ладони над кнопкой Submit. Более подробно свойства кнопок, доступных в клипах, рассмотрены в описании класса *MovieClip* в «Справочнике по языку».

Режим вложенных кнопок не поддерживается клипами. Это означает, что клипы, находящиеся внутри клипа с поведением кнопки, не могут вести се-

бя как кнопки. Кроме того, если обработчики кнопок определены для маски и маскированного клипа, действуют только обработчики кнопок маски.

У разных типов обработчиков событий для кнопок и клипов есть важные различия в областях видимости, которые обсуждались в главе 10 в разделе «Область видимости обработчиков событий» и были приведены в табл. 10.1.

Фокус ввода и клипы

Во Flash Player 6 экземпляры клипов (но не главные фильмы) могут получать фокус для ввода с клавиатуры, что позволяет управлять ими с помощью нажатий клавиш подобно объектам кнопок. Фокус ввода чаще всего используется компонентами, принимающими вводимые данные в сложных графических интерфейсах пользователя. Например, окно списка может прокручиваться с помощью клавиш со стрелками. Клавиша Enter активизирует выбранный элемент.

Вообще говоря, если действие, управляемое клавишами, не привязано непосредственно к конкретному клипу, для перехвата нажатий клавиш надо использовать события приемника объекта *Key*, а не клипы с фокусом ввода. См. описание *Key* в «Справочнике по языку».

Вопреки вашим возможным предположениям, клипы не получают фокуса ввода с клавиатуры, когда по ним выполняется щелчок. Они получают фокус при нажатии клавиши Tab или программно с помощью метода *setFocus( )*. Чтобы программно дать возможность клипу получить фокус ввода, надо явно установить для его свойства `focusEnabled` значение `true`. Напротив, клипы с обработчиками кнопок всегда могут получить фокус ввода, даже если `focusEnabled` имеет значение `false`. Если `focusEnabled` клипа равен `true`, то фокус можно поместить в клип программно с помощью *Selection.setFocus( )*. Пользователь может также поместить фокус в клип с помощью клавиши Tab, при условии, что свойство `tabEnabled` этого клипа равно `true`. Очередность получения клипом фокуса при обходе элементов нажатием клавиши Tab определяется значением его свойства `tabIndex`.

При получении клипом фокуса ввода обработчики событий *onKeyUp( )* и *onKeyDown( )* активизируются в виде функций обратного вызова. Из этих обработчиков можно реализовать связанное с клавиатурой поведение, такое как раскрытие иерархического меню при нажатии правой кнопки или переход к ближайшему элементу списка при нажатии клавиши с буквой.

Обнаружить получение или утрату фокуса ввода можно с помощью событий *onSetFocus( )* и *onKillFocus( )*, определенных в классе *MovieClip* и объекте *Selection*.

По умолчанию на фокус ввода указывает желтый прямоугольник. Такое выделение может быть удобным при отладке порядка табуляции, но может показаться зрительно непривлекательным. Чтобы убрать желтый прямоугольник, установите для свойства `MovieClip._focusrect` значение `false`. В этом случае надо дать конечному пользователю какую-то возможность

узнать, в каком элементе находится фокус ввода с клавиатуры. Используйте обработчики событий клипа *onSetFocus( )* и *onKillfocus( )*, чтобы переключать некоторый индикатор ввода, как показано ниже в примере.

Во Flash Player 6 перемещение мыши в то время, когда клип или кнопка находятся в фокусе, удаляет фокус с этого клипа или кнопки.

Следующий упрощенный код демонстрирует, как можно организовать реакцию элемента в корзине покупок на получение фокуса и удаление его при нажатии клавиши X. В реальном приложении такое поведение скорее всего будет реализовано на уровне класса, а не конкретного экземпляра клипа. Разработка методов *doHighlight( )* и *doRemoveHighlight( )* оставляется в качестве упражнения для читателя. Можно использовать эти обработчики для перемещения головки воспроизведения на метку, которая показывает или скрывает некий индикатор фокуса, такой как пунктирная линия или утолщенная рамка, которые можно видеть на кнопках с фокусом в диалоговых окнах Windows и Macintosh.

```
// Разрешить пользователю перемещать фокус на item_mc с помощью Tab
item_mc.tabEnabled = true;

// Разрешить Selection.setFocus() помещать фокус на item_mc
item_mc.focusEnabled = true;

// Выделить item_mc при получении фокуса
item_mc.onSetFocus = function () {
  this.doHighlight();
}

// Снять выделение с item_mc при потере фокуса
item_mc.onKillFocus = function () {
  this.doRemoveHighlight();
}

// Удалить item_mc при нажатии "x", когда item_mc в фокусе
item_mc.onKeyDown = function () {
  if (Key.getCode() == 88) {  // Клавиша "X" имеет код 88
    this.removeMovieClip();
  }
}
```

Значительно более подробную информацию о фокусе ввода клипов можно найти в «Справочнике по языку» в статьях, посвященных свойствам и методам, обсуждавшимся в этом разделе.

Создание часов с помощью клипов

Теперь, когда мы изучили основы программирования клипов, применим полученные знания для разработки простого приложения аналоговых часов, которое показывает пример типичного использования клипов как основных контейнеров содержимого. См. также разные версии вопросника с вариантами ответов, размещенные в сетевом хранилище кода.

В этой главе было показано, как создавать клипы с помощью *attachMovie( )* и как устанавливать свойства клипов посредством оператора «точка». Этих сравнительно простых средств и некоторой помощи со стороны классов *Date* и *Color* достаточно, чтобы изготовить часы с действующими часовой, минутной и секундной стрелками.

Прежде всего, изготовим циферблат и стрелки часов, для чего необходимо совершить следующие шаги (обратите внимание, что мы не станем размещать детали часов на главной сцене, а целиком создадим часы с помощью ActionScript):

1. Создать новый пустой фильм Flash.
2. Создать символ с именем *clockFace* (циферблат), содержащий черный круг шириной 100 пикселов, центр которого расположен в точке регистрации клипа.
3. Создать символ клипа с именем *hand* (стрелка), содержащий красную вертикальную черту длиной 50 пикселов.
4. Выделить линию в *hand*, затем выбрать пункт меню Window → Info (Flash MX) или Window → Panels → Info (Flash 5).
5. Мы хотим, чтобы стрелки вращались вокруг центра циферблата, поэтому расположим линию так, чтобы один ее конец находился в точке регистрации (центре) клипа стрелки, а нижний конец черты поместим в центр клипа, установив x-координату черты равной 0 и y-координату равной –50.

Теперь надо экспортировать символы `clockFace` и `hand`, чтобы их экземпляры можно было динамически прикреплять к фильму:

1. Выбрать клип *clockFace* в библиотеке, затем выбрать Linkage во всплывающем меню Options. Появляется окно диалога Linkage Properties.
2. Выбрать флажок Export For ActionScript.
3. В поле Identifier ввести `clockFace` и щелкнуть по кнопке OK.
4. Экспортировать клип `hand` (шаги с 1 по 3) и дать ему идентификатор `hand`.

Циферблат и стрелки готовы и могут присоединяться к фильму. Теперь напишем сценарий, который помещает детали часов на сцену и раз в секунду обновляет их:

1. Добавить сценарий, приведенный в примере 13.7, в кадр 1 *слоя 1* главной временной диаграммы.
2. Переименовать *слой 1* в *scripts*.

Сначала просмотрите пример 13.7 в целом, затем мы разберем его по частям.

Пример 13.7. Аналоговые часы

```
// Создать циферблат и стрелки часов
attachMovie("clockFace", "clockFace_mc", 0);
attachMovie("hand", "secondHand_mc", 3);
attachMovie("hand", "minuteHand_mc", 2);
attachMovie("hand", "hourHand_mc", 1);

// Установить положение и размеры циферблата
clockFace_mc._x = 275;
```

```
clockFace_mc._y = 200;
clockFace_mc._height = 150;
clockFace_mc._width = 150;

// Установить положение, размеры и цвет стрелок
secondHand_mc._x = clockFace_mc._x;
secondHand_mc._y = clockFace_mc._y;
secondHand_mc._height = clockFace_mc._height / 2.2;
secondHandColor = new Color(secondHand_mc);
secondHandColor.setRGB(0xFFFFFF);
minuteHand_mc._x = clockFace_mc._x;
minuteHand_mc._y = clockFace_mc._y;
minuteHand_mc._height = clockFace_mc._height / 2.5;
hourHand_mc._x = clockFace_mc._x;
hourHand_mc._y = clockFace_mc._y;
hourHand_mc._height = clockFace_mc._height / 3.5;

// Обновлять угол поворота стрелок в каждом проходимом кадре
function updateClock() {
  var now = new Date();
  var dayPercent = (now.getHours() > 12 ? now.getHours() - 12 : now.getHours()) / 12;
  var hourPercent = now.getMinutes()/60;
  var minutePercent = now.getSeconds()/60;
  hourHand_mc._rotation = 360 * dayPercent + hourPercent * (360 / 12);
  minuteHand_mc._rotation = 360 * hourPercent;
  secondHand_mc._rotation = 360 * minutePercent;
}
// Обновлять часы каждые 100 миллисекунд
setInterval(updateClock, 100);
```

Здесь много кода; рассмотрим его.

Сначала прикрепляется клип `clockFace`, которому присваивается глубина 0 (необходимо, чтобы он находился за стрелками часов):

```
attachMovie("clockFace", "clockFace", 0);
```

Затем прикрепляются три экземпляра символа *hand*, которым присваиваются имена `secondhand_mc`, `minuteHand_mc` и `hourHand_mc`. Каждая стрелка располагается в собственном слое в программно генерируемом стеке клипов над главной временной диаграммой. Клип `secondHand_mc` (глубина 3) расположен поверх клипа `minuteHand_mc` (глубина 2), находящегося поверх клипа `hourHand_mc` (глубина 1):

```
attachMovie("hand", "secondHand_mc", 3);
attachMovie("hand", "minuteHand_mc", 2);
attachMovie("hand", "hourHand_mc", 1);
```

Мы хотим поместить часы в центре, а не в левом верхнем углу сцены. Поэтому мы помещаем клип `clockFace_mc` в центр сцены и увеличиваем его с помощью свойств `_height` и `_width`. В данном примере предполагается, что фильм имеет размеры по умолчанию (550 × 400 пикселов), но мы могли бы с помощью `Stage.width` и `Stage.height` динамически получить размеры фильма во

время выполнения. В качестве упражнения используйте *Stage.onResize( )* для сохранения положения часов в центре при изменении размеров фильма.

```
clockFace_mc._x = 275;
clockFace_mc._y = 200;
clockFace_mc._height = 150;
clockFace_mc._width = 150;
```

После этого помещаем на часы клип secondHand_mc и делаем его немного меньше, чем радиус клипа clockFace_mc:

```
secondHand_mc._x = clockFace_mc._x;
secondHand_mc._y = clockFace_mc._y;
secondHand_mc._height = clockFace_mc._height / 2.2;
```

Вспомните, что линия в символе *hand* красная, поэтому пока все наши экземпляры *hand* тоже красные. Чтобы выделить клип secondHand, окрасим его в белый цвет, прибегнув к классу *Color*. Обратите внимание на использование шестнадцатеричного значения цвета 0xFFFFFF (дополнительные сведения о работе с цветом есть в статье, посвященной классу Color, в «Справочнике по языку»):

```
// Создать новый объект Color для управления secondhand_mc
secondHandColor = new Color(secondhand_mc);
// Назначить secondHand_mc белый цвет
secondHandColor.setRGB(0xFFFFFF);
```

Затем устанавливаются положение и размер клипов minuteHand_mc и hourHand_mc точно так же, как это делалось для клипа secondHand_mc:

```
// Поместить minuteHand_mc поверх clockFace_mc
minuteHand_mc._x = clockFace_mc._x;
minuteHand_mc._y = clockFace_mc._y;
// Сделать minuteHand_mc короче, чем secondHand_mc
minuteHand_mc._height = clockFace_mc._height / 2.5;
// Поместить hourHand_mc поверх clockFace_mc
hourHand_mc._x = clockFace_mc._x;
hourHand_mc._y = clockFace_mc._y;
// Сделать hourHand_mc самой короткой
hourHand_mc._height = clockFace_mc._height / 3.5;
```

Теперь надо установить угол поворота стрелок часов в соответствии с текущим временем. Однако угол поворота должен устанавливаться не просто один раз. Необходимо устанавливать его многократно, чтобы стрелки двигались с течением времени. Поэтому код для поворота помещен в функцию с именем *updateClock( )*, которая будет вызываться периодически:

```
function updateClock() {
  // Сохранить текущее время в now
  var now = new Date();

  // getHours() работает по 24-часовому циферблату. Если текущий час
  // больше 12, то вычтем 12 для преобразования в обычный 12-часовой.
  var dayPercent = (now.getHours() > 12 ? now.getHours() - 12 : now.getHours()) / 12;

  // Определим число прошедших минут часа как процент
```

```
    var hourPercent = now.getMinutes()/60;

    // Определим число прошедших секунд минуты как процент
    var minutePercent = now.getSeconds()/60;

    // Повернуть стрелки на циферблате на соответствующую величину
    hourHand_mc._rotation = 360 * dayPercent + hourPercent * (360 / 12);
    minuteHand_mc._rotation = 360 * hourPercent;
    secondHand_mc._rotation = 360 * minutePercent;
}
```

Первая задача *updateClock( )* состоит в том, чтобы получить и запомнить текущее время. Это осуществляется путем создания экземпляра класса *Date* и помещения его в локальную переменную `now`. Затем определим (в процентах) величину перемещения стрелки по циферблату – примеримся, где разрезать пирог. Текущий час всегда представляет некоторую долю 12, в то время как текущая минута и секунда всегда представляют некоторую долю 60. Исходя из этого присваивается значение свойству `_rotation` для каждой стрелки. Рассчитаем положение клипа `hourHand_mc`, чтобы оно отражало не только процент дня, но и процент текущего часа.

Наши часы, в сущности, готовы. Все, что осталось сделать, это вызывать функцию *updateClock( )*, которая будет обновлять отображение каждые 100 миллисекунд, с помощью следующей строки кода:

```
setInterval(updateClock, 100);
```

Опробуйте фильм и проверьте, работают ли ваши часы. Если не работают, сравните их с примером часов в файле *.fla*, имеющимся в сетевом хранилище кода, или сравните свой код с примером 13.7. Обновление часов 10 раз в секунду должно обеспечить плавное перемещение секундной стрелки, но это может оказаться излишеством (или уменьшит ресурсы, доступные другому фильму).

Упражнение: Модифицируйте пример 13.7 так, чтобы часы обновлялись только раз в секунду, и добавьте звук «тик-так», синхронизированный с перемещением секундной стрелки (он должен имитировать четкий щелчок).

Подумайте о том, как еще усовершенствовать приложение «часы». Сможете ли вы программно отобразить часы средствами Drawing API? А улучшить переносимость приложения, преобразовав его в компонент? А что если динамически добавлять в клип `clockFace_mc` метки часов и минут на циферблате? Сможете ли вы модифицировать приложение, чтобы оно показывало одновременно несколько часовых поясов? Попробуйте сделать секундомер, снабдив часы кнопкой сброса в нулевое положение.

Вперед!

Мы забрались так далеко, что осталось совсем немного! Досконально разобравшись с объектами и клипами, вы сможете самостоятельно справиться с большинством проектов ActionScript. Но впереди еще есть некоторый интересный материал. В главе 14 показано, как создавать подклассы *MovieClip* и компоненты, такие как UI Components, поставляемые с Flash MX.

14

Подклассы класса MovieClip и компоненты

В главе 12 мы построили пользовательский класс *Ball* и с его помощью создавали отдельные экземпляры объектов *Ball*. Когда потребовалось отобразить объекты шариков, перед нами предстало несколько возможностей. Можно поместить шарики в отдельный класс *Room* и сделать его ответственным за рисование их на экране с помощью Drawing API. Можно ввести в объект *Ball* свойство `mc`, которое будет содержать ссылку на клип, представляющий шарик. В некоторых случаях наличие объекта, который представляет сущность (например, шарик), и отдельного клипа, отображающего его на экране, оказывается избыточным. Клипы сами по себе уже являются объектами, поэтому часто оказывается разумным использовать клип одновременно для описания объекта и его отображения на экране. Во Flash MX это можно сделать, создав подкласс на базе класса *MovieClip*, который и станет специализированным типом объекта, обладающим встроенной возможностью отображения на экране.

Во Flash 5 создание подкласса *MovieClip* требовало ряда трудных обходных маневров, недоступных большинству разработчиков. Flash MX вводит формальную архитектуру для создания подклассов *MovieClip*, в которой участвуют следующие новые функции и выражения: `#initclip`, *new MovieClip( )*, *Object.registerClass( )* и параметр `initObj` функции *attachMovie( )*.

Чтобы посмотреть, как создаются подклассы *MovieClip*, мы реализуем класс *Ball* из главы 12 в виде подкласса *MovieClip*. В нашем примере каждый экземпляр шарика будет клипом, ответственным за собственное перемещение по экрану. Просмотрите код примера 14.1; мы разберем его в последующих разделах. В целом в коде использована технология ООП, изученная нами в главе 12, но с некоторыми ухищрениями, необходимыми в архитектуре создания подклассов *MovieClip*. Все примеры данной главы можно загрузить из сетевого хранилища кода.

Пример 14.1. Класс Ball, пример подкласса MovieClip

```
// ==========================================
// Создать пространство имен для хранения классов
// ==========================================
```

```
if (_global.org == undefined) {
  _global.org = new Object();
}
if (_global.org.moock == undefined) {
  _global.org.moock = new Object();
}
// ========================================
// Создать класс Ball
// ========================================
/*
 * Класс Ball. Расширяет MovieClip.
 *   Версия: 1.0.0
 *   Описание: Подкласс класса клипа для показа шариков.
 *             Требует наличия в библиотеке символа с именем ballSymbol.
 *
 * Параметры конструктора:
 *   radius          - Размер шарика (половина его диаметра)
 *   ballColor       - Цвет шарика
 *   x               - Начальная x-координата шарика
 *   y               - Начальная y-координата шарика
 *   velocity        - Вектор перемещения шарика: {x:0, y:0}
 *
 * Методы:
 *   setSize()       - Установить размер клипа в соответствии с заданным радиусом
 *   setPosition()   - Поместить клип шарика в точку с координатами (x,y)
 *   setColor()      - Задать RGB-значение цвета для клипа шарика
 *   move()          - Применить вектор перемещения шарика к его текущей позиции
 *
 * Обработчики событий:
 *   onEnterFrame()  - Вызывает move() в каждом кадре.
 */
/*
 * Конструктор класса Ball (параметры передаются через initObj функции attachMovie())
 */
org.moock.Ball = function () {
  // Создать свойства экземпляра
  this.velocity = this.params.velocity;
  // Инициализировать экземпляр
  this.setPosition(this.params.x, this.params.y);
  this.setColor(this.params.ballColor);
  this.setSize(this.params.radius);
  this.onEnterFrame = this.move;
  // Удалить из экземпляра объект params конструктора
  delete this.params;
}
/*
 * Задать MovieClip в качестве надкласса Ball
 */
org.moock.Ball.prototype = new MovieClip();
/*
 * Связать библиотечный ballSymbol с классом Ball
 */
Object.registerClass("ballSymbol", org.moock.Ball);
```

```
/*
 * Методы экземпляра
 */
/*
 * Метод: Ball.setSize()
 *   Описание: Устанавливает размер клипа шарика
 *
 * Параметры:
 *   newRadius     - Новый радиус шарика
 */
org.moock.Ball.prototype.setSize = function (newRadius) {
  this._width = this._height = (newRadius * 2);
};
/*
 * Метод: Ball.setPosition()
 *   Описание: Помещает клип шарика в точку (x, y)
 *
 * Параметры:
 *   x             - Новое положение по горизонтали
 *   y             - Новое положение по вертикали
 */
org.moock.Ball.prototype.setPosition = function (x, y) {
  this._x = x;
  this._y = y;
};
/*
 * Метод: Ball.setColor()
 *   Описание: Изменяет цвет клипа шарика
 *
 * Параметры:
 *   newColor      - Численное RGB-значение цвета, устанавливаемого для клипа
 *
 */
org.moock.Ball.prototype.setColor = function (newColor) {
  var col = new Color(this);
  col.setRGB(newColor);
};
/*
 * Метод: Ball.move()
 *   Описание: Передвигает шарик в соответствии с его вектором перемещения
 *
 * Параметры: Нет
 *
 */
org.moock.Ball.prototype.move = function () {
  this._x += this.velocity.x;
  this._y += this.velocity.y;
}
// ========================================
// ЗАПУСК ПРИЛОЖЕНИЯ
// ========================================
main();
// Функция: main()
```

```
// Описание: Точка входа в приложение
function main () {
  // Создать объект, свойства которого будут использованы как параметры конструктора Ball
  var ballParams = new Object();
  ballParams.radius = 10;
  ballParams.ballColor = 0xFF0000;
  ballParams.x = 250;
  ballParams.y = 250;
  ballParams.velocity = {x:12, y:4};

  // Присвоить ссылку на объект ballParams единственному свойству объекта initObj,
  // который мы передадим в attachMovie(). Свойства initObj копируются в каждый
  // экземпляр ball во время прикрепления.
  var initObj = new Object();
  initObj.params = ballParams;

  // Создать шарик
  this.attachMovie("ballSymbol", "redBall", 1, initObj);
}
```

Создание символа в библиотеке

Прежде чем вернуться к коду примера 14.1, создадим элементы, которые будет использовать наш фильм. Каждый подкласс клипа связывается с символом, хранящимся в библиотеке. Чтобы создавать экземпляры подкласса программным способом, требуется экспортировать этот символ для использования с ActionScript. Ниже перечислены операции, которые надо выполнить, чтобы создать и экспортировать символ для нашего класса *Ball*:

1. Создать символ с именем *ballSymbol*, содержащий круг.
2. В Library выбрать *ballSymbol*.
3. Во всплывающем меню Options для Library выбрать Linkage. Появится диалоговое окно Linkage Properties.
4. Выставить флажок Export For ActionScript.
5. В поле Identifier ввести `ballSymbol`.
6. Щелкнуть по кнопке OK.

Теперь мы создадим функцию-конструктор *Ball*, которая будет автоматически вызываться при каждом создании экземпляра *ballSymbol*.

Создание и вызов конструктора подкласса

В данный момент конструктор *Ball* (из примера 14.1) определен на главной временной диаграмме файла *.fla*. Позже мы увидим, как переместить этот конструктор в библиотечный символ *ballSymbol*. Конструктор *Ball* очень похож на функции-конструкторы, которые мы видели в главе 12, с той дополнительной особенностью, что он определен в глобальном пространстве имен, соответствующем доменному имени `moock.org` (ваши собственные классы должны быть определены в пространстве имен вашего личного домена типа `com.yoursite.YourClass`):

```
/*
 * Конструктор класса Ball.
 */
org.moock.Ball = function () {
  // Создать свойства экземпляра.
  // ...
  // Инициализировать экземпляр.
  // ...
};
```

Однако наш конструктор класса не определяет никаких параметров! Помните, что мы создаем подкласс *MovieClip*, а экземпляры клипов не создаются стандартным способом, как *new Object( )*, что способствовало бы использованию параметров. Экземпляры клипов создаются с помощью функций *attachMovie( )* или *duplicateMovieClip( )*, порождающих экземпляр и вызывающих его конструктор класса (если таковой определен). Поэтому передавать аргументы функции-конструктору *Ball* надо через *attachMovie( )* или *duplicateMovieClip( )*. Но передать аргументы непосредственно, как в синтаксисе *new Object(arg1,...argn)*, нельзя. Поэтому мы прибегаем к так называемому инициализирующему объекту *initObject*, свойства которого будут скопированы в новый экземпляр непосредственно перед выполнением функции-конструктора.

Например, следующий код прикрепляет экземпляр символа, экспортированный как "boxSymbol"":

```
var initObj = new Object();
initObj.description = "A four-sided figure";
this.attachMovie("boxSymbol", "box_mc", 2, initObj);
```

Этот код дает новому экземпляру имя "box_mc", помещает его на глубину 2 и передает ему initObject с единственным свойством description. После выполнения этого кода экземпляр box_mc будет обладать свойством description со значением "A four-sided figure".

Вместо настоящих параметров конструкторы подкласса *MovieClip* должны использовать свойства initObject в *attachMovie( )* или *duplicateMovieClip( )*. Например, наш конструктор *Ball* (показанный ниже) использует свойство params объекта initObject, которое само является объектом, для передачи в конструктор параметров x, y, velocity, radius и ballColor:

```
org.moock.Ball = function () {
  // Создать свойства экземпляра
  this.velocity = this.params.velocity;

  // Инициализировать экземпляр
  this.setPosition(this.params.x, this.params.y);
  this.setColor(this.params.ballColor);
  this.setSize(this.params.radius);
  this.onEnterFrame = this.move;

  // Удалить из экземпляра объект params конструктора
  delete this.params;
}
```

Прикрепляя новый экземпляр, мы создаем initObject следующим образом:

```
// Создать объект, свойства которого будут использованы
// как параметры конструктора Ball
var ballParams = new Object();
ballParams.radius = 10;
ballParams.ballColor = 0xFF0000;
ballParams.x = 250;
ballParams.y = 250;
ballParams.velocity = {x:12, y:4};

// Присвоить ссылку на объект ballParams единственному свойству initObj.params
var initObj = new Object();
initObj.params = ballParams;

// Теперь передать initObj в экземпляр через attachMovie(). Свойство params
// копируется в новый экземпляр вместе со всеми параметрами конструктора,
// определенными в ballParams.
this.attachMovie("ballSymbol", "redBall", 1, initObj);
```

В примере 14.1 все инструкции для создания экземпляра находятся в функции *main( )*, поэтому переменные `ballParams` и `initObj` оказываются локальными и автоматически удаляются по завершении выполнения *main( )*. В противном случае мы должны были бы удалить `ballParams` и `initObj` явно или передать объект initObject в виде литерала, чтобы он не сохранялся после передачи в *attachMovie( )*.

Может возникнуть вопрос, зачем обременять себя свойством `params`, а не определить свойства `radius`, `velocity`, `ballColor`, `_x` и `_y` непосредственно в initObject и прямо не использовать их в нашем конструкторе *Ball*, например так:

```
// Код конструктора
org.moock.Ball = function () {
  // Инициализировать экземпляр.
  this.setColor(this.ballColor);
  this.setSize(this.radius);
  this.onEnterFrame = this.move;
};

// Создание экземпляра
var initObj = new Object();
initObj.radius = 10;
initObj.velocity = {x:44, y:2};
initObj.ballColor = 0xFF0000;
initObj._x = 30;
initObj._y = 12;
this.attachMovie("ballSymbol", "redBall", 1, initObj);
```

Установить свойства непосредственно для объекта `initObj` можно, но при этом возникают две проблемы:

- Страдает централизация кода: ответственность за создание свойств экземпляра возлагается на функцию *attachMovie( )*, тогда как заниматься этим должна функция-конструктор.

- Параметры конструктора (которые должны быть временными переменными) сохраняются как постоянные свойства нового экземпляра. Например, в свойстве `ballColor` после выполнения конструктора нет необходимости, однако оно остается определенным, пока существует экземпляр.

Чтобы централизовать код и не загружать экземпляр ненужными свойствами параметров конструктора, мы определяем все параметры в одном свойстве `params` объекта initObject. Завершая свою работу, конструктор удаляет свойство `params`:

```
// Удалить объект конструктора params из экземпляра.
delete this.params;
```

Назначение MovieClip надклассом

Мы создали конструктор класса *Ball* и научились создавать экземпляры объектов *Ball* с помощью *attachMovie( )*. Теперь мы должны сделать класс *Ball* подклассом класса *MovieClip*. В главе 12 мы видели, как объявляется надкласс для класса:

```
SubClass.prototype = new SuperClass();
```

Поэтому неудивительно, что объявление *MovieClip* в качестве надкласса *Ball* выглядит так:

```
/*
 * Задать MovieClip в качестве надкласса Ball.
 */
org.moock.Ball.prototype = new MovieClip();
```

Однако хотя мы и определили *Ball* как подкласс *MovieClip*, нам еще надо сообщить интерпретатору, чтобы он выполнил конструктор *Ball* при создании нового экземпляра *ballSymbol*, экспортированного из библиотеки. Чтобы связать экспортированный символ с нужным конструктором класса, воспользуемся *Object.registerClass( )*. Ниже мы связываем *ballSymbol* с классом *Ball*:

```
/*
 * Связать библиотечный ballSymbol с классом Ball.
 */
Object.registerClass("ballSymbol", org.moock.Ball);
```

На человеческий язык это указание можно перевести так: «При создании экземпляра *ballSymbol* вызвать конструктор *org.moock.Ball( )*, так чтобы ключевое слово `this` указывало на новый экземпляр *ballSymbol*».

Инструкция *new MovieClip( )* только устанавливает класс в качестве подкласса *MovieClip* (то есть только задает наследование). С ее помощью *нельзя* создать новый клип, используемый в фильме! Создание новых экземпляров клипов или подклассов клипов, таких как *Ball*, осуществляется с помощью *attachMovie( )*, *duplicateMovieClip( )* или *createEmptyMovieClip( )*.

Подробнее об *Object.registerClass( )* см. в «Справочнике по языку».

Объединение кода подкласса с библиотечным символом

Назначив надкласс, мы создали полностью функциональный подкласс *MovieClip* с именем *Ball*. Добавление в него новых свойств и методов осуществляется через `Ball.prototype` так же, как и для любого класса (как показано в примере 14.1). Допустим, однако, что этот класс *Ball* надо использовать совместно с другим разработчиком. Разумно поместить наш класс во внешний файл *.as* и сообщить другому разработчику, что он может включить файл с классом в свой код с помощью `#include`. Но при этом нужно также сообщить ему о необходимости создать для этого класса символ в библиотеке и экспортировать его под именем `"ballSymbol"`. Если разработчик не выполнит экспорт символа или укажет при экспорте неверный идентификатор связи, класс работать не будет.

Наличие файла класса и сопутствующего символа клипа неудобно и чревато возникновением ошибок. Чтобы сделать подкласс *MovieClip* более самодостаточным, можно поместить код внутрь самого символа между прагмами `#initclip` и `#endinitclip` (создав так называемый *блок* `#initclip`).

Например, чтобы расположить класс *Ball* внутри клипа *ballSymbol*, поместим в кадр 1 временной диаграммы *ballSymbol* следующий код. (Для краткости код тела функции и ряд комментариев опущены.)

```
#initclip

// ==========================================
// Создать пространство имен для хранения классов
// ==========================================
if (_global.org == undefined) {
  _global.org = new Object();
}
if (_global.org.moock == undefined) {
  _global.org.moock = new Object();
}

// =======================================
// Создать класс Ball
// =======================================

// Конструктор класса
org.moock.Ball = function () { ... };

// Назначить MovieClip надклассом Ball
org.moock.Ball.prototype = new MovieClip();

// Связать библиотечный ballSymbol с классом Ball
Object.registerClass("ballSymbol", org.moock.Ball);

// Метод: Ball.setSize()
org.moock.Ball.prototype.setSize = function (newRadius) { ... };

// Метод: Ball.setPosition()
org.moock.Ball.prototype.setPosition = function (x, y) { ... };

// Метод: Ball.setColor()
```

```
org.moock.Ball.prototype.setColor = function (newColor) { ... };

// Метод: Ball.move()
org.moock.Ball.prototype.move = function () { ... };

#endinitclip
```

Блок `#initclip` может находиться только в кадре 1 временной диаграммы клипа. Если код размещен на главной временной диаграмме или не в кадре 1 временной диаграммы клипа, возникнет ошибка компиляции. Код внутри блока выполняется один и только один раз для каждого символа, непосредственно перед созданием первого экземпляра символа. Определив наш класс *Ball* внутри блока `#initclip` в *ballSymbol*, мы гарантируем доступность конструктора класса *Ball* перед появлением первого экземпляра *ballSymbol*. Методы и свойства, определенные *Ball*, сразу становятся доступными каждому новому экземпляру *Ball*.

Когда класс *Ball* хранится в *ballSymbol* в блоке `#initclip`, становится легко перемещать весь пакет между фильмами, перетаскивая *ballSymbol* из библиотеки одного файла *.fla* в библиотеку другого. Без всяких дополнительных усилий файл *.fla* получает возможность прикреплять и использовать экземпляры *ballSymbol*.

Подробнее об `#initclip` см. в «Справочнике по языку».

Создание компонентов

До сих пор мы не обращали внимания на то, что экземпляр нашего *ballSymbol* можно вручную перетаскивать на сцену в среде разработки Flash. В этом случае конструктор *Ball( )* будет запускаться на этапе выполнения даже для тех экземпляров, которые были помещены на сцену в среде разработки (при условии, что *Ball* был определен как подкласс перед размещением экземпляра на временной диаграмме). Но при создании экземпляра вручную во время разработки нет возможности задать параметры через initObject, как при использовании *attachMovie( )* или *duplicateMovieClip( )*. Чтобы решить эту проблему, надо определить символ клипа в виде компонента.

Компонент – это символ, предоставляющий GUI для задания свойств экземпляров в среде разработки. Обычно компоненты являются также подклассами *MovieClip*, которые используют свойства, заданные на этапе разработки, в качестве параметров конструктора.

Компоненты Flash MX представляют собой новую и улучшенную реализацию Smart Clips из Flash 5.

Создание компонентов мы более подробно изучим в главе 16. А сейчас посмотрим, как преобразовать наш *ballSymbol* в компонент. Продолжим работу с *ballSymbol* из предыдущего раздела, в блоке `#initclip` которого определен конструктор класса *Ball*.

1. Выбрать в библиотеке *ballSymbol*.

2. Во всплывающем меню Options окна Library выбрать Component Definition. Появится диалоговое окно Component Definition.
3. Нажать кнопку «+» пять раз, чтобы вставить позиции для пяти новых параметров компонента.
4. В поле Name для пяти параметров ввести `radius`, `velocity`, `color`, `x` и `y`. Это «косметические» имена параметров этапа разработки, которые увидит конечный пользователь. В них можно узнать параметры нашего конструктора *Ball*.
5. В поле Variable для этих пяти параметров ввести `cp_radius`, `cp_velocity`, `cp_ballColor`, `cp_x` и `cp_y`. Это имена свойств, которым будут присваиваться значения в каждом экземпляре *ballSymbol*. Префикс `cp_` представляет собой введенное нами соглашение, которое означает «component parameter». С его помощью мы найдем и удалим параметры компонента после того, как их конструктор завершит их использование.
6. Для этих пяти параметров вызвать всплывающее меню Type и задать Number, Object, Color, Number и Number. Это значения типа данных каждого из параметров. Тип Color предоставляет интерфейс для выбора пользователем цвета.
7. В поле Value для пяти параметров ввести `0`, `{x:0, y:0}`, `#000000`, `0` и `0`. Обратите внимание, что для `velocity` и `color` используется специальный интерфейс, упрощающий ввод свойств объекта и значений цвета. Это значения, которые устанавливаются по умолчанию, но пользователь может изменить их при перетаскивании экземпляра на сцену.
8. Щелкнуть по кнопке OK.

Определение нашего компонента *ballSymbol* показано на рис. 14.1.

В результате, когда пользователь будет перетаскивать экземпляр *ballSymbol* на сцену, инспектор свойств предоставит GUI для задания «параметров компонента» этого экземпляра, как показано на рис. 14.2.

Значения параметров, заданные пользователем в инспекторе свойств, автоматически присваиваются свойствам экземпляра *ballSymbol* (при этом берутся те имена свойств, которые мы задали раньше на шаге 5 – `cp_radius`, `cp_velocity`, `cp_ballColor`, `cp_x` и `cp_y`). Хотя можно было бы просто использовать эти свойства экземпляра «как есть», мы модифицируем конструктор *Ball*, чтобы улучшить модульность кода.

Вспомните, что наша исходная функция конструктора *Ball* требует, чтобы параметры были переданы ей в едином пакете – объекте `params`. Поэтому конструктор обращается к каждому параметру как к `this.params.`*`paramName`*, что показано в примере 14.1. Объект `params` также аккуратно хранит все параметры конструктора, чтобы их удобно было удалить после создания экземпляра. Напротив, параметры нашего компонента (`cp_radius`, `cp_velocity` и т. д.) определены непосредственно как свойства новых экземпляров *ballSymbol*, а не в объекте `params`. Так же как и в initObject функции *attachMovie( )*, можно использовать эти непосредственно заданные свойства в имеющемся виде, не привлекая функцию-конструктор. Однако это было бы посягательством

Рис. 14.1. *Определение компонента ballSymbol*

на обязанности конструктора. Правильная функция-конструктор отвечает за создание свойств экземпляра и осуществление инициализации экземпляра; она также использует параметры конструктора только для инициализации нового экземпляра.

Кроме того, обычные параметры конструктора являются локальными для функции конструктора и автоматически прекращают существование, когда конструктор завершает свою работу. Наш конструктор *Ball* заставляет параметры компонента (`cp_radius`, `cp_velocity` и т. д.) выполнить роль параметров конструктора, используя их только для инициализации нового экземпляра *Ball* и явным образом удаляя их в конце функции. В нашем конструкторе *Ball* параметры компонента являются временными и не приравнены к внутренним свойствам экземпляра. Например, можно было бы попросить поль-

Рис. 14.2. *GUI для параметров компонента ballSymbol*

зователя задать диаметр шарика в качестве параметра компонента, но получить из него радиус и сохранить в виде свойства. Проведение различия между параметрами компонента и свойствами нашего класса *Ball* делает код централизованным: свойства создаются, как это и должно быть, конструктором, а не через GUI определения компонента.

Ниже приведен исправленный код для конструктора *Ball*, превращенного в компонент. Прочий код класса *Ball* остался прежним. Обратите внимание на важность, которую приобретают комментарии, – конструктор не определяет параметры, поэтому мы должны определить необходимые параметры, основываясь на комментариях. (Для краткости в последующем коде опущены комментарии, относящиеся к номеру версии, методам и обработчикам.) Конструктор распознает параметры компонента, которые надо удалить, по префиксу `cp_`, который мы задали ранее в диалоговом окне Component Definition.

```
/*
 * Класс Ball. Расширяет MovieClip.
 *
 * Параметры компоненты:
 *   cp_radius           - Размер шарика
 *   cp_ballColor        - Цвет шарика
 *   cp_x                - Начальная x-координата шарика
 *   cp_y                - Начальная y-координата шарика
 *   cp_velocity         - Вектор перемещения шарика: {x:0, y:0}
 *
 * Методы: ...
 *
 * Обработчики событий: ...
 */
/*
 * Конструктор класса. параметры передаются через initObj функции attachMovie().
 */
org.moock.Ball = function () {
  // Создать свойства экземпляра.
  this.velocity = this.cp_velocity;

  // Инициализировать экземпляр.
  this.setPosition(this.cp_x, this.cp_y);
  this.setColor(this.cp_ballColor);
  this.setSize(this.cp_radius);
  this.onEnterFrame = this.move;

  // Удалить из экземпляра параметры компонента (свойства, имена которых
  // начинаются с префикса cp_, являются параметрами компонента).
  for (var p in this) {
    if (p.indexOf("cp_") != -1) {
      delete this[p];
    }
  }
};
```

Конечно, теперь, после модификации конструктора *Ball*, мы должны модифицировать initObject, чтобы он использовал имена свойств компонента при

создании экземпляров *ballSymbol* с помощью *attachMovie( )*. Это позволит использовать одну и ту же функцию-конструктор независимо от того, создается ли компонент вручную на этапе разработки или программно на этапе выполнения.

Вот модифицированный вариант с применением *attachMovie( )*:

```
var initObj = new Object();
initObj.cp_radius = 10;
initObj.cp_ballColor = 0xFF0000;
initObj.cp_x = 250;
initObj.cp_y = 250;
initObj.cp_velocity = {x:12, y:4};

// Создать шарик.
this.attachMovie("ballSymbol", "redBall", 1, initObj);
```

Если все экземпляры подкласса клипа будут создаваться программно с помощью *attachMovie( )* или *duplicateMovieClip( )*, конструктор должен принимать параметры, которые прикреплены к единственному объекту `params` в объекте initObject. Однако при создании компонентов, экземпляры которых будут создаваться на этапе разработки, конструктор должен принимать отдельные параметры компонента, начинающиеся с префикса `cp_` (а при создании экземпляров компонента программным способом следует прикрепить одноименные параметры непосредственно к initObject).

Упаковка графики компонента

Многие визуальные компоненты динамически создают свое изображение на экране на этапе выполнения путем прикрепления экспортированных клипов к экземплярам компонентов. При перемещении компонента, зависящего от экспортированных элементов, из одного файла *.fla* в другой каждый такой элемент должен быть перемещен в библиотеку целевого файла. Во Flash MX нет формальных средств для упаковки компонента, состоящего из нескольких символов, в один объект. Однако традиционный подход состоит в размещении всех элементов в *направляющем слое* (*guide layer*) внутри компонента (см. описание направляющих слоев в электронной справке Flash). Элементы, расположенные в направляющем слое, автоматически перемещаются при перетаскивании компонента между библиотеками, но не видны в компоненте на этапе выполнения (если только компонент не прикрепит их явным образом).

Шаблон компонента

Ниже представлен код общего шаблона компонента, реализующего подкласс *MovieClip*. В шаблоне не используется пространство имен типа `org.moock`, как в нашем примере *Ball*. Кроме того, шаблон не хранит все параметры конструктора в едином объекте `params`. Эти дополнительные приемы вы можете реализовать по своему усмотрению. Однако в шаблоне предполагается, что параметры конструктора имеют префикс `cp_`, и после создания объекта все параметры компонента удаляются. Хотя удалять параметры конструктора не обязательно, такая практика написания кода полезна.

```
// КОД В КАДРЕ 1 БИБЛИОТЕЧНОГО СИМВОЛА someClassSymbol.

#initclip

/*
 * Класс SomeClass. Расширяет MovieClip
 *   Версия: 1.0.0
 *   Описание: Здесь находится ваше описание
 *
 * Параметры компонента:
 *   cp_param1       - Краткое описание
 *   cp_param2       - Краткое описание
 *
 * Методы:
 *   someMethod()    - Краткое описание
 */

/*
 * Конструктор класса
 * (Параметры передаются через initObj или устанавливаются как параметры компонента)
 */
_global.SomeClass = function () {
  // Создать свойства экземпляра
  this.prop1 = this.cp_param1;
  // Инициализировать экземпляр
  this.someMethod(this.cp_param2);
  // Удалить из экземпляра параметры (с префиксом cp_)
  for (var p in this) {
    if (p.indexOf("cp_") != -1) {
      delete this[p];
    }
  }
};

/*
 * Задать MovieClip в качестве надкласса SomeClass
 */
SomeClass.prototype = new MovieClip();

/*
 * Связать библиотечный someClassSymbol с классом SomeClass
 */
Object.registerClass("someClassSymbol", SomeClass);

/*
 * Методы экземпляра
 */

/*
 * Метод: SomeClass.someMethod()
 *   Описание: Краткое описание
 *
 * Параметры:
 *   param     - Краткое описание
 *
 */
```

```
SomeClass.prototype.someMethod = function (param) {
  trace("someMethod invoked with parameter of: " + param);
};

#endinitclip
```

В следующем коде показан пример применения шаблона *SomeClass*. Чтобы посмотреть на экземпляр *SomeClass*, созданный вручную, загрузите файл *componentTemplate.fla* из хранилища кода.

```
// Создать экземпляр
var initObj = new Object();
initObj.cp_param1 = 15;
initObj.cp_param2 = "Hello world";
this.attachMovie("someClassSymbol", "someClassInstance", 1, initObj);
```

Подклассы подклассов MovieClip

Как и обычный класс, подкласс *MovieClip* можно расширить посредством нового подкласса. Если родительский класс определен вне соответствующего ему символа (как в самом первом примере *Ball* в этой главе), определить его дочерний подкласс можно так:

- Присвоить прототипу дочернего класса новый экземпляр родительского класса
- Зарегистрировать дочерний класс в его символе с помощью *Object.registerClass( )*

Не забудьте также вызвать родительский конструктор из конструктора дочернего класса. Например, следующий код показывает, как создать подкласс *Baseball* из существующего класса *Ball*:

```
// Конструктор класса Baseball
org.moock.Baseball = function () {
  // ... код инициализации ...

  // Вызвать конструктор класса Ball
  super();
}

// Назначить Ball надклассом Baseball
org.moock.Baseball.prototype = new org.moock.Ball();

// Связать класс Baseball с собственным символом в библиотеке
Object.registerClass("baseballSymbol", org.moock.Baseball);
```

Однако когда внутри блока `#initclip` символа определены подкласс (*Ball*) и подкласс этого подкласса (*Baseball*), надо обеспечить инициализацию подкласса перед инициализацией любого из его собственных подклассов. Для этого применяется параметр `order` прагмы `#initclip`, который является неотрицательным целым числом, задающим порядок выполнения конкретного блока `#initclip` относительно всех остальных блоков `#initclip` в фильме. Блоки `#initclip` с меньшими значениями *order* выполняются раньше, чем блоки с большими значениями. Все блоки `#initclip`, для которых не задан параметр *order*, выполняются раньше блоков с заданным параметром *order*.

Например, чтобы создать класс *Baseball* внутри клипа *baseballSymbol*, мы должны заменить первую строку кода класса *Ball* следующей:

```
#initclip 0
```

После этого можно создать класс *Baseball* так:

```
// Использовать более высокий порядок #initclip, чем у Ball, чтобы гарантировать
// доступность Ball к моменту выполнения этого кода.
#initclip 1

// ================================================
// Создать пространство имен для хранения классов
// ================================================
if (_global.org == undefined) {
  _global.org = new Object();
}
if (_global.org.moock == undefined) {
  _global.org.moock = new Object();
}

// ================================================
// Создать класс Baseball
// ================================================

/*
 * Класс Baseball. Расширяет Ball.
 *   Версия: 1.0.0
 *   Описание: Подкласс клипа для отображения baseball
 *
 * Параметры конструктора:
 *   signature       - Сигнатура шарика
 *
 * Методы:
 *   setSignature()  - Установить сигнатуру для шарика
 */

/*
 * Конструктор класса. Параметры передаются через объект initObj метода attachMovie()
 */
org.moock.Baseball = function () {
  // Создать свойства экземпляра.
  this.sig = null;

  // Инициализировать экземпляр, прежде чем вызывать конструктор Ball.
  // (Иначе Ball удалит объект params прежде, чем мы успеем его использовать.)
  this.setSignature(this.params.signature);
  // Вызвать конструктор класса Ball.
  super();

  // Не требуется удалять объект params, поскольку это сделает конструктор Ball.
};
/*
 * Назначить Ball надклассом Baseball.
 */
```

```
org.moock.Baseball.prototype = new org.moock.Ball();

/*
 * Связать библиотечный baseballSymbol с классом Baseball
 */
Object.registerClass("baseballSymbol", org.moock.Baseball);
/*
 * Методы экземпляра
 */

/*
 *     Метод: Ball.setSignature()
 *  Описание: Устанавливает сигнатуру шарика
 *
 * Параметры:
 *   newSig     - Новая сигнатура шарика
 */
org.moock.Ball.prototype.setSignature = function (newSig) {
  this.sig = newSig;
};

#endinitclip
```

Обратите внимание, что у нескольких блоков `#initclip` может быть одинаковое значение *order*. Например, если мы создадим три подкласса *Ball*, то можем во всех их блоках `#initclip` сделать параметр `order` равным 1. Значение имеет только то, что в *Ball* порядок блоков `#initclip` ниже, чем во всех его подклассах.

Резюме

В этой главе мы изучили три взаимосвязанных, но различных типа подклассов *MovieClip*:

- Подкласс, определенный вне своего библиотечного символа.
- Подкласс, определенный в своем библиотечном символе (внутри директивы `#initclip`).
- Компонент, позволяющий задавать свои параметры с помощью GUI на этапе разработки.

Тип подкласса *MovieClip*, который вам следует использовать в своем приложении, целиком зависит от вашей ситуации. Если надо разрешить размещение экземпляров на этапе разработки, то вполне вероятно, что придется создать компонент. Аналогично, если вы собираетесь распространять свой подкласс, передавая его другим разработчикам, то компонент способствует созданию хорошего транспортного механизма. В противном случае шаг определения компонента можно пропустить. Но, как правило, следует определить подкласс непосредственно внутри связанного с ним библиотечного символа в рамках блока `#initclip`. Хотя это не требуется правилами, но хранение подкласса в связанном с ним символе облегчает обслуживание пакета в целом.

Компоненты и подклассы клипов – два краеугольных камня технологии разработки приложений Flash MX. Создавая библиотеки кода в виде компонентов, можно быстро собирать приложения из повторно используемых элементов. Macromedia включила во Flash MX ряд полезных компонентов интерфейса пользователя (UI Components). Обязательно ознакомьтесь с этими компонентами, их методами и параметрами (см. приложение G). Исходный код Flash UI Components также поставляется вместе с Flash MX, поэтому можно изучать эти компоненты и расширять их, создавая свои собственные.

Вот материалы для дополнительного изучения компонентов:

Введение в учебный материал по компонентам

В среде разработки Flash MX выберите Help → Tutorials → Introduction to Components.

Применение компонентов

В среде разработки Flash MX выберите Help → Using Flash → Using Components.

Создание адресной книги с помощью компонентов (Colin Moock)

http://www.macromedia.com/desdev/mx/flash/articles/addressbook.html

Создание форм и компонентов в Macromedia Flash MX

http://www.macromedia.com/support/flash/applications/creating_forms/

Создание компонентов в Macromedia Flash MX

http://www.macromedia.com/support/flash/applications/creating_comps/

Создание компонента Flash MX

http://www.flashcomponents.net/tutorials/triangle/triangle.html

Расширение компонентов в Macromedia Flash MX

http://www.macromedia.com/desdev/mx/flash/articles/fmx_components.html

Модификация компонентов ListBox и ComboBox

http://www.macromedia.com/desdev/mx/flash/articles/extending_components.html

Коллекция компонентов Macromedia Flash

http://www.macromedia.com/exchange/flash

Коллекция компонентов сторонних разработчиков

http://www.flashcomponents.net/

Вперед!

Компоненты и подклассы клипов – последняя большая тема в изучении основ языка ActionScript. Прежде чем перейти к «Справочнику по языку», мы поговорим о *лексической структуре* (о тонкостях синтаксиса ActionScript) и объединим все вместе в заключительном учебном приложении главы 17. Дополнительные сведения о процедуре определения компонентов см. в главе 16.

15

Лексическая структура

Лексическая структура (*lexical structure*) языка – это набор правил, управляющих его синтаксическим построением. Этим правилам необходимо следовать при написании исходного кода сценариев.

Пробельные символы

Символы табуляции, пробела и возврата каретки (то есть перевода строки) используются в ActionScript так же, как и в человеческом языке: чтобы отделять слова друг от друга и не позволять им *сливатьсявместевоттакимобразом*. На жаргоне программистов такие символы называются *пробельными* (*whitespaces*) и используются в исходном коде для разделения *лексем* (*tokens*). Ключевые слова, идентификаторы и выражения примерно соответствуют словам, фразам и предложениям обычного языка. Вот пример правильного и неправильного использования пробельных символов:

```
varx   // Плохо! нет пробельного символа между ключевым словом var и переменной x.
var x  // Теперь лучше… благодаря пробельному символу интерпретатор
       // сможет понять код.
```

Пробельный символ не обязателен, если есть какой-то другой *разделитель* (*delimiter*), сообщающий ActionScript о конце одной лексемы и начале другой. Следующий код совершенно корректен, так как операторы =, + и / отделяют `x`, 10, 5 и `y` друг от друга:

```
x=10+5/y;         // Скученно, но допустимо
x = 10 + 5 / y;   // Легче прочесть, но равносильно предыдущему
```

Точно так же пробельные символы не обязательны, если есть другие символы, действующие как разделители, такие как квадратные, круглые или фигурные скобки, запятые, символы «больше» или «меньше». Следующие выражения совершенно законны, хотя и могут вызвать приступ клаустрофобии:

```
for(var i=0;i<10;i++){trace(i);}
if(x==7){y=[1,2,3,4,5,6,7,8,9,10];}
myMeth=function(arg1,arg2,arg3){trace(arg1+arg2+arg3);};
```

Добавление избыточных пробельных символов – дело вкуса, поскольку они игнорируются интерпретатором ActionScript. Несмотря на это существуют соглашения, которых следует придерживаться, чтобы сделать код более читаемым. Например, вышеприведенное выражение присваивания можно переписать и так, но очевидно, что прочесть его становится труднее:

```
x =
  10
  + 5
  / y;
```

Обратите внимание, что инструкция завершается точкой с запятой, а не переводом строки. Почти всегда переводы строк несущественны и не выступают в качестве символов, завершающих инструкцию. Поэтому часто один или более переводов строки (вместе с пробелами или символами табуляции) добавляются для того, чтобы облегчить чтение сложных предложений:

```
myNestedArray = [[x, y, z],
                 [1, 2, 3],
                 ["joshua davis", "yugo nakamura", "james patterson"]];
// Значительно красивее, чем:
myNestedArray = [[x, y, z], [1, 2, 3],["joshua davis","yugo nakamura",
"james patterson"]];
// И красивее, чем:
myNestedArray = [[x, y, z],
[1, 2, 3],
["joshua davis","yugo nakamura","james patterson"]];
```

Для того чтобы разбить предложение на несколько строк, никаких особых действий не требуется. Просто добавьте возврат каретки и продолжайте ввод. Полезные советы по поводу улучшения читаемости кода можно найти в книге Стива Мак-Конелла (Steve McConnell) «Code Complete» (Microsoft Press). Однако, как будет видно из следующего раздела, иногда перенос строки может ошибочно интерпретироваться как окончание инструкции.

Обратите внимание, что пробел *внутри* кавычек, в которые заключена строка, имеет значение, тогда как пробел *вне* кавычек игнорируется. Сравните следующие два примера:

```
x = 5;
trace("The value of x is"  +   x);   // Выводит: "The value of x is5"
trace("The value of x is "+x);       // Выводит: "The value of x is 5"
```

Символы, завершающие инструкции (точка с запятой)

Как мы видели в главе 6, точка с запятой завершает инструкцию ActionScript. Принято всегда завершать инструкцию точкой с запятой, но, строго говоря, в ActionScript это не обязательно. Если точка с запятой опущена, интерпретатор пытается догадаться, где заканчивается инструкция. Например:

```
// Такой формат предпочтителен
var x = 4;
var y = 5;
// Но такой тоже допускается
var x = 4
var y = 5
```

Интерпретатор ActionScript предполагает, что в этом коде перенос строки завершает инструкцию. (В более строгих языках, например в C, компилятор сообщит об ошибке.) Однако пропуск точки с запятой в инструкциях программного кода подобен пропуску точек в предложениях при обычном письме: большинство предложений будет, вероятно, понятно читателю, но всегда может возникнуть путаница, не говоря уже о том, что чтение потребует больших усилий. Посмотрим, например, что произойдет, если опустить точку с запятой после инструкции *return*:

```
function addOne (value) {
  return
  value + 1
}
```

ActionScript предположит, что имелось в виду следующее:

```
function addOne (value) {
  return;
  value + 1;
}
```

Вместо возврата `value + 1` функция всегда будет возвращать `undefined`, потому что ключевое слово `return` само по себе является допустимым предложением. Даже если `return` находится на строке в одиночестве, и даже если после `value + 1` добавлена точка с запятой, инструкция *return* рассматривается как законченная.

Чтобы избежать такого рода неоднозначности, полезно всегда ставить точку с запятой. Более того, в данном конкретном случае не отделяйте ключевое слово `return` от его выражения разрывом строки, поскольку это изменяет значение инструкции. Поэтому предшествующий пример должен быть записан так:

```
function addOne (value) {
  return value + 1;
}
```

Аналогично оператор `++` или `--` всегда должен быть на одной строке со своим операндом:

```
// Правильно
x++
// Неправильно
y
++
```

Обратите внимание, что точка с запятой завершает отдельные инструкции, но не требуется там, где есть разделители блока инструкций (фигурные скобки). Например:

```
for (var i=0;i<10;i++) {  // Здесь нет точки с запятой
  trace(i);               // Здесь точка с запятой
}                         // Здесь нет точки с запятой

if(x == 10) {             // Здесь нет точки с запятой
  trace("x is ten");      // Здесь точка с запятой
} else {                  // Здесь нет точки с запятой
  trace("x is not ten");  // Здесь точка с запятой
}                         // Здесь нет точки с запятой

on (release) {            // Здесь нет точки с запятой
  trace("Click");         // Здесь точка с запятой
}                         // Здесь нет точки с запятой
```

Однако точка с запятой обязательно должна стоять за литералом функции:

```
function (param1, param2, ... paramn) { statements };
```

Специальная директива `#include` и прагмы `#initclip` и `#endinitclip` не должны заканчиваться точкой с запятой, иначе возникает ошибка. О правильном их использовании рассказано в «Справочнике по языку».

Комментарии

Так называемые *комментарии* представляют собой текст, который игнорируется интерпретатором и вводится программистом исключительно для удобства людей, читающих код. Комментарии призваны пояснять действия, выполняемые кодом, предоставлять сведения для контроля версий или описывать другую важную информацию, например об использовании структуры данных или о причинах тех или иных программных решений. Комментарии должны описывать код на понятийном уровне, а не просто отражать синтаксис самого кода. Например, данный комментарий бесполезен:

```
// Присвоить x значение 5
var x = 5;
```

в то время как следующий комментарий объясняет, почему `x` присваивается значение 5, что помогает понять логику кода:

```
// Инициализировать x-координату игрока
var x = 5;
```

Обратите внимание, что можно значительно приблизиться к созданию «самодокументированного кода», если давать переменным и обработчикам содержательные имена. Какой из вариантов представленного ниже кода понятнее?

```
x = y / z;                      // Весьма таинственно
average = sum / numberOfItems;  // Едва ли требует объяснения
```

ActionScript поддерживает как однострочные комментарии, так и расширенные многострочные. Однострочники, встречавшиеся в этой книге, начинаются с двух символов косой черты (//):

```
// Адресовано человеку... интерпретатор это не прочтет
```

Однострочные комментарии автоматически завершаются переносом строки. Чтобы поместить дополнительный текст комментария на следующей строке, надо повторить символы //:

```
// Здесь начало комментария...
// ...а здесь его продолжение.
```

Однострочные комментарии могут следовать за программным кодом на той же строчке:

```
var x;    // Это допустимый комментарий на одной строчке с кодом
```

ActionScript поддерживает и многострочные комментарии, предназначенные для размещения крупных блоков комментариев к коду и *не* завершающиеся с переводом строки. Многострочные комментарии начинаются с последовательности из двух символов /* и продолжаются, пока не встретится завершающая последовательность из двух символов */, например:

```
/* -----BEGIN VERSION CONTROL INFO----
  Name: MyApplication, Version 1.3.1
  Author: Killa Programma
  Last Modified: August 07, 2002
   ------END VERSION CONTROL INFO-----
*/
```

Многострочными комментариями можно пользоваться только в режиме Expert Mode в панели Actions или во внешних файлах *.as*, подключаемых с помощью директивы #include. Многострочные комментарии преобразуются в однострочные при переключении из Expert Mode в Normal Mode. Во Flash MX переключение обратно в Expert Mode восстанавливает многострочные комментарии. Обратите внимание, что разрешены вложенные комментарии:

```
/* Это пример
/* вложенного // комментария */

/* Это еще один пример
/* вложенного комментария */
Комментарий заканчивается, когда будет обнаружен этот второй ограничитель */
```

Комментарии следующего вида не приняты, но допустимы:

```
/* За этим комментарием следует действительный код */ var x = 5;
```

При внесении в код крупных (или мелких) исправлений распространена практика включения в комментарий инициалов и даты изменения:

```
// Исправлена ошибка в финансовых расчетах. BAE 12-04-02. V1.01
```

Комментарии позволяют временно отключить код (или отключить устаревший код навсегда), не удаляя его. Код, отключенный путем превращения

его в комментарий, называют *закомментированным*. Он легко восстанавливается путем удаления ограничителей комментария:

```
/* Отключу пока этот блок кода
duplicateMovieClip("character", "newCharacter", 1);
newCharacter._rotation = 90;
*/
```

Можно отключить одну строку кода, просто поместив в ее начало два символа косой черты:

```
// newCharacter._rotation = 90;
```

Зарезервированные слова

Некоторые *зарезервированные слова* для интерпретатора ActionScript обозначают специфические встроенные элементы языка, например инструкции и операторы. Следует избегать их употребления в качестве идентификаторов в собственном коде. Использование этих слов в иных целях в большинстве случаев приводит к ошибкам. Зарезервированные слова ActionScript перечислены в табл. 15.1.

Таблица 15.1. Зарезервированные слова ActionScript

add[a]	delete	ge*	le*	onClipEvent	this
and*	do	gt*	lt*	or*	typeof
break	else	if	ne*	return	var
case	eq*	ifFrameLoaded*	new	super	void
continue	for	in	not*	switch	while
default	function	instanceof	on	tellTarget*	with

[a] Зарезервированные слова Flash 4, устаревшие во Flash 5.

Следует также избегать употребления ключевых слов, перечисленных в табл. 15.2. Они не входят в число зарезервированных для ActionScript во Flash MX, но могут стать частью языка в будущем, так как запланированы для возможного применения стандартом ECMA-262.

Таблица 15.2. Слова, которые могут быть зарезервированы в будущем

abstract	const	final	int	protected	transient
boolean	debugger	finally	interface	public	try
byte	double	float	long	short	volatile
catch	enum	goto	native	static	
char	export	implements	package	synchronized	
class	extends	import	private	throws	

Помимо формально определенных ключевых слов, следует избегать использования в коде имен встроенных свойств, методов и объектов, чтобы не изменить режим их действия, установленный по умолчанию.

Следующий код неудачен, поскольку переменная *Date* перекрывает встроенный конструктор с таким же именем:

```
Date = new Object();  // Ай! Мы только что отключили конструктор Date()
```

В результате становится невозможным создание объектов *Date*:

```
var now = new Date();  // Устанавливает now в undefined
trace(now);            // Выводит пустую строку вместо времени и даты
```

Идентификаторы

Все переменные, функции и свойства объектов именуются с помощью идентификаторов. Идентификаторы ActionScript должны образовываться по следующим правилам:

- Идентификаторы должны начинаться с буквы, символа подчеркивания или знака доллара (но не цифры).
- Идентификаторы не должны совпадать с зарезервированными словами.
- Для верности идентификаторы должны содержать только буквы (*A–Z* или *a–z*), цифры, символ подчеркивания и знак доллара. Особенно следите за тем, чтобы в идентификаторах не было пробелов, точек, обратной косой черты и прочих знаков пунктуации. Однако фактически в идентификаторах допускается наличие большей части символов Unicode согласно правилам, приведенным в разделе 3.16 стандарта Unicode. (См. *http://www.unicode.org/unicode/standard/standard.html*.)
- В идентификаторах нельзя задавать символы с помощью escape-последовательностей Unicode (\uxxxx). (Это расхождение с ECMA-262, v3, допускающим escape-последовательности в идентификаторах.)
- О чувствительности идентификаторов к регистру символов рассказано в следующем разделе.

Хотя и необязательно, но считается хорошим стилем следовать приведенным правилам и при составлении имен экземпляров клипов, символов, идентификаторов экспортируемых символов, меток кадров и имен слоев. Информацию об использовании стандартных суффиксов в идентификаторах (таких как `_mc` для обозначения клипа) см. в табл. 2.1 главы 2.

Чувствительность к регистру

Если язык полностью чувствителен к регистру букв, то все лексемы языка, включая идентификаторы и ключевые слова, должны вводиться с точным соблюдением заглавных букв. Например, в языках, чувствительных к регистру, инструкция:

```
If (x == 5) {
  x = 10;
}
```

вызовет ошибку, потому что ключевое слово `if` ошибочно записано с заглавной буквы – `If`. Кроме того, в языках, чувствительных к регистру, следующие две инструкции объявляют две различные переменные (одну с именем `firstName` и другую с именем `firstname`):

```
var firstName = "doug";
var firstname = "terry";
```

Спецификация ECMA-262, на которой базируется ActionScript, требует полной чувствительности к регистру. Однако ActionScript отходит от стандарта в этой области, чтобы обеспечить обратную совместимость с фильмами Flash 4. В ActionScript ключевые слова, приведенные в табл. 15.1, чувствительны к регистру, а идентификаторы, метки кадров и ID экспортируемых символов – нет. В следующем примере в ключевом слове `onClipEvent` некорректно использованы заглавные буквы, что превращает его в `onclipevent` и вызывает ошибку:

```
// ОШИБКА! Должно быть onClipEvent (enterFrame) {
onclipevent (enterFrame) {
  this._x++;
}
```

Однако, в отличие от ключевых слов, идентификаторы в ActionScript не чувствительны к регистру, поэтому следующие инструкции будут присваивать значение *одной и той же* переменной:

```
var firstName = "margaret";
var firstname = "michael";
trace(firstName);   // Выводит "michael"
trace(firstname);   // Тоже выводит "michael" (переменные одинаковы)
```

Даже внутренние идентификаторы, такие как имена свойств и функций, не чувствительны к регистру в ActionScript. Следующая строка вызывает ошибку в JavaScript, но благополучно создает массив в ActionScript:

```
myList = new array();   // Предпочтительнее запись new Array();
```

Из-за этого при переносе кода JavaScript в ActionScript или при изучении программистами JavaScript языка ActionScript могут возникнуть трудности. Чувствительность к регистру часто служит в JavaScript для использования одного и того же имени с разными целями. Например, инструкция

```
date = new Date();   // Действует в JavaScript, но не в ActionScript
```

ведет к разрушительным последствиям в ActionScript, который не отличает идентификатор `date` от класса объектов *Date*. В ActionScript этот код отключает встроенный класс *Date*. Поэтому необходимо следить, чтобы не один только регистр отличал идентификаторы один от другого или от всех пред-

определенных идентификаторов типа *Date* или *Array*. Вот как можно переписать предыдущий код в ActionScript:

```
theDate = new Date(); // Так надо писать в ActionScript
```

Метки кадров и идентификаторы экспортируемых символов тоже не чувствительны к регистру. Например, следующая функция *gotoAndPlay( )* будет правильно работать, даже если нужному кадру присвоена метка `INTRO`, а не `Intro`:

```
gotoAndPlay("Intro");
```

Аналогично символ, экспортируемый как «SHIP», можно прикрепить так:

```
this.attachMovie("ship", "enemyShip", 1);
```

Однако допускать такое несоответствие регистра на практике не рекомендуется. Главное – во всех случаях стремиться к единообразию, даже если язык строго не требует этого. Единообразное выделение заглавными буквами переменных, функций, экземпляров и других элементов облегчит чтение кода и не позволит возможным будущим изменениям в правилах ActionScript, касающихся использования регистров, уничтожить плоды наших трудов.

Вперед!

Что касается основ языка ActionScript, то с ними мы закончили. Теперь вы абсолютно готовы к чтению и написанию кода ActionScript. В главе 16 рассказывается о среде разработки ActionScript.

16

Среда разработки ActionScript

В этой главе изучаются практические стороны написания кода ActionScript. Будут рассмотрены следующие темы:

- Размещение кода на кнопках, клипах и кадрах с помощью панели Actions
- Загрузка кода из внешнего файла
- Создание компонентов (многократно используемых модулей кода)

Панель Actions

Панель Actions – это среда разработки ActionScript во Flash. Все сценарии фильма создаются в панели Actions, открываемой с помощью Window → Actions (клавиша F9).

Панель Actions может работать в режиме эксперта (Expert Mode) и в обычном режиме (Normal Mode). В режиме Expert Mode панель Actions разделена на две части, как показано на рис. 16.1: окно инструментов (слева) и окно сценариев (справа).

В окне сценариев находится код, прикрепленный к выбранному в данный момент кадру, кнопке или клипу. Окно инструментов действует и как краткий справочник, и как средство помещения кода в окно сценариев. Двойной щелчок по любому элементу в окне инструментов добавляет этот элемент в окно сценариев. Можно и перетаскивать элементы из окна инструментов в окно сценариев.

Заголовок панели Actions указывает, где находится текущий код: в кадре («Actions – Frame»), на кнопке («Actions – Button») или клипе («Actions – Movie Clip»). Если к текущему выбранному элементу нельзя прикрепить код, то заголовок панели Actions выглядит просто как «Actions», а окно Script отключается.

Кнопка всплывающего меню Options в панели Actions предоставляет доступ к различным инструментам и настройкам, таким как печать или вывод номеров строк. Панель навигации по сценариям показывает точное местонахож-

Рис. 16.1. *Панель Actions*

дение текущего сценария и обеспечивает быстрое перемещение между сценариями текущей временной диаграммы.

Структура элементов в окне инструментов несколько отличается от категорий, использовавшихся при описании возможностей ActionScript в данной книге. Прежде всего, *инструкции* не выделены в особую папку, а классы и объекты объединены вместе в папке *Objects*. В этой книге проводится различие между инструкциями, классами и объектами и соблюдается более традиционная терминология программирования. Алфавитный список элементов языка можно получить с помощью указателя в нижней части окна инструментов и, конечно, в «Справочнике по языку».

Режимы редактирования

Панель Actions может работать в двух разных режимах – обычном режиме (Normal Mode) и режиме эксперта (Expert Mode), которые определяют способ помещения кода в окно сценариев.

Обычный режим

В обычном режиме новые инструкции создаются с помощью окна инструментов, а окно сценариев предназначено для просмотра этих предложений. Чтобы поместить в окно сценариев новую инструкцию, находясь в обычном режиме, надо дважды щелкнуть по требуемому элементу Action или перетащить его из окна инструментов в окно сценариев. Чтобы добавить инструкцию с произвольным кодом, надо выбрать действие *evaluate* (через Actions →

Miscellaneous Actions) и ввести инструкцию в поле Expression окна Parameters (не показано на рис. 16.1) в верхней части панели Actions.

После добавления инструкции в окно сценариев ее можно настроить через окно Parameters в верхней части панели Actions. Структура и содержимое окна Parameters меняются в зависимости от инструкции, выбранной в окне сценариев. Обратите внимание, что при работе панели Actions в обычном режиме *нельзя* непосредственно вводить код в окно сценариев с клавиатуры. В обычном режиме окно сценариев служит не для редактирования текста, а предоставляет доступный только для чтения список инструкций. Изменения в инструкции, находящиеся в окне Script, вносятся с помощью окна Parameters.

Обычный режим удобен для начинающих программистов, но создает ограничения при разработке реальных проектов. Поэтому в данной книге обсуждается только создание кода в режиме эксперта. Однако, за исключением многострочных комментариев (которые можно добавлять только в режиме эксперта), любой код, созданный в одном режиме, можно создать и в другом.

Режим эксперта

В режиме эксперта окно сценария функционирует как обычное окно для редактирования текста. При создании или изменении кода в режиме эксперта обычно непосредственно вводят текст в окно сценариев. Однако сохраняется и возможность добавления кода путем двойного щелчка по элементу окна инструментов. Окно параметров в режиме эксперта не используется, а параметры вводятся непосредственно в окно сценариев.

Задание режима редактирования

Во Flash 5 режим редактирования в панели Actions устанавливается индивидуально для каждого кадра и для каждого объекта. Это означает, что Flash запоминает режим редактирования панели Actions, выбранный для каждого объекта и каждого кадра в фильме. Если выбрать обычный режим для кадра 2 и режим эксперта для кадра 3, то панель Actions автоматически переключится в обычный режим во время редактирования кода в кадре 2 и в режим эксперта во время редактирования кода в кадре 3.

Во Flash MX устанавливается постоянный режим редактирования в панели Actions посредством всплывающего меню Options, показанного на рис. 1.2 в главе 1.

Во Flash 5 режим редактирования можно установить глобально с помощью Edit → Preferences → General → Actions Panel → Mode.

Во Flash 5 переключение из режима эксперта в обычный разрушает имеющееся форматирование исходного кода. Новый формат кода будет соответствовать стандартным правилам форматирования во Flash: без лишних пробельных символов и с отступами для блоков текста в стиле, соблюдающемся в данной книге. Во Flash MX форматирование исходного кода сохраняется при переключении режимов.

Помещение сценариев в кадры

Структура документов Flash основана на анимации. Каждый документ Flash содержит линейную последовательность *кадров*, или фрагментов видео- и звукового содержимого. Большая часть кода ActionScript привязана к этой кадровой структуре. Помещая блок кода в кадр, мы задаем момент выполнения этого кода в процессе воспроизведения фильма. Добавляя код в кадр, всегда следует помнить не только о том, какие действия произведет этот код, но и *когда* этот код должен быть выполнен.

Например, надо написать код, который помещает на сцену новый клип, когда выводится 20-й кадр фильма:

```
_root.attachMovie("myClip", "clip1", 0);
```

Для того чтобы этот код выполнился при выводе 20-го кадра фильма, он должен быть прикреплен к ключевому кадру этого кадра.

Код ActionScript, прикрепляемый к кадру, должен находиться в ключевом кадре. Чтобы создать ключевой кадр, к которому можно прикрепить код, выберите Insert → Keyframe или Insert → Blank Keyframe.

Чтобы прикрепить код к ключевому кадру, надо выделить ключевой кадр на временной диаграмме, открыть панель Actions и ввести код в окно сценариев. Во время воспроизведения фильма перед выводом содержимого кадра выполняется код, находящийся в ключевом кадре.

Код в ключевых кадрах предназначен для выполнения задач, синхронизированных с воспроизведением фильма, и для задания программных элементов, таких как переменные, функции, объекты, которые будут использоваться на протяжении всего клипа или фильма. Простым примером синхронизированной задачи служит цикл временной диаграммы. Пусть к кадру номер 15 прикреплен такой код:

```
gotoAndPlay(10);
```

Когда головка воспроизведения доходит до кадра 15, выполняется этот код, и воспроизведение фильма опять начинается с кадра 10. Когда кадр 15 будет достигнут второй раз, воспроизведение фильма опять продолжится с кадра 10. В результате воспроизведение фильма зацикливается между кадрами 10, 11, 12, 13, 14 и 15 до бесконечности.

Однако код, находящийся в ключевых кадрах, не всегда управляет головкой воспроизведения или синхронизируется с ней. Ключевые кадры можно также использовать для представления состояний приложения (например, как точку останова фильма, как для каждого вопроса в главе 1) или просто как контейнеры для хранения функций, переменных и объектов. Например, можно прикрепить к первому кадру клипа функцию *moveClipTo( )* и позднее вызывать *moveClipTo( )* из какой-либо кнопки:

```
function moveClipTo (x, y) {
```

```
    _x = x;
    _y = y;
  }
```

Предостережение: поскольку выполнение кода, находящегося в кадрах, зависит от воспроизведения фильма, необходимо следить, чтобы переменные, функции и другие элементы программ были доступны к моменту обращения к ним. Нельзя, например, вызывать некоторую функцию в кадре 3, если определена эта функция будет только в кадре 10. Поэтому код, используемый в фильме глобально, должен быть помещен в первый кадр главной временной диаграммы. См. описание объекта `_global` в «Справочнике по языку» для получения дополнительной информации о размещении глобального кода. См. также главы 2 и 9, где обсуждаются области видимости переменных и функций.

Необходимо также гарантировать отсутствие в коде обращений к участкам фильма, которые еще не загружены. Чтобы проверить, загружена ли некоторая часть фильма, можно воспользоваться свойством клипа `_framesloaded` или методом *getBytesLoaded( )*. Пример кода можно найти в статье, посвященной *Movieclip.*`_framesloaded` в «Справочнике по языку».

Добавление кода к кнопкам

Во Flash 5 код кнопок прикрепляется непосредственно к объектам кнопок на сцене. Например, если требуется, чтобы щелчок по кнопке перемещал головку воспроизведения в кадр с меткой `section1`, то во Flash 5 в эту кнопку помещается такой код:

```
on (release) {
  gotoAndStop("section1");
}
```

В большинстве случаев вместо прежнего синтаксиса *on(event)* во Flash MX следует применять обработчики событий кнопок на основе функций обратного вызова. Однако в некоторых простейших случаях удобнее поместить код непосредственно в кнопку.

Чтобы поместить код в кнопку, надо выделить ее на сцене, после чего добавить код в окно сценариев панели Actions. Код в кнопке всегда следует помещать в обработчик события, указывающий на обстоятельства, в которых должен быть выполнен код. Например, событием, запускающим большинство действий кнопок, является *release* (*освобождение*). Можно применять и событие *rollOver*, которое вызывает выполнение кода не тогда, когда кнопка нажата, а когда курсор мыши появляется над ней:

```
on (rollOver) {
  this.gotoAndStop("section1");
}
```

Полное описание обработчиков событий кнопок см. в главе 10.

В кнопку можно поместить хоть тысячу строк кода, но делать этого не следует. При всякой возможности обобщайте код кнопки и упаковывайте его в функции, прикрепляемые к временной диаграмме кнопки. Например, *можно* добавить в кнопку такой код:

```
on (release) {
  this.title._xscale = 20;
  this.title._yscale = 20;
  this.title._alpha  = 50;
  this.title.gotoAndPlay("fadeout");
}
```

Но лучше поместить код в функцию и вызывать ее из кнопки:

```
// КОД В КАДРЕ 1 ФИЛЬМА ДОБАВЛЯЕТ МЕТОД fadeOut() ВО ВСЕ КЛИПЫ
MovieClip.prototype.fadeOut = function (scale, transparency) {
  this._xscale = scale;
  this._yscale = scale;
  this._alpha = transparency;
  this. gotoAndPlay("fadeout");
}
// КОД В КНОПКЕ
on (release) {
   this.title.fadeOut(20, 50);
}
```

Во Flash MX еще лучше добавить к экземпляру *Button* обработчик события так:

```
theButton.onRelease = function () {
  this._parent.title.fadeOut(20, 50);
}
```

При таком подходе весь код хранится централизованно, что облегчает дальнейшую работу с ним и позволяет быстро и с минимальными усилиями организовывать такое же поведение для других кнопок.

Код кнопки должен быть прикреплен к объекту кнопки на сцене и содержать обработчик события. Код кнопки без обработчика события приводит к ошибке; нельзя также прикреплять код к внутренним кадрам `Up`, `Over`, `Down` и `Hit` символа кнопки.

Добавление сценариев в клипы

Как мы уже видели, можно прикреплять код к кадрам временной диаграммы клипа. Можно прикрепить код и к самому экземпляру клипа. Для этого надо выделить экземпляр клипа на сцене, после чего добавить код в окно сценариев панели Actions. Как и для кнопок, любой код, прикрепляемый к объектам клипов, должен содержаться в обработчиках событий. Обработчики событий сообщают интерпретатору, когда должен выполняться код клипа. Например, в следующем коде переменной `x` присваивается значение 10 при загрузке клипа:

```
onClipEvent (load) {
  var x = 10;
}
```

Обработчики событий клипов могут реагировать на действия с мышью и клавиатурой, загрузку данных, вывод кадров на экран, рождение и смерть клипов. Полное описание клипов и обработчиков событий можно найти в описании объекта *MovieClip* в «Справочнике по языку».

Во Flash MX появилась модель событий с функциями обратного вызова, необходимость в непосредственном прикреплении кода к экземплярам клипов возникает не часто. Обработчики событий следует применять к экземплярам клипов из центрального хранилища кода на главной временной диаграмме или, если обработчик применяется ко всем экземплярам символа, из самого символа клипа. Обработчики событий можно также задавать в целом для символа клипа с подклассом *MovieClip* (см. главу 14). Некоторые особые исключения можно найти в «Справочнике по языку» в статьях, посвященных обработчикам событий *MovieClip*.

Куда подевался код?

Даже у опытных пользователей Flash поиск кода в фильме иногда вызывает затруднения. Код можно прикрепить к любому кадру любой временной диаграммы, к любой кнопке или любому клипу на сцене, поэтому он часто теряется в море содержимого. Очевидно, развитая структура и тщательное документирование кода могут сберечь десятки, а то и сотни часов в период работы над проектом. Если код в фильме по таинственным причинам невозможно найти, откройте панель Actions и попробуйте разобраться с помощью следующих приемов:

- Чтобы найти код ActionScript в текущей сцене, воспользуйтесь функцией поиска в окне Movie Explorer, описаннной сразу после этого списка.
- Щелкните по любому кадру, на временной диаграмме которого расположен маленький круглый значок. Кружок означает наличие кода ActionScript.
- Найдите на сцене и выделите все белые кружки с черной каемкой. Они обозначают пустые клипы, в которых часто не содержится ничего, кроме кода. Если в пустом клипе кода не видно, дважды щелкните по нему, чтобы начать редактирование, и просмотрите его кадры.
- Выделите поочередно каждую кнопку фильма. Некоторые программисты не склонны размещать код централизованно и располагают важные длинные сценарии прямо в кнопках.
- Поищите на временной диаграмме скрытые или маскированные слои. Слой, рядом с которым находится красный значок X, скрыт для редактирования, но может содержать клипы и кнопки с кодом, появляющиеся во время воспроизведения фильма. Аналогично в маскированных слоях

могут содержаться закрытые объекты с кодом. Снимите защиту с маскированных слоев, чтобы увидеть их содержимое.

- Разблокируйте все слои. Пустые клипы (маленькие кружки с черной каемкой) скрываются, если слой, на котором они находятся, заблокирован.
- С помощью Ctrl+F (Windows) или Command+F (Macintosh) откройте диалоговое окно поиска Find. Оно полезно при поиске заданного текста в больших сценариях. Ниже описывается, как с помощью Movie Explorer выполнить поиск во всех сценариях.

Иногда и эти приемы не помогают. Если некто решил спрятать свой код, то во Flash есть много мест, где это можно сделать. Например, поместить пустой клип далеко за пределами сцены, что делает его обнаружение почти невозможным. К счастью, когда кажется, что все потеряно, можно найти любой сценарий фильма с помощью Movie Explorer. Выберите Window → Movie Explorer, чтобы с высоты птичьего полета рассмотреть все элементы текущей сцены, в том числе сценарии, прикрепленные к кадрам, кнопкам и клипам. Для того чтобы поиск осуществлялся также в сценариях, необходимо, чтобы значок Show Action Scripts в меню Show в верхней части окна Explorer был выделен. Простого средства поиска по всему фильму не существует: для каждой сцены поиск придется выполнять отдельно. Сценарии помечаются значком Actions с голубой стрелкой, похожим на значки в окне инструментов панели Actions. Чтобы выводить только результаты поиска сценариев, выберите значок Show Action Scripts в меню Show и снимите выделение со всех остальных значков.

Производительность

Вот некоторые советы по оптимизации редактирования исходного кода ActionScript:

- Все сценарии временной диаграммы храните в независимом слое под названием *scripts*. Не помещайте в этот слой содержимое: он предназначен только для кода. Храните этот слой вверху окна временной диаграммы, чтобы его код выполнялся после загрузки всех остальных слоев (или в самом низу, если определен порядок загрузки сверху вниз). Если всегда хранить слой *scripts* в одном и том же месте, то код найти легко. (В предыдущем разделе поясняется, насколько это важно!)
- Все метки кадров храните в отдельном слое *labels*. Не помещайте содержимое в этот слой: он предназначен только для меток.
- Обратите внимание на всплывающее меню Options в правом верхнем углу панели Actions. Вы найдете там такие полезные инструменты, как поиск и замена, распечатка исходного текста и управление шрифтами окна сценариев.
- Работая с библиотеками кода, участвующими в нескольких проектах, сохраняйте код во внешних файлах. Подробности можно найти в следующем разделе «Сохранение кода ActionScript во внешних файлах».

- Для ускорения ввода используйте клавиши сокращенного набора, определенные в Actions (например, Esc+G+P для *gotoAndPlay( )*). Эти «горячие» клавиши показываются при нажатии кнопки (+) панели Actions.
- Обратите внимание, что размеры окна инструментов можно изменять. Для экономии места при вводе кода можно частично или полностью скрыть инструменты, передвинув границу между окном инструментов и окном сценариев.
- Краткое описание любого элемента Toolbox можно получить, выбрав его и щелкнув по значку Reference (Книжка) в правом верхнем углу панели Actions.

Сохранение кода ActionScript во внешних файлах

Код ActionScript можно сохранять во внешних текстовых файлах (которым принято давать расширение *.as*) или в файлах *.swf* и импортировать в документ Flash. При размещении кода во внешних файлах облегчается совместное использование библиотек стандартного кода в нескольких проектах. Импортировать внешний код во Flash можно с помощью пункта меню Import From File, директивы `#include` или библиотеки совместного доступа.

Импорт из файла на этапе разработки

Во время редактирования файла *.fla* можно импортировать код в окно сценариев через пункт меню Import From File, который открывается при нажатии на кнопку всплывающего меню Options, находящуюся в правом верхнем углу панели Actions (см. рис. 16.1). Import From File – одноразовая операция, которая копирует содержимое внешнего файла в панель Actions, *заменяя* находящийся там сценарий. Импортированный через Import From File код никоим образом не становится постоянно связанным с файлом *.fla*. Чтобы добавить или вставить текст в сценарий, а не заменять его, необходимо вручную выполнить копирование и вставку во внешнем текстовом редакторе.

#include (импорт на этапе компиляции)

При экспорте (*компиляции*) файла *.swf* из файла *.fla* можно включить код из внешнего текстового файла с помощью директивы `#include`. О применении `#include` см. «Справочник по языку».

Библиотеки совместного доступа (импорт на этапе выполнения)

Для импорта кода из внешнего источника во время фактического воспроизведения фильма необходимо создать файл библиотеки совместного доступа *.swf*, в котором находится клип, содержащий импортируемый код. Импорт на этапе выполнения обеспечивает самый гибкий подход к совместному ис-

пользованию кода в разных фильмах, потому что не требует перекомпиляции фильмов, импортирующих совместно используемый код: когда обновляется библиотечный файл *.swf*, это обновление автоматически отражается в связанных с ним фильмах.

Следующая процедура описывает, как сделать простую проверочную функцию из фильма *codeLibrary.swf* доступной в фильме *myMovie.swf*. Сначала создадим фильм *codeLibrary.swf*:

1. Создать новый документ Flash.
2. Создать новый символ клипа с именем *sharedFunctions*.
3. В кадр 1 клипа *sharedFunctions* поместить следующий код:

```
function test () {
  trace("The shared function, test, was called.");
}
```

4. Выбрать в библиотеке клип *sharedFunctions*.
5. В панели библиотеки выбрать Linkage во всплывающем меню Options.
6. Выбрать Export for Runtime Sharing.
7. В окне Identifier ввести `sharedFunctions`.
8. В окне URL ввести `codeLibrary.swf`.
9. Нажать кнопку OK.
10. Сохранить документ как *codeLibrary.fla*.
11. С помощью File → Export Movie создать *codeLibrary.swf* из *codeLibrary.fla*.
12. Закрыть файлы *codeLibrary.swf* и *codeLibrary.fla*.

Теперь создадим файл *myMovie.swf*, который будет выполнять код, импортированный из *codeLibrary.swf*:

1. Создать новый документ Flash.
2. Сохранить новый документ как *myMovie.fla* в одной папке с *codeLibrary.fla*.
3. Переименовать *Layer 1* в *sharedCode*.
4. Выполнить File → Open As Library и выбрать *codeLibrary.fla*. Появится библиотека *codeLibrary. fla*.
5. Перетащить экземпляр клипа *sharedFunctions* из библиотеки *codeLibrary.fla* на сцену в кадр 1 *myMovie.fla*.
6. Дать экземпляру имя *sharedFunctions*.
7. На главной временной диаграмме *myMovie.fla* создать новый слой с именем *scripts*.
8. Добавить ключевой кадр в кадр 2 слоя *scripts*.
9. Добавить новый кадр в слой *sharedCode*.
10. К кадру 2 слоя *scripts* прикрепить следующий код:

```
stop();
sharedFunctions.test();
```

11. Экспортировать *myMovie.swf*.
12. В окне Output появится текст: «The shared function, test, was called».

Обратите внимание, что библиотека совместного доступа представляет собой присоединенный ресурс, вроде графического файла *.gif*, присоединенного к файлу *.html*. Поэтому библиотечные файлы *.swf* должны всегда быть правильно размещены на сервере – вместе с файлами, которые их используют. Чтобы изменить значение ссылки от импортированного символа к библиотеке совместного доступа, нужно выполнить следующие действия:

1. Выбрать символ в библиотеке.
2. Во всплывающем меню Options панели библиотеки выбрать Linkage.
3. Указать в URL новый адрес.

Создание компонентов

Компонент – это клип, позволяющий присваивать значения некоторым своим свойствам с помощью особого графического интерфейса пользователя в среде разработки Flash. Компоненты предоставляют возможность пользователям-непрограммистам настраивать программно управляемые клипы. В компонентах переменные, определяющие поведение, отделены от кода клипа, что позволяет рассматривать их как «черные ящики»: их устройство неизвестно, но известны данные на входе и на выходе, а также поведение.

Компоненты Flash MX представляют собой новую и улучшенную версию Smart Clips из Flash 5.

В главе 14 мы рассмотрели, как определить компонент для подкласса *MovieClip*. Тогда нашей целью было создание простого в работе GUI для построения экземпляров подклассов. Теперь мы рассмотрим непосредственно процедуру определения параметров компонента для любых клипов, а не только подклассов *MovieClip*.

Обычно инициализация переменных происходит в исходном коде клипа. Вот, например, настройка переменных, управляющих фейерверком:

```
// Переменные, определяемые пользователем
this.numSparks = 10;            // Число клипов вспышек во взрыве
this.randomDispersion = true;   // Тип взрыва (true для случайного,
                                // false для единообразного)
this.duration = 1300;           // Длительность взрыва в миллисекундах
```

Модификация такого типа кода может испугать непрограммиста. Но если встроить этот код в компонент, непрограммист сможет настроить действие фейерверка через инспектор свойств с помощью знакомого ему интерфейса. На рис. 16.2 показан интерфейс компонента, эквивалентный нашему коду инициализации переменных.

В интерфейсе компонента имена и значения всех переменных выводятся четко разделенными по строкам и колонкам. Имена переменных редактиро-

Рис. 16.2. *Пример интерфейса настройки компонента*

вать нельзя, чтобы случайная опечатка не привела к краху системы. У каждой переменной есть также более пространное имя, из которого становится яснее, что она делает. Наконец, переменным с ограниченным множеством допустимых значений (таким как `randomDispersion`) можно присваивать значение с помощью выпадающего меню.

Интерфейс, показанный на рис. 16.2, конечно, более доступен, чем исходный код. Но компоненты можно делать еще более дружественными к пользователю. Интерфейс компонента, устанавливаемый по умолчанию, можно заменять собственным, например таким, как на рис. 16.3. Заметьте, что в этом интерфейсе имена переменных вообще скрыты, и непрограммист настраивает каждый экземпляр фейерверка с помощью текстовых полей и выпадающих меню. Интерфейс позволяет даже выполнить предварительный просмотр настроек в действии!

Посмотрим, как это работает.

Рис. 16.3. *Специализированный интерфейс настройки компонента*

Создание компонента со стандартным интерфейсом

Как только что было отмечено, компоненты могут обладать стандартным системным интерфейсом, установленным по умолчанию, или специализированным интерфейсом. Сначала научимся строить интерфейс стандартного вида.

При построении любого компонента сначала надо создать обычный клип, управляемый значениями одной или нескольких переменных. Например,

в следующем коде переменные xPos и yPos определяют положение клипа на сцене:

```
this._x = this.xPos;
this._y = this.yPos;
```

При создании клипа в виде компонента для собственных нужд или для кого-то другого предполагается, что некоторые переменные будут установлены с помощью инспектора свойств при помещении клипа на сцену. Эти переменные называют *параметрами компонента*. Когда клип, поведение которого определяется одним или более параметрами компонента, создан, надо придать ему интерфейс компонента, посредством которого можно устанавливать эти параметры.

Добавление стандартного интерфейса в компонент

Чтобы добавить в клип интерфейс компонента по умолчанию, надо выполнить следующие действия:

1. Выбрать клип в библиотеке.
2. Выбрать Component Definition во всплывающем меню Options окна библиотеки. (Появится окно диалога Component Definition.)
3. Над окном Parameters щелкнуть по кнопке «плюс» (+), чтобы добавить параметр компонента.
4. Повторить шаг 3 для каждого параметра компонента.
5. Настроить параметры компонента, как описано в следующем разделе.

Настройка стандартных параметров компонента

Добавив параметр в компонент, надо присвоить ему экранное имя, имя переменной и необязательное значение по умолчанию. Как и переменные, параметры компонентов могут содержать данные различных типов. Однако типы параметров компонентов не совсем совпадают с типами, которые поддерживаются переменными. Параметрами компонента могут быть *массивы*, *объекты*, *списки*, *строки*, *числа*, *булевы величины*, *названия шрифта* и *цвет*. От типов данных, поддерживаемых переменными, они отличаются следующим:

- Параметры компонентов поддерживают три типа значений, которые используются только в интерфейсах: *список*, *название шрифта* и *цвет*. *Список* предназначен для ограничения значений, которые могут быть присвоены параметру, некоторым заранее определенным множеством. Например, для параметра difficulty (сложность) может быть установлен список значений: "высокая", "обычная", "низкая". *Имя шрифта* позволяет выбрать установленный на компьютере шрифт с помощью особого меню. Выбранный шрифт может быть использован объектами *TextFormat*. *Цвет* дает возможность выбрать целочисленное RGB-значение.
- Элементарные типы null и undefined нельзя задавать непосредственно в качестве значений параметров компонентов. Это ограничение накладывается интерфейсом компонентов, а не самим параметром: код, нахо-

дящийся внутри компонентов, может присваивать значения `null` или `undefined` переменным, инициализированным как параметры.

Чтобы дать параметру компонента экранное имя, имя переменной и необязательное значение по умолчанию, надо сделать следующее:

1. Щелкнуть по имени (Name) параметра и ввести короткую строку, описывающую параметр.
2. Щелкнуть по переменной (Variable) параметра и ввести фактическое имя переменной параметра.
3. Щелкнуть по типу (Type) параметра и выбрать один из следующих:
   - Default для параметров, принимающих строковые или числовые значения.
   - Array для параметров, принимающих значение массива.
   - Object для параметров, имеющих значением объект, в том числе точки, содержащие координаты x и y.
   - List для параметров с предустановленным набором возможных строковых или числовых значений.
   - String для строковых значений.
   - Number для числовых (целых или с плавающей точкой) значений.
   - Boolean для значений `true` или `false`.
   - Font Name для создания меню выбора шрифта.
   - Color для создания меню выбора цвета.
4. Щелкнуть по значению параметра и ввести значение по умолчанию, если оно требуется. Это значение будет выведено в интерфейсе компонента в качестве начального. Способ ввода значений по умолчанию зависит от типа параметра:
   - Для параметров Default, Sring и Number щелкнуть по значению параметра и ввести строку или число.
   - Для параметров Array, Object и List щелкнуть по значению параметра. В диалоговом окне Values можно добавлять, удалять и переупорядочивать элементы с помощью кнопок «плюс», «минус» и стрелок. Закончив настройку, щелкнуть по кнопке OK.
   - Для параметров Boolean, Font Name и Color щелкнуть по значению параметра. В появившемся меню выбрать булево значение, имя шрифта или цвет.
5. Чтобы запретить пользователю компонента изменять имя параметра, установить флажок Parameters are Locked in Instances.
6. В диалоговом окне Component Definition щелкнуть по кнопке OK для завершения настройки параметров.

Удаление и изменение порядка стандартных параметров компонента

Иногда требуется удалить параметры компонента или изменить их порядок.

Для удаления параметра компонента надо выполнить следующие действия:

1. Выбрать в библиотеке компонент.
2. Выбрать Component Definition во всплывающем меню Options библиотеки (появится окно диалога Component Definition).
3. Выбрать удаляемый параметр.
4. Щелкнуть по кнопке «минус» (–).
5. Щелкнуть по кнопке OK.

Для изменения порядка параметров компонента необходимо сделать следующее:

1. Выбрать в библиотеке компонент.
2. Выбрать Component Definition во всплывающем меню Options библиотеки.
3. Выбрать перемещаемый параметр.
4. Щелкать по кнопкам со стрелками, пока параметр не окажется в нужном месте.
5. Щелкнуть по кнопке OK.

Создание компонентов с настраиваемым интерфейсом

Чтобы построить компонент с настраиваемым интерфейсом, сначала надо создать обычный клип, поведение которого управляется рядом параметров компонента, как это было описано в предыдущем разделе. После этого создать независимый файл *.swf* (так называемый *интерфейсный .swf*), который будет использоваться для задания параметров компонента через текстовые поля, меню, кнопки и другие элементы GUI. Эти значения автоматически считываются и пересылаются в компонент как параметры.

Компонент обменивается данными с интерфейсом *.swf* через экземпляр `xch` (от *exchange*) – экземпляр со специальным названием в интерфейсном *.swf*. (О том, как создавать экземпляр `xch`, рассказывается ниже.) На рис. 16.4 показано, как имена и значения параметров пересылаются из интерфейсного *.swf* в компонент.

Обмен данными между интерфейсным *.swf* и компонентом происходит в цикле. Когда экземпляр компонента выбирается на сцене, соответствующий интерфейсный *.swf* загружается в панель инспектора свойств или окно Component Parameters. После этого текущие параметры экземпляра компонента передаются экземпляру `xch` файла *.swf*. Файл *.swf* должен извлечь эти параметры и настроить соответствующим образом состояние интерфейса. Переменные, которые будут потом установлены в `xch` файлом *.swf*, автоматически передаются компоненту в качестве параметров. Когда с экземпляра компонента снимается выделение, интерфейсный *.swf* удаляется из панели Compo-

Рис. 16.4. *Обмен данными в компоненте*

nent Parameters. Однако значения параметров не теряются, они сохраняются экземпляром компонента. При каждом выделении экземпляра компонента его параметры передаются клипу `xch` файла *.swf*. Этот цикл позволяет сохранять синхронизацию интерфейсного файла *.swf* с панелью Component Parameters.

В последующих разделах описывается, как создать пользовательский интерфейсный *.swf* и связать его с компонентом. Для более глубокого изучения можно получить из сетевого хранилища кода пример компонента с особым интерфейсом пользователя под названием «Playhead Control».

Создание интерфейсного файла .swf

Чтобы создать файл *.swf*, используемый в качестве особого интерфейса компонента, необходимо сделать следующее:

1. Создать новый документ Flash.
2. Создать новый слой с именем *xchLayer*.
3. Выбрать Insert → New Symbol и создать пустой символ клипа.
4. Дать новому символу имя *xchClip*.
5. В слое *xchLayer* расположить экземпляр символа *xchClip*.
6. Дать экземпляру имя `xch`.
7. В кадре 1 главной временной диаграммы разместить кнопки, текстовые поля и прочие элементы интерфейса, необходимые для установки значений переменных (задача облегчается использованием встроенных Flash UI Components).
8. В кадре 2 главной временной диаграммы разместить код для чтения параметров компонента из экземпляра `xch`. Настроить пользовательский интерфейс так, чтобы он отражал значения параметров компонента.

Например, следующий код инициализирует текстовое поле `param1Input_txt` значением `param1` из компонента:

```
param1Input_txt.text = xch.param1;
```

9. Остановить головку воспроизведения в кадре 3 главной временной диаграммы. Добавить код интерфейса, который устанавливает параметры компонента как переменные в экземпляре `xch`. Переменные в `xch` будут автоматически добавлены в компонент в качестве параметров. Например, вот так можно задать значения для двух таких параметров:

```
xch.param1 = value1;  // Значения могут иметь любой тип данных
xch.param2 = value2;
```

10. Экспортировать файл *.swf*.

Добавление собственного интерфейса в компонент

После создания интерфейсного *.swf* надо добавить его к компоненту, для чего требуется:

1. Закрыть интерфейсный *.swf* и вернуться к исходному файлу *.fla*, содержащему компонент.
2. Выбрать компонент в библиотеке.
3. Выбрать Component Definition во всплывающем меню Options библиотеки (появится окно диалога Component Definition).
4. Рядом с Custom UI щелкнуть по кнопке Set.
5. В окне Custom UI указать для Type одно из следующего:
   - Чтобы встроить интерфейсный файл *.swf* прямо в файл компонента *.fla*, выберите «Custom UI with *.swf* file embedded in *.fla* file». Встроенные интерфейсные файлы *.swf* хранятся внутри файлов компонентов *.fla*, что позволяет использовать и перемещать файл компонента *.fla* независимо, без оглядки на файл интерфейса *.swf*. При модификации встроенного файла интерфейса *.swf* его следует обновить вручную с помощью кнопки Update в окне Custom UI.
   - Чтобы установить внешнюю связь с файлом интерфейса *.swf*, выберите «Custom UI in external *.swf* file». Внешние присоединенные файлы интерфейса *.swf* всегда должны быть доступны по адресу, заданному в «Custom UI *.swf* file». Они обновляются автоматически каждый раз, когда выделяется экземпляр компонента.
6. Для Display выберите одно из следующего:
   - Если интерфейсный файл *.swf* не велик и помещается в инспекторе свойств, выберите «Display in Property Inspector».
   - Если интерфейсный файл *.swf* слишком велик, чтобы поместиться в инспекторе свойств, выберите «Display in Component Parameters Panel».
7. Для файла *.swf* Custom UI укажите адрес файла *.swf*, выступающего в качестве пользовательского интерфейса, относительно текущего файла *.fla*. (Файл *.swf* можно также выбрать с помощью кнопки Browse.)

Средства для создания профессиональных компонентов

Для опытных программистов и сторонних разработчиков, распространяющих библиотеки кода в виде компонентов, Flash MX предоставляет ряд возможностей, предназначенных для придания компонентам профессионального блеска. Эти возможности, обзор которых приводится ниже, подробно описаны в статье «Creating Components in Macromedia Flash MX», доступной по адресу:

http://www.macromedia.com/support/flash/applications/creating_comps/

Эти возможности также описаны в обновленной документации Macromedia по Flash MX, которую можно найти здесь:

http://www.macromedia.com/support/flash/ts/documents/doc_update_mx.htm

Описание компонента

Описание компонента можно добавить в окно справки Flash MX Reference и в панель инструментов окна Actions с помощью обычного текстового файла или файла XML.

Пользовательский значок компонента

Значок компонента, устанавливаемый по умолчанию, можно заменить готовым значком из окна Component Definition или пользовательским значком в формате BMP или PNG из библиотеки файла *.fla*. Значок располагается рядом с компонентом в окнах Library, Components и Movie Explorer.

Всплывающая подсказка в окне Components

К компонентам, отображаемым в окне Components, можно добавить всплывающую подсказку. Всплывающая подсказка состоит из строки описания компонента, отображаемой при наведении на него курсора мыши.

Предварительный просмотр компонента

Можно устроить предварительный просмотр компонента на сцене (так называемый *live preview*), благодаря которому пользователь сможет получить общее представление о внешнем виде компонента на этапе разработки. Это реализуется в виде файла *.swf*, весьма схожего с Custom UI. Во Flash MX предварительный просмотр может обновляться раз в секунду.

Установка в панели Components

Компонент можно установить в среде разработки Flash MX, так чтобы он был доступен любому новому документу через панель Components. Установка осуществляется путем перетаскивания файла *.fla* компонента вручную в папку среды разработки *Flash MX\Configuration\Components* или с помощью менеджера расширений Macromedia, доступного по адресу *http://www.macromedia.com/exchange/flash*.

Использование компонентов

Клип, которому присвоены параметры компонента, официально становится «компонентом». По умолчанию компоненты помечаются специальным значком: .

Компоненты отображаются в окне Library или панели Components (если это задано).

Для того чтобы использовать экземпляр компонента в фильме, надо выполнить следующие действия:

1. Перетащить компонент из библиотеки или панели Components на сцену.
2. Чтобы задать параметры компонента, открыть инспектор свойств (Window → Properties) или панель Component Parameters (Window → Component Parameters), которая используется только для компонентов с пользовательскими интерфейсами, не умещающимися в окне инспектора свойств.
3. Если у компонента стандартный интерфейс, установить значение каждого параметра следующим образом:
   - Для параметров Default, String и Number щелкнуть по значению параметра и ввести строку или число.
   - Для булевых параметров щелкнуть по значению параметра и ввести `true` или `false`.
   - Для параметров Array дважды щелкнуть по значению параметра. В диалоговом окне Values щелкнуть по каждому значению элемента массива и ввести строку или число. Щелкнуть по кнопкеи OK, чтобы принять значения элементов массива.
   - Для параметров Object дважды щелкнуть по значению параметра. В диалоговом окне Values щелкнуть по каждому значению свойства объекта и ввести строку или число. Щелкнуть по кнопке OK, чтобы сохранить значения свойств.
   - Для параметров List щелкнуть по значению параметра, после чего выбрать вариант значения.
   - Для параметров Color щелкнуть по значению параметра и выбрать цвет в палитре либо просто ввести целое число (в шестнадцатеричном или десятичном виде), представляющее RGB-значение цвета.
   - Для параметров Font Name щелкнуть по значению параметра и выбрать шрифт в выпадающем меню.
4. Если у компонента особый интерфейс, используйте для настройки его параметров те средства, которые он предоставляет.

Вперед!

Наконец-то мы завершили обзор основ ActionScript. Пора привести пример приложения, в котором все полученные знания найдут применение. В следующей главе мы соединим ActionScript с простым серверным приложением для создания Flash-формы.

17

Создание формы Flash

Подобно HTML, Flash поддерживает заполняемые формы, которые могут передавать введенные пользователем данные приложению или сценарию на стороне сервера. В этой главе мы узнаем, как получить данные, введенные пользователем, и организовать связь со сценарием Perl, выполняемым на сервере, с помощью класса Flash MX *LoadVars*. Благодаря этой технологии можно сохранить наилучший результат, достигнутый в видеоигре, или получить с сервера бегущую строку с котировками.

Цикл данных форм Flash

Прежде чем погружаться в частности, рассмотрим в общих чертах типичные этапы процесса передачи формы:

1. Flash получает данные, введенные пользователем.
2. Flash готовит данные для отправки веб-серверу (получает значения переменных и проверяет их допустимость).
3. Flash отправляет данные веб-серверу по HTTP (или HTTPS).
4. Веб-сервер получает данные и передает их серверному приложению (например, сценарию Perl, сценарию PHP, Cold Fusion или ASP) для обработки.
5. Обрабатывающее приложение производит анализ и обработку переданной информации.
6. Обрабатывающее приложение передает результаты веб-серверу, а тот пересылает результаты обратно Flash.
7. Flash запоминает результаты и при необходимости выводит их.

Поэтому для действующей формы Flash необходимы:

- Клиентская часть (то, что видит пользователь).
- Некоторый сценарий Flash, передающий содержимое формы сценарию или приложению, выполняемым на сервере.
- Сценарий или приложение, выполняемые на сервере.

- Некоторый сценарий Flash, обрабатывающий данные, полученные от сервера.

Изучим эти составляющие подробнее.

Ввод данных во Flash на стороне клиента

Для того чтобы осуществить передачу своих данных, пользователи вводят текст в текстовые поля. В ActionScript находящееся на экране текстовое поле соответствует экземпляру класса *TextField*, с помощью которого осуществляется доступ к данным в поле. Пользователи могут также вводить данные с помощью таких элементов интерфейса, как выпадающие меню, переключатели и флажки. В версиях, предшествовавших Flash MX, эти типовые элементы форм разработчик должен был создавать с нуля. Во Flash MX появились готовые графические элементы форм, называемые *компонентами пользовательского интерфейса Flash* (*Flash UI Components*): CheckBox, ComboBox, ListBox, PushButton, RadioButton, ScrollBar и ScrollPane. Эти компоненты не являются частью ActionScript, это сложные клипы, поставляемые вместе со средой разработки Flash MX. Чтобы узнать, как применять компоненты пользовательского интерфейса Flash MX, выберите Help → Tutorials → Introduction to Components Tutorial. Свойства и методы Flash UI Components описаны в приложении G.

Прежде чем отправлять веб-серверу данные, введенные пользователем, их надо проверить, чтобы обрабатывающее серверное приложение получило пригодные для работы данные. При проверке данных обычно выясняется, все ли необходимые поля заполнены и правилен ли тип введенных в них данных. Например, адрес электронной почты должен содержать имя, за которым следуют символ @ и имя домена.

Передача данных серверу для обработки

Если правильность данных проверена, можно безбоязненно передать их веб-серверу. ActionScript предоставляет несколько средств для пересылки данных формы на веб-сервер:

- Класс *LoadVars*, описываемый ниже в этой главе.
- Функция *loadVariables( )* (применявшаяся во Flash 4 и Flash 5, но уступившая место *LoadVars* во Flash MX).
- Методы *send( )* и *sendAndLoad( )* класса *XML*, описываемые в «Справочнике по языку».
- Глобальная функция getURL*( )*, также рассмотренная в «Справочнике по языку».

Веб-сервер пересылает данные Flash выполняемому на стороне сервера приложению (например, Macromedia ColdFusion или Microsoft ASP) или сценарию CGI (например, сценарию Perl или PHP либо сервлету Java).

Описывая оборот данных между клиентом и веб-сервером, мы обращаем особое внимание на различие между веб-сервером и приложением, обрабатываю-

щим данные. Часто это различие скрыто: клиент всегда должен послать запрос HTTP, чтобы отправить данные приложению, которое их обрабатывает, поэтому естественно, что в процессе участвует веб-сервер. Однако при разработке форм Flash необходимо помнить о невидимых взаимодействиях между веб-сервером и приложением, обрабатывающим данные. Данные перемещаются из Flash на сервер либо дописанными в конец URL (если применяется метод GET), либо в потоке имен переменных и их значений (если применяется метод POST). Если на веб-сервере возникает ошибка, то Flash не выводит сообщений HTTP об ошибках, которые посылает сервер (что обычно делает броузер). Например, если веб-сервер не может найти сценарий CGI, он посылает сообщение «404 Not Found», но Flash его не выводит. Аналогично, если для сценария CGI неправильно установлены права доступа, Flash не выводит сообщение об ошибке выполнения. Чтобы выяснить, какие проблемы клиента/сервера возникли во время работы с Flash, полезно посмотреть, что записал веб-сервер в журнал ошибок HTTP при попытках выполнения сценариев. Журнал веб-сервера может рассказать о том, чего не сообщает Flash. О неполадках, связанных с методом HTTP POST, рассказано в приложении F.

Приложение, обрабатывающее данные

Получив часть данных, обрабатывающее приложение должно произвести их *разбор* (*to parse*), то есть интерпретировать разумным образом и разбить на более мелкие части. Проанализировав данные, серверное приложение может распорядиться ими массой способов, например сохранить в базе данных или в простом текстовом файле для извлечения в будущем.

Когда обработка данных завершена, приложение, обрабатывающее данные, выводит результат, который должен быть передан обратно Flash. Этот результат может быть простым подтверждающим сообщением («Спасибо за переданные вами сведения»), списком записей из базы данных или ценой товара в данный момент.

Приложение передает результат своей работы веб-серверу, который пересылает его Flash для сохранения или вывода.

Разработчикам приложений для обработки данных следует учитывать, что их приложение должно устанавливать MIME-тип результата как «application/x-www-form-urlencoded». При отсутствующем или неправильно указанном типе MIME Flash не сможет правильно интерпретировать результат.

Передача международных символов во Flash

Если в ответе сервера употребляются символы с кодами Unicode, большими 127 (то есть из верхней части таблицы ASCII или символы, не входящие в Latin 1), для передачи данных следует использовать XML в кодировке UTF-8. Для использования потока переменных классом *LoadVars* можно также кодировать его в UTF-8, но не все приложения на стороне сервера могут задать эту кодировку. В качестве последней меры, если точно известно, что *кодовая*

страница сервера (то есть его основной набор символов) совпадает с кодовой страницей системы клиента Flash, можно с помощью `System.useCodepage` заставить Flash Player 6 интерпретировать символы в соответствии с текущей кодовой страницей, а не Unicode (во Flash 5 это фактически единственный поддерживаемый режим). Дополнительную информацию см. в разделе «Кодировка символов» главы 4 и описании `System.useCodepage` в «Справочнике по языку».

Flash получает и интерпретирует результаты

Мы приблизились к концу цикла работы формы Flash, но еще не достигли его. Представим себе фильм, который ищет цены акций. Пользователь вводит биржевой символ акции и нажимает кнопку Get Stock Price. Чтобы можно было вывести цену, приложение, работающее с биржевыми данными, должно найти цену акции и возвратить ее. Пока фильм находится в ожидании, нужно показывать сообщение «Loading» (идет загрузка). Когда будет получена цена (то есть завершена загрузка данных), Flash запустит обработчик *onLoad( )* объекта *LoadVars*, в котором можно интерпретировать данные и показать их пользователю. (Данные, полученные через *LoadVars.sendAndLoad( )*, сохраняются в объекте *LoadVars*.)

Когда цикл работы формы Flash завершен, мы получаем возможность как угодно обработать проделавшие долгий путь драгоценные байты содержимого. Теперь применим полученные знания на практике, создав простую заполняемую форму.

Создание заполняемой формы Flash

Этот пример демонстрирует отправку одной переменной текстового поля из Flash в сценарий Perl с названием *echo.pl*, а также получение ответа от *echo.pl* во Flash. Фильм, который мы создадим, будет состоять из интерфейса и двух функций – *initForm( )* и *submitForm( )*. Интерфейс состоит из текстового поля ввода, кнопки Submit и клипа, показывающего ответ *echo.pl*. Функция *initForm( )* создает объекты для связи с сервером, которые пересылают нашу переменную между Flash и *echo.pl*. Функция *submitForm( )* форматирует данные и отправляет их *echo.pl*. Для упрощения код прикреплен к кадру 1 слоя *scripts*. Фактически надо прикреплять код к кадру, следующему за вашим предзагрузчиком.

Файлы примера можно загрузить из сетевого хранилища кода.

Создание интерфейса

В HTML интерфейс формы создается с помощью тегов `<SELECT>`, `<OPTION>` и `<INPUT>`. Во Flash интерфейс формы можно сделать вручную в среде разработки (как в данном примере) или на этапе выполнения с помощью ActionScript. Наши текстовое поле и кнопка эквивалентны тегам HTML `<INPUT TYPE="TEXT">`, `<INPUT TYPE="TEXTAREA">` и `<INPUT TYPE="SUBMIT">`.

Flash MX предлагает сложные графические элементы для форм в панели компонентов (Window → Components). О создании экземпляров компонентов вручную и с помощью ActionScript см. Help → Tutorials → Introduction to Components Tutorial. Создание собственных компонентов рассмотрено в главах 14 и 16.

Начните построение интерфейса с создания трех слоев: *scripts*, *input form* и *output*.

Затем поместите в слой *input form* текстовое поле для ввода пользователем данных:

1. В панели инструментов выберите Text.
2. В слое *input form* начертите прямоугольник, размеры которого достаточны для ввода пользователем одной строки текста.
3. Откройте инспектор свойств (Window → Properties).
4. В инспекторе свойств выберите для Text Type значение Input Text.
5. В инспекторе свойств выберите для Line Display значение Single Line.
6. В инспекторе свойств введите в поле Instance Name `input_txt`. (Суффикс `_txt` активизирует всплывающую подсказку в панели Actions.)
7. В инспекторе свойств выберите кнопку Show Border Around Text.

Пользователи Flash MX могут создать текстовое поле `input_txt` на этапе выполнения с помощью метода *MovieClip.createTextField( )*:

```
// Создать текстовое поле и расположить его на временной диаграмме.
// Параметры поля:
// имя экземпляра, глубина, x-координата, y-координата, ширина, высота
_root.createTextField("input_txt", 1, 250, 235, 150, 20);

// Определить текстовое поле как поле для ввода
input_txt.type = "input";

// Задать отображение в виде одной строки
input_txt.multiline = false;

// Включить рамку и фон
input_txt.border = true;
input_txt.background = true;

// Необязательная установка цвета рамки и фона
input_txt.borderColor = 0x666666;
input_txt.backgroundColor = 0x999999;
```

Затем добавьте в слой *input form* кнопку Submit:

1. Выберите Window → Common Libraries → Buttons.
2. Из папки Push Buttons библиотеки *Buttons.fla* перетащите на сцену экземпляр кнопки *push button - blue*.

Для того чтобы кнопка Submit вызывала пользовательскую функцию *submitForm( )*, которую мы напишем позже, выполните следующее:

1. Выберите экземпляр кнопки *push button - blue*.
2. Выберите Window → Actions.
3. Введите в панели Actions следующий код:

```
on (release, keyPress "<Enter>") {
  submitForm();
}
```

Flash MX позволяет достичь того же результата путем назначения *submitForm( )* в качестве функции обратного вызова для обработчика *onRelease( )*. Для этого присвойте кнопке в инспекторе свойств имя экземпляра, например `submit_btn`. Затем прикрепите к главной временной диаграмме следующий код (лучше всего в *initForm( )*):

```
submit_btn.onRelease = submitForm;
```

Как правило, этот синтаксис оказывается предпочтительнее, поскольку он отделяет код от физической кнопки на сцене. Однако нам нужно, чтобы форма реагировала на клавишу Enter, поэтому мы используем обработчик кнопки в стиле Flash 5 *on(keyPress "<ENTER>")*, который позволяет легко перехватывать нажатие клавиши Enter. В более сложных случаях (например, при работе с несколькими кнопками Submit) приходится прибегать к более сложным обработчикам нажатий клавиш. Подробности см. в описании *Key.addListener( )*, *Key.onKeyDown( )* и *Key.onKeyUp( )* в «Справочнике по языку».

Завершает создание клиентской формы построение клипа с именем `response`, который выводит результаты, получаемые от *echo.pl*. У клипа `response` три состояния, представляемые тремя ключевыми кадрами с метками `idle`, `loading` и `doneLoading`:

- При загрузке фильма `response` показывает кадр `idle` (обычно невидимый пользователю), в котором `response` ждет начала загрузки данных.
- При передаче переменных `response` показывает кадр `loading`, указывающий, что данные переданы на сервер и Flash ждет ответа.
- При получении переменных `response` показывает кадр `doneLoading` с текстовым полем `output_txt`, в котором пользователю выводятся результаты.

Чтобы построить `response`, сначала создайте новый символ:

1. Выберите Insert → New Symbol, в результате чего откроется диалоговое окно Create New Symbol.
2. В поле Name введите `responseClip`.
3. Для Behavior выберите значение Movie Clip.
4. Щелкните по кнопке OK.

Теперь создайте экземпляр *responseClip*:

1. Перетащите из библиотеки в слой сцены *output* экземпляр *responseClip*.
2. Дайте экземпляру *responseClip* имя `response`.

Наконец, создайте в символе *responseClip* помеченные кадры «состояний»:

1. Выполните двойной щелчок по символу *responseClip* в библиотеке (для редактирования клипа).
2. Создайте четыре слоя временной диаграммы с именами (сверху вниз) *scripts*, *labels*, *loading* и *outputField*.
3. Создайте в каждом слое три ключевых кадра в кадрах 1, 2 и 3.
4. В слое *labels* дайте ключевым кадрам 1, 2 и 3 имена `idle`, `loading` и `doneLoading`.
5. В слое *scripts* добавьте в кадр 1 код `stop();`.
6. В слое *loading* добавьте в кадр 2 статический текст `"loading, please wait"`.
7. В слое *outputField* нарисуйте в кадре 3 текстовое окно с помощью инструмента Text.
8. В инспекторе свойств выберите для Text Type значение Dynamic Text.
9. В инспекторе свойств выберите для Line Display значение Multiline.
10. В инспекторе свойств введите для Instance Name значение `output_txt`.

Отправка данных серверу

Чтобы отправить значение `input_txt` в *echo.pl*, создадим пользовательскую функцию *initForm( )*, показанную в примере 17.1, и пользовательскую функцию *submitForm( )*, показанную в примере 17.2. Функция *initForm( )* создает один экземпляр класса *LoadVars* для отправки данных (`dataSender`), а другой – для получения ответа *echo.pl* (`dataReceiver`). Кроме того, функция *initForm( )* присваивает функцию обратного вызова обработчику `dataReceiver` *onLoad( )*, который выполняется при получении ответа *echo.pl*. Функция обратного вызова показывает ответ в клипе `response`. Обратите внимание, что *initForm( )* создает глобальную переменную для ссылки на временную диаграмму формы, поэтому все наши ссылки на клип `response` оказываются полностью квалифицированными.

Пример 17.1. Функция initForm( )

```
/*
 *  Функция: initForm()
 *  Описание: Создает объекты LoadVars для отправки данных на сервер
 *            и получения данных от сервера.
 *  Параметры: Нет.
 */
function initForm () {
  // Создать ссылку на временную диаграмму формы.
  _global.form_mc = this;

  // Создать объекты LoadVars.
  // Один для отправки...
  dataSender = new LoadVars();
  // ...и второй для приема.
  dataReceiver = new LoadVars();

  // Присвоить функцию обратного вызова обработчику onLoad объекта dataReceiver.
```

```
    // Она будет обрабатывать полученные переменные.
    dataReceiver.onLoad = function () {
      // Подготовить клип response к отображению данных.
      form_mc.response.gotoAndStop("doneLoading");

      // Показать значение загруженной переменной output в поле output_txt.
      form_mc.response.output_txt.text = this.output;
    }
  }
```

Функция *submitForm( )*, показанная в примере 17.2, собирает данные и передает их сценарию *echo.pl*.

Пример 17.2. Функция submitForm( )

```
/*
 * Функция: submitForm()
 * Описание: Извлекает данные, введенные пользователем, и посылает их на сервер.
 * Параметры: Нет.
 */
function submitForm () {
  // Присвоить значение input_txt свойству dataSender. Функция escape()
  // кодирует символ &, чтобы избежать ошибок при передаче в сети.
  dataSender.formInput = escape(input_txt.text);

  // Показать в клипе response сообщение об идущей загрузке.
  form_mc.response.gotoAndStop("loading");

  // Для отладки показать в окне Output данные, которые мы посылаем.
  trace("Sending " + dataSender.toString());

  // Отправить свойства dataSender серверу и ждать ответа.
  dataSender.sendAndLoad("http://www.yourserver.com/cgi-bin/echo.pl",
                         dataReceiver, "GET");
}
```

Свойство `formInput` объекта `dataSender` содержит данные, которые нужно передать – текущее содержимое текстового поля `input_txt`, преобразованное (закодированное) для передачи по сети:

```
dataSender.formInput = escape(input_txt.text);
```

Пока пользователь ждет от *echo.pl* ответа на переданную форму, код показывает в клипе `response` сообщение о загрузке:

```
form_mc.response.gotoAndStop("loading");
```

С целью проверки мы выведем отправляемые данные в окно Output с помощью метода *toString( ):*

```
trace("Sending " + dataSender.toString());
```

Наконец, передача формы осуществляется с помощью *sendAndLoad( )*, а не *send( )* или *load( )*, поскольку мы одновременно отправляем и получаем переменные из *echo.pl*. Метод *sendAndLoad( )* посылает `formInput` (и все остальные свойства `dataSender`) для *echo.pl*:

```
dataSender.sendAndLoad("http://www.yourserver.com/cgi-bin/echo.pl",
                       dataReceiver, "GET");
```

Метод *sendAndLoad( )* принимает три параметра: *url* (адрес сценария сервера), *targetObject* (объект *LoadVars*, который должен получить результаты) и *method* (метод HTTP, «GET» или «POST»). При выполнении *sendAndLoad( )* свойства dataSender преобразуются в одну строку из пар имя–значение переменной, разделенных амперсандами, которая отправляется *echo.pl*.

Сценарий Perl echo.pl

При щелчке по кнопке Submit формы Flash начинается выполнение запроса HTTP GET, который запускает сценарий Perl *echo.pl*. Чтобы наша форма работала, *echo.pl* должен быть помещен в каталог веб-сервера, подключенный к CGI, и настроен администратором сервера следующим образом:

- Сценарий должен быть исполняемым (обычно для этого нужно установить права доступа к файлу сценария как 755).
- Если сценарий выполняется в Unix, в нем должен быть указан путь к интерпретатору Perl.

В примере 17.3 показан исходный код *echo.pl*. Учтите, что в Perl символ # обозначает комментарий.

Пример 17.3. Исходный код echo.pl

```
#! /usr/local/bin/perl
#-----------------------------------------------------------------------------
#  Название: Simple Flash Echo
#  Версия:   1.2.1
#  Автор:    Derek Clayton  derek@moock.org
#  Описание: Возвращает пары имя-значение, полученные от Flash GET или POST.
#-----------------------------------------------------------------------------
# MAIN
#-----------------------------------------------------------------------------
use CGI;                    # Использовать для простоты анализа CGI.pm
$query = new CGI;           # Объект запроса
$echoString = "output=";    # Инициализировать выходную строку
&getInput;                  # Получить данные от Flash
&writeResponse;             # Вывести ответ обратно Flash
exit;                       # Завершить сценарий
#-----------------------------------------------------------------------------
sub getInput {
  # Для каждого ключа получить связанное с ним значение и добавить в строку
  foreach $key ($query->param) {
    $value = $query->param($key);
    $echoString .= "$key:$value\n";
  }
  # Удалить завершающий символ перевода строки (\n) перед выводом ответа
  chomp($echoString);
}
```

```
sub writeResponse {
  # Установить тип контента для Flash
  print $query->header({"Content-Type"=> "application/x-www-form-urlencoded"});
  # Вывести данные
  print $echoString;
}
```

Сценарий *echo.pl* возвращает одну переменную, `output`, которая содержит одну строку, представляющую все переменные, отправленные нами серверу. Вот конкретные задачи, которые выполняет *echo.pl*:

1. Принимает данные, отправленные Flash, и выделяет в них имена переменных и их значения. Например, он преобразует свойство `dataSender.formInput` в переменную `formInput`. Остальные свойства, прикрепленные к `dataSender`, преобразуются в переменные аналогичным образом.
2. Собирает из этих имен переменных и их значений одну строку, которая должна быть возвращена Flash. Формат строки следующий:

   output=*name1*:*value1*\nname2:*value2*\n...*namen*:*valuen*

   Например, в ответ на передачу нашей формы *echo.pl* создает строку:

   output=formInput:*theInputValue*

   Если мы зададим для `dataSender` еще одно свойство, `country`, со значением «Canada», то *echo.pl* создаст строку:

   output=formInput:*theInputValue*\ncountry:Canada

3. Возвращает строку Flash.

При получении строки от *echo.pl* Flash находит переменную `output` и автоматически добавляет к `dataReceiver` свойство `output`. Значением `output` становится строка, содержащая список переменных, которые мы передали в *echo.pl*.

Ясно, что *echo.pl* служит только для проверки идеи. Однако применение этой идеи может привести к интересным и эффективным результатам. Образец более тщательно проработанной системы на Perl можно найти в примере с базой данных в текстовом файле, который есть в сетевом хранилище кода в разделе «Server Communication».

Получение результатов с сервера

Как вы помните, при отправке свойств `dataSender` сценарию *echo.pl* мы потребовали, чтобы возвращаемое сценарием *echo.pl* значение было сохранено в объекте `dataReceiver`:

```
dataSender.sendAndLoad("http://www.yourserver.com/cgi-bin/echo.pl",
                       dataReceiver, "GET");
```

При получении данных от *echo.pl* Flash выполняет функцию обратного вызова, присвоенную *dataReceiver.onLoad( )*. Мы присвоили эту функцию обратного вызова в функции *initForm( )*, показанной в примере 17.1. Рассмотрим ее теперь внимательнее:

```
dataReceiver.onLoad = function () {
    // Подготовить клип response к отображению данных.
    form_mc.response.gotoAndStop("doneLoading");

    // Показать значение загруженной переменной output в поле output_txt.
  form_mc.response.output_txt.text = this.output;
}
```

Эта функция перемещает головку воспроизведения response в кадр doneLoading, в котором расположено текстовое поле output_txt. Затем она отображает значение output (переданное сценарием *echo.pl*) в текстовом поле output_txt клипа response. Обратите внимание, что функция обратного вызова выполняется как метод dataReceiver, поэтому ссылка на объект dataReceiver внутри этой функции осуществляется с помощью this.

Осталось только проверить, как работает форма! Экспортируйте фильм, введите какой-нибудь текст в поле input и щелкните по кнопке Submit. Если ввести «test», то в текстовом поле output_txt должна появиться строка «formInput: test». Если форма сразу не сработает, проверьте правильность настройки сценария на сервере. И помните, что можно получить действующую версию из сетевого хранилища кода.

Упражнение для читателей: попробуйте поместить в форму кнопку Reset, которая очищает значение поля ввода. Создайте в formClip новую кнопку и прикрепите к ней код:

```
on (release) {
  input_txt.text = "";
}
```

Попробуйте также поместить на форму несколько полей ввода: сценарий Perl услужливо вернет все переменные, которые вы ему отправите, объединив их в переменной output. Сможете ли вы разделить их с помощью методов класса *String* и вывести значение каждой отдельно?

Вперед!

Должен сказать, мы неплохо поработали в этой главе. Подумайте, сколько всего можно делать с помощью форм Flash: запоминать настройки пользователя, создавать дискуссионные форумы или загружать в каталог динамические сведения о товарах. Пример с Perl и базой данных в текстовом файле из сетевого хранилища кода позволит вам еще дальше продвинуться по пути создания подобных приложений.

Наш разговор на ActionScript завершен, но ваш – только начинается. Прочтя часть I «Основы ActionScript», вы научились говорить на ActionScript, а теперь настало время применить эти знания к собственным проектам. Вот некоторые заключительные мысли:

- Постижение программирования, как и любого вида искусства, представляет собой процесс, а не событие. Во время написания программ вы больше узнаете о программировании. Вспомните хотя бы вопросник с вариантами

ответов – примеры 1.1–1.4 в главе 1, пример 9.13 в главе 9, пример 11.7 в главе 11 и табл. 12.2–12.5 в главе 12. Мы переделывали его четыре раза! Каждый раз мы уточняли подход, добавляли возможности и узнавали нечто, чего не рассматривали прежде. Так же как каждая картина учит художника чему-то новому в его области, так и создание и модификация приложений открывают новые подходы.

- В части II «Справочник по языку» вы найдете практическую помощь и подробное описание встроенных функций, свойств, классов и объектов ActionScript. Не обязательно читать его от начала до конца, но определенно стоит пробежать каждую тему, чтобы получить представление о возможностях ActionScript.
- Чаще заглядывайте в эту книгу, в том числе и в «Справочник по языку». Каждый раз вы будете обнаруживать что-то новое, поскольку будете изучать информацию на более высоком уровне понимания и сможете связать теорию с практическим опытом. Относитесь к «Справочнику по языку» как к словарю и храните его поблизости во время работы.
- Бурно растет сообщество разработчиков Flash, которые предлагают идеи и решения, а также, что очень важно, делятся исходным кодом! Разбирайте как можно больше кода. Формулируйте вопросы, которые вам не понятны, и ищите соответствующие темы в книге. Macromedia поддерживает список веб-сайтов и почтовых списков рассылки, посвященных Flash, на *http://www.macromedia.com/support/flash/ts/documents/flash_websites.htm*.
- При изучении ActionScript не ограничивайтесь этой книгой. Попробуйте взглянуть на предмет под другим углом, обратившись к другим источникам информации и вдохновения, включая перечисленные в приложении А и в предисловии. Вы также найдете большой список ресурсов на *http://www.moock.org/moockmarks* и рецензии на заслуживающие внимания книги по Flash на *http://www.moock.org/webdesign/books*.
- Наконец, не забывайте заглядывать на сайт поддержки *ActionScript for Flash MX: The Definitive Guide*, *http://www.moock.org/asdg*, где можно найти массу примеров кода, технических заметок и обсуждение новых тем.

На этом я желаю вам удачи в программировании! Пожалуйста, сделайте как-нибудь лишний проход в цикле *while* для меня лично. :)

II

Справочник по языку

Настоящий «Справочник по языку» разъясняет применение каждого встроенного класса, объекта, функции, свойства, обработчика событий, директивы и прагмы, поддерживаемых ActionScript. Он будет вашим верным помощником в решении конкретных задач, связанных с повседневной работой.

Справочник по языку ActionScript

В этом справочнике документируются все поддерживаемые ActionScript классы и объекты, описываются их общее назначение, применение, свойства, методы и обработчики событий. Кроме этого справочник охватывает все *глобальные функции* и *глобальные свойства* (то есть такие, которые не закреплены за каким-либо классом или объектом, а могут использоваться самостоятельно в любом месте фильма).

Все описания расположены в алфавитном порядке независимо от того, относятся ли они к свойствам, методам, глобальным функциям или обработчикам событий. Например, глобальная функция *duplicateMovieClip( )* представлена под литерой «D» после класса *Date* (а не, скажем, в отдельном разделе «Глобальные функции»). Однако в заголовке каждой статьи ясно указан тип описываемого элемента, поэтому путаницы в назначении или использовании элемента возникнуть не может. Кроме того, многие элементы приведены в алфавитном порядке объекта или класса, к которому они принадлежат. Например, *duplicateMovieClip( )* может иметь формат глобальной функции или метода клипа, поэтому даются две статьи: *duplicateMovieClip( ) глобальная функция* под «D» и *MovieClip.duplicateMovieClip( ) метод* под «M». Некоторые операторы, например *delete* и *typeof*, включены в справочник, но полное их описание (как и описание логических и арифметических операторов) следует искать в главе 5. О ключевых словах и циклах, таких как *if* и *for*, рассказывается в главах 6, 7 и 8. Если неясно, где искать какой-либо элемент, обратитесь к предметному указателю в конце книги.

Все статьи справочника обновлены и отражают дополнения и изменения, реализованные во Flash MX и Flash Player 6 (для краткости среду разработки Flash MX и подключаемый к броузеру модуль Flash Player 6 мы собирательно называем «Flash 6»). Приводится информация о совместимости с предыдущими версиями. Добавлены новые объекты и классы, а в существующих классах появились новые методы и свойства, поэтому следует бегло просмотреть весь справочник, чтобы познакомиться с возможностями текущей версии языка. В ряде случаев описания элементов, поддерживаемых Flash 5, дополнены более удачными примерами, даже если эти изменения не касаются непосредственно Flash 6.

Описание принятых соглашений по форматированию, принятых в данном справочнике, см. в предисловии в разделе «Типографские соглашения».

Глобальные функции

Глобальные функции – это встроенные функции, доступные в любом месте фильма: они могут быть вызваны из любого кадра, кнопки или обработчика событий фильма (в отличие от методов, которые должны вызываться с помощью ссылки на конкретный объект).

В табл. С.1 перечислены имеющиеся во Flash 6 глобальные функции, включая две новые глобальные функции *clearInterval( )* и *setInterval( )*. Все остальные перечисленные в ней глобальные функции имеются также и во Flash 5. Здесь же можно найти директивы и прагмы, начинающиеся с символа #, а именно `#include`, `#initclip` и `#endinitclip`.

Таблица С.1. Глобальные функции ActionScript

Boolean( )	*gotoAndStop( )*	*parseInt( )*	*stopAllSounds( )*
call( )	`#include`	*play( )*	*stopDrag( )*
*clearInterval( )**	`#initclip`*	*prevFrame( )*	*String( )*
Date( )	*int( )**	*prevScene( )*	*targetPath( )*
duplicate-MovieClip( )	*isFinite( )*	*print( )*	*tellTarget( )*†
`#endinitclip`*	*isNaN( )*	*printAsBitmap( )*	*toggleHighQuality( )*†
escape( )	*loadMovie( )*	*printAsBitmapNum( )*	*trace( )*
eval( )	*loadMovieNum( )*	*printNum( )*	*unescape( )*
fscommand( )	*loadVariables( )*	*random( )*†	*unloadMovie( )*
getProperty( )	*loadVariablesNum( )*	*removeMovieClip( )*	*unloadMovieNum( )*
getTimer( )	*nextFrame( )*	*setInterval( )**	*updateAfterEvent( )*
getURL( )	*nextScene( )*	*setProperty( )*	
getVersion( )	*Number( )*	*startDrag( )*	
gotoAndPlay( )	*parseFloat( )*	*stop( )*	

* Поддерживается Flash 6, но не более ранними версиями.

† Не рекомендуется применять во Flash 5 и более поздних версиях.

Глобальные свойства

К глобальным свойствам, как и к глобальным функциям, можно обращаться из любого сценария в фильме. В них хранится информация, которая может быть полезна в любой точке фильма. Многие из них оказывают воздействие на Flash Player в целом, а не только на конкретный клип или фильм.

В табл. С.2 перечислены глобальные свойства, имеющиеся во Flash 6, в том числе новое глобальное свойство `_global`. Остальные глобальные свойства, перечисленные в таблице, имеются также и во Flash 5.

Таблица C.2. Глобальные свойства ActionScript

Имя свойства	Описание
_focusrect	Подсвечивание кнопок, активизируемых с клавиатуры
_global*	Ссылка на глобальный объект
_highquality†	Качество вывода фильма проигрывателем
Infinity	Константа, представляющая значение Infinity *числового* типа
(Infinity	Константа, представляющая значение -Infinity *числового* типа
_leveln	Уровень документа проигрывателя, например -level0
NaN	Not-A-Number; значение *числового* типа, представляющее неверные числовые данные
_quality	Качество вывода проигрывателя
_root	Ссылка на главную временную диаграмму текущего уровня
_soundbuftime	Количество секунд предварительной загрузки потокового аудио
$version†	Версия Flash Player

* Поддерживается Flash 6, но не более ранними версиями.

† Не рекомендуется применять во Flash 5 и более поздних версиях.

Встроенные классы и объекты

В данном справочнике предполагается, что читателю понятны термины *класс*, *объект* и *экземпляр*, исчерпывающим образом рассмотренные в главе 12. Встроенные классы ActionScript применяются для создания объектов, которые могут управлять фильмом и манипулировать данными. Во Flash MX добавлены следующие встроенные классы ActionScript:

Button
Function
LocalConnection
LoadVars
TextField
TextFormat

Они дополнили классы *Array*, *Boolean*, *Color*, *Date*, *MovieClip*, *Number*, *Object*, *Sound*, *String*, *XML*, *XMLNode* и *XMLSocket*, которые поддерживались Flash 5.

Класс *MovieClip* был значительно усовершенствован во Flash 6. Самыми важными дополнениями стали Drawing API и поддержка событий кнопок.

Для создания экземпляра конкретного класса применяются функция *конструктора* класса и оператор *new*. Так, например, можно создать новый объект класса *Color* посредством конструктора *Color*:

```
myColor = new Color(_root);
```

Для каждого класса в этом справочнике под заголовком *Конструктор* показывается, как создавать объекты данного конкретного класса (то есть демонстрируется синтаксис функции конструктора каждого класса). Приводятся также сводка свойств, методов и обработчиков событий, имеющихся у объектов класса, и описание назначения и типичного применения класса. В некоторых классах определены методы или свойства, доступ к которым осуществляется через сам конструктор класса, а не отдельные экземпляры. Эти методы и свойства перечисляются в *Методах класса* и *Свойствах класса*. Полное описание свойств, методов и обработчиков событий для каждого класса приводится в алфавитном порядке после общего знакомства с классом.

Во Flash MX были добавлены следующие встроенные объекты ActionScript:

Accessibility

Capabilities

SharedObject

Stage

System

Они дополнили объекты *Arguments*, *Key*, *Math*, *Mouse* и *Selection*, имевшиеся во Flash 5. (См. в предисловии замечания по поводу объектов *CustomActions* и *LivePreview*, добавленных в Flash MX, но не попавших в справочник.) Объекты приводятся в алфавитном порядке, чередуясь с описаниями классов, но их можно отличить по слову *объект* (например, *Math объект*). В отличие от настоящих классов, которые создают экземпляры объектов с помощью функции конструктора, экземпляры этих самостоятельных объектов никогда не создаются (то есть для их образования не применяется оператор *new*). Напротив, это готовые объекты, существующие исключительно для объединения родственных функций в одном пакете. Объект *Math*, к примеру, предоставляет удобный доступ к математическим функциям и константам.

Заголовки статей

Заголовки статей справочника имеют следующий базовый формат:

Название элемента **Доступность**

Назначение **Тип доступа**

Синтаксис

Эти и другие заголовки, предшествующие описанию каждого элемента, описаны в табл. С.3.

Таблица С.3. Заголовки в «Справочнике по языку»

Заголовок	Описание
Доступность (Availability)	Указывает версию проигрывателя, начиная с которой элемент добавлен в ActionScript (или в состав действий, если элемент предшествует формальному ActionScript). Например, если указана *Доступность* «Flash 5», то эта функция поддерживается и во Flash 6, если не указано иное. Кроме того, если доступность функции явно или косвенно указывает на Flash 6, это означает ее поддержку как в среде разработки Flash MX, так и в подключаемом к броузеру модуле Flash Player 6, если не сказано иное. В этом заголовке могут также приводиться данные об обратной совместимости и статусе устаревшей функции
Тип доступа (Access)	Относится только к свойствам. Описывает, можно ли значение свойства только читать («только для чтения») или как получать, так и присваивать («чтение/запись»)
Назначение (Purpose)	Краткие сведения о назначении элемента
Синтаксис (Synopsis)	Иллюстрирует абстрактный синтаксис, требуемый элементом. Курсив в коде обозначает текст, который должен вводиться программистом в виде строки, числа, выражения или идентификатора. Некоторые функции можно вызывать, передавая им аргументы различных типов и в разном количестве; в таких случаях здесь показано несколько вариантов поддерживаемого синтаксиса. Обратите внимание, что в этом подразделе не указывается, являются ли аргументы функции, метода или конструктора необязательными. Такие данные приводятся в подразделе «Аргументы»
Аргументы	Относится только к методам и функциям. Описывает параметры метода или функции, перечисленные в описании *синтаксиса*
Операнды	Только в описаниях операторов. Описываются операнды рассматриваемого оператора, перечисленные в описании *синтаксиса*
Возвращаемое значение	Относится только к методам и функциям (свойства всегда «возвращают» значение, которое в них хранится). Описывает значение, возвращаемое методом, если таковое имеется. Если метод не возвращает никакого значения, этот заголовок опускается. Функции с недокументированными возвращаемыми значениями всегда возвращают `undefined`
Описание	Разъясняет действие элемента и возможности его применения в практических ситуациях
Особенности применения	Описывает примечательные особенности элемента
Ошибки в реализации	Описывает известные проблемы, связанные с элементом
Пример	Предлагает пример кода, в котором используется элемент. Короткие примеры могут встречаться в подразделе «Описание»
См. также	Список перекрестных ссылок на родственные темы в этом справочнике и материале части I «Основы ActionScript». Также приводятся ссылки на предметный указатель и ресурсы, перечисленные в предисловии и приложении А

Алфавитный справочник по языку

В последующих статьях документированы обработчики событий, методы и свойства объектов и классов, а также глобальных функций и свойств ActionScript. Чтобы найти операторы и инструкции, о которых говорится в каком-либо месте книги, обратитесь к предметному указателю. В данном справочнике документирован базовый язык ActionScript. Описание методов и свойств компонентов Flash UI, входящих в поставку Flash MX, а также разъяснение причин, по которым они здесь не документируются, см. в приложении G. Подробнее о компонентах и ресурсах, описывающих Flash UI, см. главу 14.

Accessibility, объект

Flash 6 (ActiveX только для Windows IE)

средства разработки компонентов интерфейса с облегченным доступом

Методы

isActive( )

Указывает, связана ли с Flash Player программа облегчения доступа (обычно «читалка» экрана).

sendEvent( )

Отправляет извещение о событии для Microsoft Active Accessibility (MSAA).

Описание

Объект *Accessibility* управляет связью между Flash Player и средствами чтения с экрана или вспомогательными средствами доступа, такими как Window-Eyes GW Micro (*http://www.gwmicro.com*). Метод *isActive( )* проверяет, связана ли с Flash Player программа облегчения доступа. Полезен при динамическом создании контента в фильме с учетом присутствия поддержки облегчения доступа. Метод *sendEvent( )* позволяет отправлять информацию о событиях компонентов интерфейса пользователя средствам облегчения доступа и предназначен разработчикам компонентов интерфейса с облегченным доступом.

На момент написания книги Macromedia ActionScript Component Accessibility (MASCA) API работает только с Microsoft Active Accessibility (MSAA), поэтому связь с внешним приложением облегчения доступа поддерживается только ActiveX-версией Flash Player 6. В будущем поддержка может быть расширена на другие версии Flash Player. Работа с MASCA API требует некоторого знания MSAA. Определить, есть ли в используемом проигрывателе поддержка облегченного доступа, можно с помощью `capabilities.hasAccessibility`.

Обратите внимание на то, что *Accessibility* – готовый объект, а не класс. Обращение к его свойствам, методам и обработчикам событий происходит непосредственно, например:

```
Accessibility.isActive();
```

Ряд деталей реализации *Accessibility* остается пока не вполне доступным сообществу разработчиков. Обычно в ActionScript этот объект требуется только для сложных компонентов интерфейса. Примеры можно найти во Flash UI Components, включенных в среду разработки Flash MX. По мере расширения производства компонентов

сторонними разработчиками Macromedia планирует опубликовать спецификации и учебные материалы по облегчению доступа.

Дополнительные сведения по облегчению доступа пользователя во Flash см. по адресу:

http://www.macromedia.com/macromedia/accessibility/features/flash/

См. также `capabilities.hasAccessibility`

Accessibility.isActive(), метод

Flash 6

указывает, связана ли с Flash Player программа облегчения доступа (обычно «читалка» экрана)

`Accessibility.isActive()`

Возвращаемое значение

Булево значение `true`, если программа облегчения доступа взаимодействует с проигрывателем, и `false` в противном случае.

Описание

Метод *isActive( )* возвращает булево значение, указывающее на то, что в данный момент работает внешняя программа облегчения доступа, такая как «читалка» экрана (screen reader), читающая содержимое, отображаемое Flash Player. Если средство чтения экрана активно, то *isActive( )* возвращает `true`, в противном случае возвращает `false`.

Не все версии Flash Player могут поддерживать связь с программой облегчения доступа. Узнать, поддерживает ли эту функцию конкретный проигрыватель, воспроизводящий фильм, можно с помощью `capabilities.hasAccessibility`. В настоящее время облегченный доступ поддерживает только ActiveX-версия Flash Player 6 для Windows Internet Explorer. Проигрыватели, не поддерживающие облегченный доступ, в том числе подключаемый модуль для Netscape для Windows и для Macintosh, всегда возвращают `false` в качестве значения *Accessibility.isActive( )*.

Метод *isActive( )* зависит от установления программой облегчения доступа связи с Flash Player. Имеет место небольшая задержка, отделяющая запуск Flash Player от получения первого такого сообщения, которой нельзя пренебрегать. Поэтому при вызове *isActive( )* в начале фильма Flash может быть возвращено значение `false`, даже если в действительности программа облегчения доступа запущена. Решить эту проблему можно с помощью задержки перед вызовом *isActive( )* в фильме в течение примерно одной секунды.

См. также *Accessibility.sendEvent( )*, `capabilities.hasAccessibility`

Accessibility.sendEvent(), метод

Flash 6 (только ActiveX для Windows IE)

отправляет извещение о событии для MSAA

`Accessibility.sendEvent(`*`movieClip`*`, `*`childID`*`, `*`event`*`, `*`isNonHtml`*`)`

Аргументы

movieClip

Экземпляр клипа, посылающий событие.

childID

Дочерний элемент интерфейса, которого касается событие (например, отдельный элемент в окне списка). Если этот аргумент равен 0, значит, событие относится к указанному `movieClip`, используйте 0.

event

Константа, обозначающая тип события. Используется часть имен событий и значений из множества констант событий MSAA, перечисленных в документации Microsoft по MSAA на *http://msdn.microsoft.com/at*.

isNonHtml

Булева величина, указывающая на то, что событие принадлежит (`false`) или не принадлежит (`true`) к одному из стандартных типов событий, которые могут генерироваться формой HTML. Установка этого аргумента в `true` предотвращает возникновение проблем в некоторых «читалках» экрана, которые интерпретируют контент Flash в Internet Explorer как часть страницы HTML. По умолчанию имеет значение `false`.

Описание

Метод *sendEvent( )* используется компонентами интерфейса с облегченным доступом для уведомления Microsoft Active Accessibility (MSAA) об изменении интерфейса компонента. MSAA передает сообщение приложению для чтения экрана, которое отображает изменение для пользователя. Например, клип, представляющий кнопку-переключатель, использует *sendEvent( )* для уведомления MSAA об изменении своего имени или состояния. Если `movieClip` не является экземпляром клипа или отсутствуют аргументы `childID` или `event`, то *sendEvent( )* завершает свою работу без генерации ошибок. Если нет запущенных средств облегчения доступа, то вызов *sendEvent( )* не оказывает никакого эффекта.

Метод *sendEvent( )* – лишь часть MASCA API, который на момент написания книги не был открыто документирован, но представляет собой важную часть стратегии компании в отношении компонентов Flash. API требует, чтобы компоненты определяли специальные объекты и константы для связи с MSAA. Например, компоненты, поддерживающие облегченный доступ, должны хранить некоторый объект в специальном свойстве `_accImpl`. В этом объекте, известном как *ActionScript Accessibility Object* (*ASAO*), реализуются методы и свойства, описывающие компонент на языке доступности и делающие его открытым для MSAA (похожие на интерфейс C++ *IAccessible*, для которого, по сути, ASAO играет роль упрощенной оболочки).

Если необходимые объекты и константы определены в соответствии с MASCA API, то компонент отображает взаимодействие с интерфейсом удобным способом при помощи *sendEvent( )*. Например, метод *setLabel( )* компонента-переключателя посредством *sendEvent( )* извещает программу чтения экрана об измененении метки кнопки:

```
FRadioButtonClass.prototype.setLabel = function(label)
{
  this.fLabel_mc.setLabel( label );
  this.txtFormat();
  // ACCESSIBILITY
  if(Accessibility.isActive()){
    Accessibility.sendEvent( this, 0, this.EVENT_OBJECT_NAMECHANGE );
  }
}
```

Дополнительные сведения об интерфейсе *IAccessible* см. на странице:

http://msdn.microsoft.com/library/en-us/msaa/msaapndx_6qlh.asp

Дополнительные сведения об MSAA можно найти на странице:

http://msdn.microsoft.com/library/en-us/msaa/msaastart_9w2t.asp

Microsoft периодически меняет ссылки на своем сайте для разработчиков. Если эти документы окажутся недоступными, обратитесь на *http://msdn.microsoft.com/at*.

См. также `capabilities.hasAccessibility`

Arguments, объект

Flash 5; во Flash 6 добавлено свойство caller

доступ к аргументам функции, к текущей функции и к вызвавшей функции

`arguments[`*`elem`*`]`
`arguments.`*`propertyName`*

Свойства

callee	Ссылка на вызываемую функцию.
caller	Ссылка на вызывающую функцию.
length	Количество аргументов, переданных вызываемой функции.

Описание

Объект *Arguments* хранится в локальной переменной `arguments` каждой функции и доступен только во время выполнения функции. *Arguments* является одновременно массивом и объектом. В качестве массива `arguments` хранит значения параметров, переданных функции, выполняемой в данный момент. Например, `arguments[0]` является первым переданным параметром, `arguments[1]` является вторым переданным параметром и т. д. В этом массиве передаются и хранятся все аргументы, включая те, которые формально не определены в качестве параметров. В качестве объекта *Arguments* хранит свойства `callee` и `caller`, которые можно использовать для идентификации или вызова текущей функции и функции, которая ее вызвала. Начиная с Flash 6 `arguments` поддерживает все методы и свойства массивов (*pop( )*, *shift( )* и т. д.); до Flash 6 `arguments` поддерживал только свойство `length`, а не методы класса *Array*.

См. также «Объект arguments» в главе 9

arguments.callee, свойство

Flash 5

ссылка на выполняемую в данный момент функцию **чтение/запись**

`arguments.callee`

Описание

Свойство `callee` содержит ссылку на выполняемую в данный момент функцию. Эта ссылка может использоваться для повторного выполнения данной функции или идентификации текущей функции путем сравнения.

Пример

```
function someFunction () {
  trace(arguments.callee == someFunction);    // Выводит: true
```

```
}

// Безымянная рекурсивная функция
countToTen = function () {
  i++;
  trace(i);
  if (i < 10) {
    arguments.callee();
  }
};

// Вызов безымянной рекурсивной функции
countToTen();
```

arguments.caller, свойство

Flash 6

ссылка на функцию, вызвавшую выполняемую в данный момент функцию **чтение/запись**

`arguments.caller`

Описание

Свойство `caller` содержит ссылку на функцию, которая вызвала текущую. Если текущая функция не была вызвана какой-то другой функцией, `caller` содержит `null`. Например:

```
// Создать функцию, которая вызывает две другие функции
function formatPage () {
  formatTitle();
  formatBody();
}

function formatTitle () {
  // Убедиться, что данную функцию вызвала formatPage()
  trace(arguments.caller == formatPage);
}

function formatBody () {
  // Убедиться, что данную функцию вызвала formatPage()
  trace(arguments.caller == formatPage);
}

// Вызвать formatTitle() и formatBody() из formatPage()

formatPage();
// Выводит: true и true

// Теперь вызвать formatTitle() непосредственно
formatTitle();
// arguments.caller будет null. Выводит: false
```

Посредством `caller` можно выполнить или идентифицировать вызывающую функцию и проверить, вызвана ли функция непосредственно или из другой функции.

Применение

Обратите внимание, что в Netscape 4 свойство `arguments.caller` JavaScript указывает на объект *Arguments* вызывающей функции, а не на саму вызывающую функцию. В JavaScript `arguments.caller` считается устаревшим, начиная с версии 1.3, и не поддерживается Internet Explorer или более свежими версиями Netscape.

Ошибки реализации

В первоначальной документации Macromedia по Flash MX ошибочно утверждается, что `arguments.caller` указывает на объект *Arguments* вызывающей функции. В действительности это ссылка на саму вызывающую функцию, а не на ее объект *Arguments*.

Пример

В следующем коде приводится функция, которая прерывает свое выполнение, если она вызвана непосредственно, что гарантирует ее работу только при вызове из более крупной подпрограммы:

```
function subFunction () {
  if (arguments.caller == null) {
    return "Function cannot be called directly.";
  }
  // продолжить выполнение функции...
}
```

См. также `arguments.callee`

arguments.length, свойство

Flash 5

количество параметров, переданных в аргумент **чтение/запись**

`arguments.length`

Описание

Свойство `length` хранит целое число, представляющее количество элементов в массиве `arguments`, равное количеству параметров, переданных функции, выполняющейся в данное время.

Пример

С помощью свойства `length` объекта `arguments` можно определить, было ли при вызове функци передано правильное количество параметров. В следующем примере проверяется, было ли функции *someFunction( )* передано два параметра. Проверка правильности типа переданных аргументов оставляется читателю в качестве упражнения. (Совет: см. описание оператора *typeof*.) Вот этот код:

```
function someFunction (y, z) {
  if (arguments.length != 2) {
    trace("Функция вызвана с неверным числом параметров");
    return;
  }
  // Выполнение обычного тела функции...
}
```

Array, класс

Flash 5; во Flash 6 добавлен метод sortOn()

поддержка упорядоченных списков данных

Конструктор

`new Array()`
`new Array(`*`len`*`)`
`new Array(`*`element0, element1, element2,...elementn`*`)`

Аргументы

len

Неотрицательное целое число, задающее размер нового массива.

element0,...elementn

Список из одного или нескольких начальных значений, которые должны быть присвоены элементам массива.

Свойства

length

Количество элементов в массиве (включая пустые элементы).

Методы

concat( )

Создать новый массив, дописав новые элементы к существующему массиву.

join( )

Преобразовать элементы массива в строки и объединить, использовав указанный разделитель.

pop( )

Удалить последний элемент массива и возвратить его.

push( )

Добавить один или несколько элементов в конец массива.

reverse( )

Изменить порядок элементов в массиве на противоположный.

shift( )

Удалить первый элемент массива и вернуть его.

slice( )

Создать новый массив из подмножества элементов существующего массива.

sort( )

Сортировать элементы массива согласно заданному правилу.

sortOn( )

Сортировать массив объектов по заданному свойству.

splice( )

Удалить элементы массива и/или добавить новые.

toString( )

Преобразовать массив в строку разделенных запятыми значений элементов.

unshift( )

Добавить в начало массива один или несколько элементов.

Описание

Свойства и методы класса *Array* используются для манипулирования данными, хранящимися в элементах объекта массива. Полное описание массивов и способов работы с ними, а также терминологии данного раздела можно найти в главе 11. См. также операторы [ ] и . (точка), применяемые для доступа к элементам массивов и образования литералов массивов, описанных в главе 5.

Применение

Если конструктор *Array* вызывается с одним целочисленным аргументом, то этот аргумент определяет длину нового массива, а не используется как значение первого элемента. Если конструктору передаются два или более аргументов, или один аргумент, который не является числом, то аргументы используются в качестве начальных значений элементов массива, а длина массива определяется количеством заданных аргументов.

Array.concat(), метод

Flash 5

создает новый массив путем расширения существующего

array.concat(*value1*, *value2*, *value3*, . . . *valuen*)

Аргументы

value1,...*valuen*

Список выражений, которые должны быть добавлены в конец *array* в качестве новых элементов.

Возвращаемые значения

Новый массив, содержащий все элементы *array*, за которыми следуют элементы *value1*, . . . *valuen*.

Описание

Метод *concat( )* возвращает новый массив, создаваемый дописыванием новых элементов в конец существующего массива. Исходный массив *array* остается в неизменном виде. Для модификации исходного массива используйте методы *push( )*, *splice( )* или *shift( )*.

Если в качестве аргумента *concat( )* используется массив, каждый его элемент добавляется отдельно. Это означает, что результатом *arrayX.concat(arrayY)* будет массив, образованный добавлением каждого элемента arrayY в конец arrayX. Для результирующего массива свойство length будет равно arrayY.length + arrayX.length. Однако вложенные массивы не выравниваются подобным образом.

Пример

```
// Создать массив
myListA = new Array("apples", "oranges");

// Присвоить myListB значение ["apples", "oranges", "bananas"]
myListB = myListA.concat("bananas");

// Создать еще один новый массив
myListC = new Array("grapes", "plums");

// Присвоить myListD значение ["apples", "oranges", "bananas", "grapes", "plums"]
myListD = myListB.concat(myListC);

// Присвоить myListA значение ["apples", "oranges", "bananas"]
myListA = myListA.concat("bananas");

// Создать массив
settings  = ["on", "off"];
// Приписать массив, содержащий вложенный массив
options = settings.concat(["brightness", ["high", "medium", "low"]]);
```

```
// Присвоить options значение: ["on", "off", "brightness", ["high", "medium", "low"]]
// не: ["on", "off", "brightness", "high", "medium", "low"]
```

См. также *Array.push( )*, *Array.shift( )*, *Array.splice( )*; «Метод *concat( )*», глава 11

Array.join(), метод

Flash 5

преобразует массив в строку

array.join()
array.join(*delimiter*)

Аргументы

delimiter

Необязательная строка, которая должна быть помещена между элементами во вновь создаваемой строке. Если разделитель не указан, по умолчанию в этом качестве выступает запятая.

Возвращаемое значение

Строка, составленная из всех элементов *array*, преобразованных в строки и разделяемых *delimiter*.

Описание

Метод *join( )* возвращает строку, созданную путем объединения всех элементов массива, для чего он:

1. Преобразует каждый элемент массива в строку в соответствии с правилами, описанными в табл. 3.2 главы 3.
2. Добавляет *delimiter* в конец каждого преобразованного в строку элемента, кроме последнего.
3. Конкатенирует преобразованные в строки элементы в одну длинную строку.

Обратите внимание, что элементы, которые сами являются массивами, преобразуются в строки посредством метода *toString( )*, поэтому элементы вложенных массивов отделяются один от другого запятыми, а не ограничителем, указанным при вызове *join( )*.

Пример

```
fruit = new Array("apples","oranges","bananas","grapes","plums");
// Присвоить fruitString значение "apples,oranges,bananas,grapes,plums"
// (разделитель по умолчанию - запятая без пробела после нее)
fruitString = fruit.join();
// Присвоить fruitString значение "apples, oranges, bananas, grapes, plums"
// (с пробелом после каждой запятой)
fruitString = fruit.join(", ");
```

См. также *Array.toString( )*, *String.split( )*; «Метод *join( )*» в главе 11.

Array.length, свойство

Flash 5

количество элементов в массиве **чтение/запись**

array.length

Описание

Свойство `length` представляет собой неотрицательное целое число, задающее количество элементов в массиве. Массив без элементов имеет длину 0, массив из двух элементов имеет длину 2. Обратите внимание, что индекс первого элемента массива равен 0, поэтому `length` всегда на единицу больше индекса последнего элемента массива.

Свойство массива `length` указывает количество пронумерованных в данный момент элементов массива, включая пустые (те, которые содержат `null` или `undefined`). Например, массив может содержать значения в элементах 0, 1, 2 и 9, а элементы с 3-го по 8-й могут быть пустыми. Длина такого массива равна 10, потому что в нем 10 позиций элементов (от 0 до 9), несмотря на то что только 4 позиции заняты полезными значениями.

Свойство `length` автоматически изменяется при добавлении или удалении элементов с помощью методов класса *Array*, таких как *push( )*. Свойство `length` отражает только количество нумерованных элементов; оно не учитывает именованные элементы, которые рассматриваются как свойства объекта *Array*. Хотя обычно свойство `length` изменяется косвенным образом в результате выполнения методов класса *Array*, `length` можно изменять непосредственно. Присваивание значения свойству `length` изменяет количество элементов в массиве. Если увеличить `length`, то в конец массива будут добавлены пустые элементы со значением `undefined`, а если уменьшить `length`, то существующие элементы удаляются из конца массива без дополнительных предупреждений, поэтому предпочтительней удалять элементы массива с помощью *splice( )* или *pop( )*.

Пример

```
myList = new Array("one", "two", "three");
trace(myList.length);  // Выводит: 3

// Обойти элементы массива в цикле
for (var i = 0; i < myList.length; i++) {
  trace(myList[i]);
}
```

См. также «Определение размера массива» в главе 11

Array.pop(), метод

Flash 5

удаляет последний элемент массива

array.pop()

Возвращаемое значение

Значение последнего элемента массива *array*, который (элемент) при этом удаляется.

Описание

Метод *pop( )* удаляет *последний* элемент массива, уменьшает свойство `length` на 1 и возвращает значение удаленного элемента. Сравните с методом *shift( )*, который удаляет *первый* элемент массива.

Пример

```
myList = new Array("one", "two", "three");
trace ("Now deleting " + myList.pop());  // myList теперь: ["one", "two"]
```

См. также

Array.push( ), *Array.shift( )*, *Array.splice( )*; «Удаление элементов из массива» в главе 11, «Оператор *delete*» в главе 5

Array.push(), метод

Flash 5

добавляет один или более элементов в конец массива

`array.push(value1, value2, ...valuen)`

Аргументы

value1,...valuen

Список из одного или более значений, которые должны быть добавлены в конец *array*.

Возвращаемое значение

Новое значение `length` массива *array*.

Описание

Метод *push( )* присоединяет список значений к концу массива в качестве новых элементов. Элементы добавляются в том порядке, в котором передаются. Отличие от *concat( )* в том, что *push( )* модифицирует исходный массив, а *concat( )* создает новый массив. Отличие от *unshift( )* в том, что элементы добавляются в конец массива, а не в его начало.

Пример

```
myList = new Array (5, 6);
myList.push(7);                  // Теперь myList равен [5, 6, 7]
myList.push(10, 8, 9);           // Теперь myList равен [5, 6, 7, 10, 8, 9]
```

См. также

Array.concat( ), *Array.pop( )*, *Array.unshift( )*; «Добавление элементов в массив» в главе 11

Array.reverse(), метод

Flash 5

изменяет порядок элементов массива на противоположный

`array.reverse()`

Описание

Метод *reverse( )* изменяет порядок элементов массива на противоположный, меняя местами последний элемент с первым, предпоследний со вторым и т. д.

Пример

```
myList = new Array(3, 4, 5, 6, 7);
myList.reverse();                // Теперь myList равен [7, 6, 5, 4, 3]
```

См. также

Array.sort( )

Array.shift(), метод

Flash 5

удаляет первый элемент массива

`array.shift()`

Возвращаемое значение

Значение первого элемента *array*, который удаляется.

Описание

Метод *shift( )* удаляет *первый* элемент массива, после чего сдвигает все оставшиеся элементы вверх на одну позицию. Длина массива уменьшается на 1. Обратите внимание на отличие *shift( )* от метода *pop( )*, удаляющего *последний* элемент массива.

Пример

```
myList = new Array ("a", "b", "c");
myList.shift();                    // myList становится равным ["b", "c"]
```

См. также

Array.pop( ), *Array.splice( )*, *Array.unshift( )*; «Удаление элементов из массива» в главе 11

Array.slice(), метод

Flash 5

создает новый массив из подмножества элементов существующего массива

array.slice(*startIndex*, *endIndex*)

Аргументы

startIndex

Целое число с отсчетом от нуля, указывающее первый элемент *array*, который должен быть добавлен в новый массив. Если значение отрицательное, то *startIndex* указывает номер элемента, отсчитываемый обратно с конца массива (-1 задает последний элемент, -2 – предпоследний и т. д.).

endIndex

Целое число, задающее элемент за *последним* элементом *array*, который должен быть добавлен в новый массив. Если значение отрицательное, *endIndex* ведет обратный отсчет с конца массива *array* (-1 задает последний элемент, -2 – предпоследний и т. д.). Если *endIndex* опущен, то по умолчанию принимается равным *array*.length.

Возвращаемое значение

Новый массив, содержащий элементы *array* от *startIndex* до *endIndex*-1.

Описание

Метод *slice( )* создает новый массив, извлекая последовательность элементов из существующего массива. Исходный массив не изменяется. Новый массив представляет собой подмножество элементов исходного массива, начиная с *array*[*startIndex*] и заканчивая *array*[*endIndex*-1].

Пример

```
myList = new Array("a", "b", "c", "d", "e");

// Присвоить myOtherList значение ["b", "c", "d"]
myOtherList = myList.slice(1, 4);

// Присвоить anotherList значение ["d", "e"]
anotherList = myList.slice(3);

// Присвоить yetAnotherList значение ["c", "d"]
yetAnotherList = myList.slice(-3, -1);
```

См. также

Array.splice( ); «Метод *slice( )*» в главе 11; «Оператор *delete*» в главе 5

Array.sort(), метод

Flash 5

сортирует элементы массива

```
array.sort()
array.sort(compareFunction)
```

Аргументы

compareFunction

Необязательная функция, которая определяет, как нужно сортировать `array`.

Описание

При вызове метода *sort( )* без необязательного аргумента `compareFunction` элементы массива временно преобразуются в строки и упорядочиваются в соответствии с кодовыми позициями этих строк в Unicode (приблизительно в алфавитном порядке для западноевропейских языков). Сравнения по алфавиту и кодовые позиции описываются в разделе «Сравнение по порядку символов и алфавиту» в главе 4.

При вызове с аргументом `compareFunction` метод *sort( )* переупорядочивает элементы `array` в соответствии со значениями, возвращаемыми `compareFunction` – определенной пользователем функцией, задающей порядок для любой пары элементов массива. Определенная пользователем функция `compareFunction` должна принимать в качестве аргументов два элемента массива и возвращать отрицательное число, если первый элемент должен следовать после второго, положительное число, если первый элемент должен предшествовать второму, и 0, если порядок элементов должен быть сохранен. Если после сортировки в массив добавляются новые элементы, они размещаются *не* в соответствии с порядком сортировки. Чтобы переупорядочить вновь добавленные элементы, необходимо заново провести сортировку массива. Обратите внимание, что по умолчанию числа сортируются в соответствии с их кодовыми позициями в Unicode (которые совпадают с кодовыми позициями в Latin 1 и ASCII). В главе 11 разъясняется, как сортировать числа по их значениям. Чтобы провести сортировку для букв Latin 1 в алфавитном порядке без учета регистра, преобразуйте строки в верхний регистр с помощью *String.toUpperCase( )*, прежде чем вызывать *sort( )*, либо сделайте это внутри функции `compareFunction`, как показано в примере 11.6.

Для сортировки группы объектов в соответствии со значением заданного свойства применяется метод *sortOn( )*.

Ошибки реализации

Реализация *Array.sort( )* во Flash Player 5 содержит ряд ошибок, о которых сказано на странице *http://moock.org/asdg/technotes/arraySortBugs*. Эти ошибки исправлены во Flash Player 6.

Пример

В следующем примере массив клипов сортируется в соответствии с их горизонтальным расположением на экране. Это удобно для создания эффекта затухания, перемещающегося по экрану слева направо, когда клипы затухают в зависимости от своего положения, даже если оно меняется динамически.

```
var clips = [clip1, clip2, clip3, clip4];

function compareXposition (element1, element2) {
  if (element1._x < element2._x) {
    return -1;
```

```
  } else if (element1._x > element2._x) {
    return 1;
  } else {
    return 0;  // У клипов одинаковая координата x
  }
}

clips.sort(compareXposition);
```

См. также

Array.reverse( ), *Array.sortOn( )* ; «Методы *sort( )* и *sortOn( )*» в главе 11

Array.sortOn(), метод

Flash 6

сортирует объекты по заданному свойству

array.sortOn(*key*)

Аргументы

key

Обязательная строка, задающая свойство, по которому должны сортироваться объекты в массиве *array*.

Описание

Метод *sortOn( )* применяется для сортировки объектов в соответствии со значением указанного свойства. Элементы *array* должны быть объектами. У каждого объекта должно быть свойство, заданное строкой *key*, по которому будут сортироваться объекты. Если *key* не задано, *sortOn( )* завершает работу, не генерируя ошибку, и сортировка не выполняется.

Следующий код заполняет массив из трех элементов с помощью литералов объектов, в которых определены свойства fruit и meat:

```
var shoppingList = new Array();
shoppingList[0] = {fruit:"mango", meat:"beef"};
shoppingList[1] = {fruit:"orange", meat:"pork"};
shoppingList[2] = {fruit:"apple", meat:"chicken"};
```

Для сортировки массива shoppingList по свойству fruit служит код:

```
shoppingList.sortOn("fruit");
// shoppingList теперь содержит:
// ELEMENT 0: {fruit:"apple", meat:"chicken"}
// ELEMENT 1: {fruit:"mango", meat:"beef"}
// ELEMENT 2: {fruit:"orange", meat:"pork"};
```

Обычно объекты массива представляют собой экземпляры некоторого класса, например:

```
// Создать простой класс Contact со свойствами firstName и lastName
function Contact (firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

// Переопределить toString() для нашего класса, чтобы он возвращал легко читаемое значение
Contact.prototype.toString = function () {
```

```
  return this.firstName + " " + this.lastName;
}

// Заполнить массив contacts экземплярами Contact
var contacts = new Array();
contacts[0] = new Contact("Colin", "Moock");
contacts[1] = new Contact("Bruce", "Epstein");
contacts[2] = new Contact("Derek", "Clayton");

// Сортировать по lastName...
contacts.sortOn("lastName");
trace(contacts);  // Выводит: Derek Clayton,Bruce Epstein,Colin Moock
// Сортировать по firstName...
contacts.sortOn("firstName");
trace(contacts);  // Выводит: Bruce Epstein,Colin Moock,Derek Clayton
```

Обратите внимание на использование в *sortOn( )* стандартной функции сравнения для сортировки значений (как при вызове *sort( )* без аргументов). Метод *sortOn( )* не дает возможности использовать пользовательскую функцию сравнения. В процедурах сортировки, требующих пользовательской функции сравнения, применяйте метод *sort( )*. Для того чтобы сортировка по алфавиту выполнялась независимо от регистра букв Latin 1, преобразуйте значения свойства в верхний регистр с помощью *String.toUpperCase( )*, прежде чем вызывать *sortOn( )*.

Если элемент не содержит объекта со свойством *key*, он преобразуется в строку, и порядок для него устанавливается в соответствии с правилами сортировки, действующими по умолчанию.

Ошибки реализации

Во Flash Player 6 применение *sortOn( )* к массиву, в котором содержатся как строки, так и объекты, может привести к аварийному завершению работы проигрывателя и броузера, в котором он выполняется. Например, следующий код приводит к аварийному завершению Flash Player 6:

```
var shoppingList = new Array();
shoppingList[0] = {fruit:"mango", meat:"beef"};
shoppingList[1] = "hi";
shoppingList[2] = {fruit:"apple", meat:"chicken"};

shoppingList.sortOn("fruit");
```

Поэтому разумно воспользоваться функцией-оболочкой, которая сначала проверяет, все ли элементы массива представляют собой объекты, и только потом вызывает *sortOn( )*, например:

```
Array.prototype.safeSortOn = function (key) {
  for (i=0; i < this.length; i++) {
    if (typeof this[i] != "object") {
      trace ("Element " + i + " is not an object.");
      return;
    }
  }
  // Если проверка успешна, вызывается sortOn()
  this.sortOn(key);
}

// Применение:
// Не приводит к аварии Flash Player 6.
shoppingList.safeSortOn("fruit");
```

Порядок сортировки оказывается неопределенным, когда свойство *key* имеет одинаковое значение для нескольких элементов. В таком случае результаты *sortOn( )* могут различаться при повторных вызовах. Например:

```
function Contact (firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Contact.prototype.toString = function () {
  return this.firstName + " " + this.lastName;
}

// Заполнить массив contacts экземплярами Contact с одинаковыми значениями свойства lastName.
var contacts = new Array();
contacts[0] = new Contact("Makiko", "Pitaru");
contacts[1] = new Contact("Amit",   "Pitaru");
contacts[2] = new Contact("Richard", "Clayton");
contacts[3] = new Contact("Derek",   "Clayton");

// Сортировать по lastName...
contacts.sortOn("lastName");
trace(contacts);  // Выводит:
                  // Richard Clayton,Derek Clayton,Makiko Pitaru,Amit Pitaru

// СНОВА сортировать по lastName...
contacts.sortOn("lastName");
trace(contacts);  // Выводит ДРУГИЕ РЕЗУЛЬТАТЫ:
                  // Richard Clayton,Derek Clayton,Amit Pitaru,Makiko Pitaru
```

См. также *Array.sort( )*

Array.splice(), метод

Flash 5

удаляет элементы из массива и/или добавить элементы

array.splice(*startIndex*)
array.splice(*startIndex*, *deleteCount*)
array.splice(*startIndex*, *deleteCount*, *value1*, ...*valuen*)

Аргументы

startIndex

Индекс элемента (отсчитываемый с нуля), начиная с которого надо удалять элементы и (необязательно) вставлять новые. Если значение отрицательное, то *startIndex* указывает номер элемента, отсчитываемый обратно с конца массива (-1 задает последний элемент, -2 – предпоследний и т. д.).

deleteCount

Необязательное неотрицательное целое число, задающее количество элементов, которые должны быть удалены из массива, включая элемент с индексом *startIndex*. Если оно равно 0, никакие элементы не удаляются. Если оно опущено, удаляются все элементы, начиная со *startIndex* и до конца массива.

value1,...*valuen*

Необязательный список из одного или более значений, добавляемых в *array* с индекса *startIndex* после удаления заданных элементов. Если список не задан, никакие элементы не добавляются.

Возвращаемое значение

Новый массив, содержащий *удаленные* элементы (исходный массив модифицируется отдельно, в соответствии с запрошенными изменениями).

Описание

Метод *splice( )* удаляет элементы массива с `array[startIndex]` по `array[startIndex + deleteCount-1]` и затем может дополнительно вставить новые элементы, начиная с `startIndex`. Метод *splice( )* не оставляет разрывов в `array`: он перемещает оставшиеся элементы, чтобы обеспечить непрерывность элементов в массиве.

Пример

```
myList = new Array (1, 2, 3, 4, 5);
// Удаляет второй и третий элементы списка
// и вставляет на их место "x", "y" и "z".
// myList становится равным [1, "x", "y", "z", 4, 5].
myList.splice(1, 2, "x", "y", "z");
```

См. также *Array.slice( )*; «Метод *splice( )*» в главе 11

Array.toString(), метод

Flash 5

преобразует массив в строку значений элементов, разделенных запятыми

`array.toString()`

Возвращаемое значение

Список с разделителем-запятой элементов `array`, преобразованных в строки.

Описание

Метод *toString( )* создает строковое представление массива `array`. Строка, возвращаемая *toString( )*, представляет собой список элементов массива, преобразованных в строки и разделенных запятыми (таким же, как возвращаемый методом *join( )* при вызове без параметров). Метод массива *toString( )* автоматически вызывается, если массив используется в строковом контексте. Поэтому необходимость в выполнении *toString( )* вручную возникает редко. Как правило, если требуется точное строковое представление массива, применяется метод *join( )*, предоставляющий больше контроля над создаваемой строкой.

Пример

```
myList = new Array("a", "b", "c");            // Создать массив
trace(myList.toString());                     // Выводит: "a","b","c"
myList = new Array([1, 2, 3], "a", "b", "c"); // Создать вложенный массив
trace(myList.toString());                     // Выводит: "1,2,3,a,b,c"
```

См. также *Array.join( )*

Array.unshift(), метод

Flash 5

добавляет один или более элементов в начало массива

`array.unshift(value1, value2,...valuen)`

Аргументы

value1,...valuen

Список из одного или более значений элементов, которые надо добавить в начало массива *array*.

Возвращаемое значение

Новое значение свойства `length` массива *array*.

Описание

Метод *unshift( )* добавляет ряд элементов в начало массива. Элементы добавляются в порядке их указания. Для добавления элементов в конец массива предназначен массив *push( )*.

Пример

```
myList = new Array (5, 6);
myList.unshift(4);         // myList становится равным [4, 5, 6]
myList.unshift(7, 1);      // myList становится равным [7, 1, 4, 5, 6]
```

См. также

Array.push( ), *Array.shift( )*; «Добавление элементов в массив» в главе 11

Boolean(), глобальная функция

Flash 5

преобразует значение к типу данных boolean

`Boolean(value)`

Аргументы

value Выражение, значение которого должно быть преобразовано к булеву типу.

Возвращаемое значение

Результат преобразования *value* в элементарное булево значение (`true` или `false`).

Описание

Глобальная функция *Boolean( )* преобразует свой аргумент в элементарное булево значение и возвращает результат преобразования. Результаты преобразований различных типов данных в элементарное булево значение описаны в табл. 3.3 главы 3. Обычно необходимости в применении функции *Boolean( )* нет: ActionScript автоматически преобразует значения в булев тип, когда это требуется.

Особенности применения

Не спутайте глобальную функцию *Boolean( )* с конструктором класса *Boolean*. Функция *Boolean( )* преобразует свой аргумент в булев тип данных, тогда как конструктор класса *Boolean* создает новый булев объект. Обратите внимание, что в ECMA-262 все непустые строки преобразуются в `true`. Во Flash 5 в `true` преобразуются только те строки, которые могут быть преобразованы в допустимое число, отличное от нуля. Поэтому даже строка «true» преобразуется в булево значение `false`.

Пример

```
var x = 1;
if (Boolean(x)) {
```

```
    trace("x is true");
  }
```

См. также Класс *Boolean*; «Преобразование типов данных» в главе 3

Boolean, класс

Flash 5

класс-оболочка для элементарных булевых данных

Конструктор

`new Boolean(value)`

Аргументы

value

Выражение, которое должно быть разрешено и при необходимости преобразовано в булево значение, после чего помещено в оболочку объекта *Boolean*. Если аргумент *value* отсутствует, записывается булево значение `false`.

Методы

toString( ) Преобразует значение объекта *Boolean* в строку.

valueOf( ) Извлекает элементарное значение объекта *Boolean*.

Описание

Класс *Boolean* создает объект *Boolean*, содержащий элементарное булево значение во внутреннем свойстве. Объекты *Boolean* используются исключительно ради удобства действий с элементарными булевыми значениями с помощью методов класса *Boolean*. Поэтому объект *Boolean* называют *объектом-оболочкой* (*wrapper object*), так как он просто упаковывает элементарное булево значение, придавая ему некоторые методы объектного вида. Сравните класс *Boolean* с классами *String* и *Number*, которые также служат оболочками элементарных значений *string* и *number*, но более полезны.

По большей части объекты *Boolean* используются внутренним образом. Они автоматически создаются интерпретатором, когда к элементарному булеву значению применяется метод, и автоматически уничтожаются после каждого использования. Объекты *Boolean* можно создавать самостоятельно с помощью конструктора *Boolean*, но в этом редко возникает необходимость.

Применение

Обратите внимание, что на практике гораздо чаще в качестве средства преобразования типа данных применяется глобальная функция *Boolean( )*, а не класс *Boolean*.

См. также *Boolean( )* глобальная функция; «Булев тип данных» в главе 4

Boolean.toString(), метод

Flash 5

значение объекта Boolean, преобразованное в строку

`booleanObject.toString()`

Возвращаемое значение

Строка «true», если элементарным значением *booleanObject* является `true`; «false», если элементарное значение *booleanObject* равно `false`. Значение булева объекта указывается при его создании, после чего хранится внутри самого объекта. Хотя внутрен-

нее значение булева объекта нельзя получить с помощью *valueOf( )*, с помощью *toString( )* можно преобразовать его в строковый эквивалент.

Описание

Метод *toString( )* извлекает элементарное значение булева объекта, преобразует это значение в строку и возвращает полученную строку.

Пример

```
x = new Boolean(true);
trace(x.toString());  // Выводит: "true"
```

См. также *Object.toString( )*

Boolean.valueOf(), метод

Flash 5

элементарное значение объекта Boolean

booleanObject.valueOf()

Возвращаемое значение

Булево значение `true`, если элементарное значение *booleanObject* равно `true`; `false`, если элементарное значение *booleanObject* равно `false`. Значение булева объекта указывается при его создании, после чего хранится внутри самого объекта.

Описание

Метод *valueOf( )* возвращает элементарное булево значение, связанное с булевым объектом. Метод *valueOf( )* возвращает аргумент *value*, с которым был вызван конструктор, преобразованным в эквивалент элементарного булева типа.

Пример

```
x = new Boolean(0);
trace(x.valueOf());  // Выводит: false
```

См. также *Object.valueOf( )*

Button, класс

Flash 6

обеспечивает управление кнопками в фильме

Конструктор

Отсутствует. Объекты *Button* создаются вручную в среде разработки. Моделировать кнопки, создаваемые на этапе исполнения, можно с помощью клипов.

Свойства

_alpha

Процент непрозрачности: 0 соответствует прозрачности, 100 – непрозрачности.

enabled

Булево значение; разрешает или запрещает взаимодействие с кнопкой.

_focusrect

Булево значение; разрешает или запрещает автоматическое выделение кнопок, активизированных посредством клавиатуры.

_height

Высота кнопки в пикселах.

_name

Имя экземпляра объекта *Button*.

_parent

Ссылка на клип, в котором находится кнопка.

_rotation

Поворот кнопки по часовой стрелке в градусах.

tabEnabled

Булево значение; включает или выключает кнопку из обхода по клавише табуляции.

tabIndex

Задает индекс кнопки при обходе по клавише табуляции.

_target

Путь к приемнику кнопки в синтаксисе с косой чертой.

trackAsMenu

Модифицирует поведение обработчика *onRelease( )* события кнопки: включает отображение меню.

_url

Сетевой адрес клипа, содержащего кнопку.

useHandCursor

Булево значение, определяющее показ курсора мыши в виде ладони при нахождении над кнопкой.

_visible

Булево значение; определяет, видима кнопка или скрыта.

_width

Ширина кнопки в пикселах.

_x

Горизонтальное положение точки регистрации кнопки в пикселах.

_xmouse

Горизонтальное положение курсора мыши относительно точки регистрации кнопки в пикселах.

_xscale

Ширина кнопки, выраженная в процентах от первоначального значения.

_y

Вертикальное положение точки регистрации кнопки в пикселах.

_ymouse

Вертикальное положение курсора мыши относительно точки регистрации кнопки в пикселах.

_yscale

Высота кнопки, выраженная в процентах от первоначального значения.

Методы

getDepth( )

Возвращает положение кнопки в стеке содержимого.

Обработчики событий

on (keyPress)

Происходит при нажатии указанной клавиши.

onDragOut( )

Происходит, когда выполняется щелчок кнопкой мыши при нахождении курсора над кнопкой, а затем курсор уводится с кнопки Flash при нажатой кнопке мыши.

onDragOver( )

Происходит, когда выполняется щелчок кнопкой мыши при нахождении курсора над кнопкой, а затем курсор уводится с кнопки Flash и снова возвращается на нее при нажатой кнопке мыши.

onKillFocus( )

Происходит при потере кнопкой Flash фокуса ввода.

onPress( )

Происходит, когда выполняется нажатие кнопки мыши при нахождении курсора над кнопкой Flash.

onRelease( )

Происходит при нажатии и последующем освобождении кнопки мыши, когда курсор находится над кнопкой Flash.

onReleaseOutside( )

Происходит, когда кнопка мыши нажимается над кнопкой Flash, а затем освобождается после ухода курсора с кнопки.

onRollOut( )

Происходит, когда курсор мыши уводится с кнопки Flash (при ненажатой кнопке мыши).

onRollOver( )

Происходит, когда курсор мыши появляется над кнопкой Flash (при ненажатой кнопке мыши).

onSetFocus( )

Происходит при получении кнопкой Flash фокуса ввода.

Описание

Класс *Button* предоставляет программное управление кнопками фильма. Каждый объект *Button* представляет единственный экземпляр символа Button, созданный вручную на этапе разработки. Кнопки нельзя создавать и уничтожать с помощью ActionScript, несмотря на формальное существование конструктора класса *Button*, в котором содержатся методы и свойства класса. Это означает, что вызов *new Button( )* не создает видимого экземпляра кнопки, которым можно пользоваться в фильме. Однако клипы, а у них есть все возможности, которыми обладают кнопки, можно и создавать, и уничтожать с помощью ActionScript на этапе исполнения. Чтобы удалить экземпляр кнопки, поместите его в клип и удалите клип с помощью *MovieClip.removeMovieClip( )*.

До версии Flash 6 кнопки были недоступны для действий с помощью ActionScript. Во Flash 6 класс *Button* позволяет выполнять с кнопками действия, аналогичные действиям с клипами, – изменять положение, размер, угол поворота и т. д. На самом деле многие свойства кнопок непосредственно заимствованы у класса *MovieClip* (например, `_x`, `_y`, `_width`, `_height` и `_rotation`). Для обеспечения обратной совместимости имена свойств кнопок, соответствующие свойствам клипов, существовавшим до Flash 6, содержат префикс – символ подчеркивания (_). Для лучшей согласованности с другими языками, основанными на ECMA-262, в свойствах, появившихся лишь во Flash 6, символ подчеркивания не употребляется.

Обработчики событий класса *Button* предоставляют средства для обеспечения навигации, создания форм, игр и других элементов интерфейса. Каждый тип событий подробно описывается ниже. Реагировать на события объект кнопки может с помощью обработчиков двух разных типов. Это может быть свойство-обработчик события:

```
go_btn.onPress = function () {
  // инструкции
};
```

либо нотация, характерная для Flash 5, с прикреплением кода непосредственно к кнопке в среде разработки:

```
on (press) {
  // инструкции
}
```

Однако есть одно событие кнопки, *keyPress*, которое поддерживает только формат Flash 5 *on(keyPress)*. См. подробности в статье «Button keyPress, обработчик события».

Общие сведения о создании обработчиков событий см. в главе 10. Обратите внимание, что во Flash 6 все обработчики событий кнопок поддерживаются также классом *MovieClip*.

События кнопок посылаются только той кнопке, с которой взаимодействует мышь. Если на экране перекрывается несколько кнопок, то все события получает самая верхняя из них. Другие кнопки не могут реагировать на события, даже если в самой верхней кнопке не определены никакие обработчики.

В последующих описаниях свойств, методов и событий для *Button* используются следующие понятия:

- *Активная область* (*hit area*) означает физическую область кнопки, которая должна находиться под указателем мыши, чтобы кнопка Flash могла быть активизирована. (Активная область кнопки Flash определяется графически при создании кнопки в среде разработки Flash.)
- Состояния *Up*, *Over* и *Down* соответствуют одноименным кадрам кнопки в среде разработки Flash. Они автоматически отображаются в тех случаях, когда курсор мыши находится вне активной области кнопки Flash (*Up*), над активной областью кнопки Flash (*Over*), или кнопка мыши нажата над активной областью кнопки (*Down*).

Для того чтобы действовала подсказка кода в панели Actions для объекта кнопки, его имя экземпляра должно содержать суффикс «`_btn`», как, например, в `submit_btn`.

Применение

С помощью объектов *Button* даже начинающие разработчики могут обеспечить интерактивность с минимальными усилиями. Однако в более сложных приложениях

кнопки по эффективности уступают клипам. Помимо всех возможностей кнопок клипы предлагают такие функции, как динамическое изменение активной области, переключение состояний кнопки (*Up*, *Over* и *Down*) и полный набор методов, событий и свойств класса *MovieClip*.

См. также Класс *MovieClip*; глава 10

Button._alpha, свойство

Flash 6

степень непрозрачности: 0 – прозрачная; 100 – непрозрачная **чтение/запись**

theButton.`_alpha`

Описание

Свойство `_alpha`[1] имеет тип числа с плавающей точкой и задает степень непрозрачности *theButton* в процентах. При этом 0 соответствует полной прозрачности, а 100 – непрозрачности. Свойство относится ко всем состояниям *theButton* (*Up*, *Over* и *Down*).

Пример

Следующий код задает прозрачность `go_btn` равной 50%.

```
go_btn._alpha = 50;
```

См. также *Button*.`_visible`, *MovieClip*.`_alpha`

Button.enabled, свойство

Flash 6

булево значение, разрешающее или запрещающее взаимодействие с кнопкой **чтение/запись**

theButton.`enabled`

Описание

Булево свойство `enabled` включает кнопку (`true`) или выключает ее (`false`). По умолчанию имеет значение `true`. Если `enabled` имеет значение `false`, *theButton* не принимает данные, вводимые пользователем; курсор «ладонь» не отрисовывается, события не генерируются и показывается только состояние *Up*. Однако даже на выключенную кнопку можно установить фокус ввода с помощью клавиши Tab. Для того чтобы кнопка не получала фокус ввода при нажатии этой клавиши, надо задать для свойства *theButton*.`tabEnabled` значение `false`.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не выполнить команду Control → Disable Keyboard Shortcuts. Следует проверить действие клавиа-

---

[1] Как правило, достаточно знать, что прозрачность задается в процентах. Но если нужны точные числовые значения этого свойства, то следует учитывать, что во Flash всего 256 уровней прозрачности объекта, и значения этих уровней меняются в диапазоне от 0 до 100 в свойстве `_alpha`. Так, после выполнения инструкции `go_btn._alpha = 80` объект `go_btn` получит значение `_alpha`, равное 79,6875%, вместо ожидаемых 80%. Минимальный шаг изменения прозрачности равен 100/256 = 0,390625. – *Примеч. науч. ред.*

туры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

Пример

Кнопки можно отключать с целью управления взаимодействием пользователя с интерфейсом. Например, кнопка формы Submit может быть выключена до того момента, когда пользователь заполнит в форме все поля для ввода текста:

```
// Вначале отключить кнопку Submit
submit_btn.enabled = false;
submit_btn.tabEnabled = false;

// Добавить код, включающий кнопку после заполнения всех полей...
// Например:
mandatoryField.onChanged = function () {
  if (this.text != "") {
    submit_btn.enabled = true;
    submit_btn.tabEnabled = true;
  }
}
```

См. также *Button*.tabEnabled, *Button*.useHandCursor

Button._focusrect, свойство

Flash 6

булево значение, включающее/отключающее автоматическое выделение кнопок, активизируемых с клавиатуры **чтение/запись**

theButton._focusrect

Описание

Когда курсор мыши находится над кнопкой Flash, отображается содержимое ее состояния *Over*. Кнопки могут получать фокус ввода с клавиатуры и в результате нажатия клавиши Tab. В первом случае Flash выводит ее состояние *Over* и заключает его в желтую прямоугольную рамку, которая может выглядеть непривлекательно. Можно отключить отображение рамки для отдельной кнопки, задав для ее свойства _focusrect значение false:

```
go_btn._focusrect = false;
```

Если _focusrect имеет значение false, то Flash только показывает состояние *Over* кнопки, имеющей фокус ввода с клавиатуры. Когда _focusrect имеет значение true, Flash показывает состояние *Over* и желтый прямоугольник. Если это свойство имеет значение null (установленное по умолчанию), то используется значение глобального свойства _focusrect. Отключение вывода желтой рамки для всех кнопок фильма достигается заданием значения false для глобального свойства _focusrect. Учтите, что значение этого глобального свойства не отражается в свойствах _focusrect отдельных экземпляров кнопок. Кроме того, значения свойства _focusrect отдельных кнопок имеют приоритет над значением глобального свойства.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при

воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

См. также `_focusrect`, *MovieClip*`_.focusrect`

Button.getDepth(), метод

Flash 6

возвращает позицию кнопки в стеке содержимого

theButton`.getDepth()`

Возвращаемое значение

Целое число, представляющее уровень глубины.

Описание

Кнопки, клипы и текстовые поля размещаются в проигрывателе в определенном порядке, от которого зависит их расположение друг над другом. Целочисленное свойство глубины задает порядок каждого элемента в этом стеке представления. Метод *getDepth( )* возвращает целое число – глубину расположения *theButton*. На каждом уровне глубины может находиться только один объект, будь то текстовое поле, кнопка или клип. Объекты, глубина которых меньше, отображаются на фоне тех, глубина которых больше.

Поскольку все кнопки создаются на этапе разработки, *Button.getDepth( )* всегда возвращает глубину в диапазоне от -16 383 до 0 (зарезервированном для содержимого, создаваемого вручную). В отличие от класса *MovieClip* у *Button* нет метода *swapDepths( )*, поэтому глубина расположения каждой кнопки остается постоянной после того, как она назначена. Дополнительные сведения о глубине можно найти в разделе «Порядок расположения фильмов и экземпляров в стеке» в главе 13.

Пример

Посмотрим, какая глубина у `cancel_btn`, размещенной в самом нижнем слое главной временной диаграммы:

```
trace(cancel_btn.getDepth());  // Displays: -16383
```

См. также

MovieClip.getDepth( ), *MovieClip.swapDepths( )*, *TextField.getDepth( )*; «Фильмы и порядок расположения экземпляров в стеке» в главе 13

Button._height, свойство

Flash 6

высота кнопки в пикселах **чтение/запись**

theButton`._height`

Описание

Свойство `height` имеет тип числа с плавающей точкой и содержит неотрицательное число, задающее высоту текущего кадра *theButton* (*Up*, *Over*, *Down*) в пикселах. Его можно получать и устанавливать (в последнем случае изменяется размер кнопки на экране). Во всех остальных отношениях оно ведет себя идентично *MovieClip*`._height`; см. подробности в описании последнего для класса *MovieClip*.

См. также *Button*`._width`, *MovieClip*`._height`

Button keyPress, обработчик события

Flash 4

происходит при нажатии заданной клавиши

```
on (keyPress key) {
 инструкции
}
```

Описание

Событие *keyPress* генерируется при нажатии заданной клавиши. Используется синтаксис *on(*eventName*)* в стиле Flash 5. Современный формат в виде функции обратного вызова для него недопустим. Во Flash 6 предпочтительнее работать с клавиатурой через *Key.onKeyUp( )* и *Key.onKeyDown( )*, поскольку этим методам не нужен объект, находящийся на сцене, и они позволяют обнаруживать одновременное нажатие нескольких клавиш.

Для перехвата событий, связанных с клавиатурой, задается клавиша, нажатие которой генерирует событие. При этом используется приведенный выше синтаксис, в котором *key* является строкой, представляющей ассоциируемую с событием клавишу. Строка может быть символом, соответствующим клавише (например, "s" или "S", потому что регистр роли не играет), или ключевым словом, представляющим клавишу в виде *"<Keyword>"*. С каждым обработчиком можно задать только одну клавишу. Для того чтобы с помощью *keyPress* перехватывать несколько клавиш, надо создать несколько обработчиков событий *keyPress*. Например:

```
// Обнаруживает клавишу "a"
on (keyPress "a") {
  trace("The 'a' key was pressed");
}

// Обнаруживает клавишу Enter
on (keyPress "<Enter>") {
  trace("The Enter key was pressed");
}

// Обнаруживает клавишу "стрелка вниз"
on (keyPress "<Down>") {
  trace("The down arrow key was pressed");
}
```

Допустимы следующие значения *Keyword* (отметим, что функциональные клавиши F1...F12 не поддерживаются *keyPress*, но могут обнаруживаться с помощью объекта *Key*):

```
<Backspace>
<Delete>
<Down>
<End>
<Enter>
<Home>
<Insert>
<Left>
<PgDn>
<PgUp>
<Right>
<Space>
<Tab>
<Up>
```

Во Flash 4 событие keyPress – единственный способ взаимодействия с клавиатурой.

Применение

В отличие от всех остальных средств обработки событий в ActionScript, событие кнопки *keyPress* перехватывает режим по умолчанию функционирования клавиатуры в проигрывателе. Например, если кнопка перехватывает нажатие «стрелки влево» через *keyPress*, стрелка влево не будет перемещать курсор в текстовом поле.

```
on ( keyPress "<Left>" ) {
  // Мы просто блокируем клавишу <Left>, но должны поместить здесь какой-нибудь код,
  // иначе оптимизатор удалит эту функцию.
  1 + 1;
}
```

См. также *Key.onKeyDown( )*, *Key.onKeyUp( )*

Button._name, свойство

Flash 6

имя экземпляра объекта Button **чтение/запись**

theButton.`_name`

Описание

Свойство `_name` имеет тип строки и указывает имя экземпляра *theButton*, заданное в инспекторе свойств на этапе разработки. Потребность в этом свойстве если и возникает, то редко, тем не менее присваивание значения свойству `_name` устанавливает для *theButton* новое имя экземпляра. После этого ссылки на прежнее имя экземпляра возвращают `undefined`.

Обычно свойство `_name` используется для чтения, а не для записи. Так, с его помощью можно идентифицировать одну из нескольких кнопок, по которой выполнен щелчок. Следующая функция выполняет различные действия в зависимости от того, какая кнопка ее вызвала:

```
function doAction () {
  if (this._name == "ok_btn") {
    doConfirm();
  } else if (this._name == "cancel_btn") {
    doCancel();
  }
}

// Назначить обработчики для кнопки
ok_btn.onRelease = doAction;
cancel_btn.onRelease = doAction;
```

Для того чтобы найти одну кнопку в группе неизвестных, можно выполнить поиск имени экземпляра кнопки в цикле *for-in*:

```
// Перебор всех свойств someClip
for (var p in someClip) {
  // Если это объект Button ...
  if (someClip[p] instanceof Button) {
    // ...показать имя кнопки:
    trace("Found " + someClip[p]._name);

    // Ищем кнопку edit_btn
    if (someClip[p]._name == "edit_btn") {
      // Отключить кнопку редактирования.
      someClip[p].enabled = false;
    }
```

```
    }
  }
```

См. также *Button.onRelease( )*, «Цикл *for-in*» в главе 8

Button.onDragOut(), обработчик события

Flash 6; on (dragOut) поддерживается с Flash 2

происходит при выводе курсора мыши с кнопки Flash при нажатой кнопке

```
theButton.onDragOut()
```

```
on (dragOut) {
  инструкции
}
```

Описание

Обработчик события *onDragOut( )* выполняется, если нажать левую кнопку мыши во время нахождения курсора мыши в активной области `theButton`, после чего, не отпуская ее, вывести курсор из активной области. Событие аналогично *onRollOut( )* за исключением того, что генерируется только тогда, когда кнопка мыши нажата при выходе курсора из активной области. За событием *onDragOut( )* следует либо событие *onReleaseOutside( )* (если пользователь отпускает кнопку мыши), либо событие *onDragOver( )* (если пользователь перемещает курсор обратно в активную область кнопки, не отпуская кнопку мыши).

Пример

```
cancel_btn.onDragOut = function () {
  // Обработать событие...
  trace("dragOut detected");
};
```

См. также *Button.onDragOver( )*, *Button.onRollOut( )*, *MovieClip.onDragOut( )*; глава 10

Button.onDragOver(), обработчик события

Flash 6; on (dragOver) поддерживается с Flash 2

происходит, когда при нажатой кнопке мыши курсор выходит из активной области и снова в нее возвращается

```
theButton.onDragOver()
```

```
on (dragOver) {
  инструкции
}
```

Описание

Обработчик события *onDragOver( )* – редко встречающийся в природе зверь. Это событие возникает при выполнении такой последовательности действий:

1. Курсор мыши перемещается в активную область кнопки (происходит событие *onRollOver( )*).
2. Нажата и удерживается главная кнопка мыши (происходит событие *onPress( )*).
3. Курсор мыши выходит из активной области кнопки (происходит событие *onDragOut( )*).
4. Курсор мыши возвращается в активную область кнопки (происходит событие *onDragOver( )*).

Таким образом, событие *dragOver* указывает на то, что пользователь вывел курсор мыши из активной области и ввел его обратно, при этом кнопка мыши была все время нажата. Обратите внимание, что если кнопка мыши остается нажатой при повторном входе курсора в активную область кнопки, то генерируется событие *dragOver*, а не *rollOver*.

Пример

```
cancel_btn.onDragOver = function () {
  // Обработать событие...
  trace("dragOver detected");
};
```

См. также

Button.onDragOut( ), *Button.onRollOver( )*, *MovieClip.onDragOver( )*; глава 10

Button.onKillFocus(), обработчик события

Flash 6

функция обратного вызова, выполняемая при потере кнопкой фокуса ввода

theButton.onKillFocus(*newFocus*)

Аргументы

newFocus

Объект *TextField*, *MovieClip* или *Button*, получивший фокус, либо `null`, если никакой объект не получил фокус.

Описание

Обработчик события *onKillFocus( )* выполняется, когда *theButton* теряет фокус. Кнопка «имеет фокус», если:

- Пользователь перемещается на нее с помощью клавиши Tab.
- *Selection.setFocus( )* помещает кнопку в фокус программным образом.

Когда `_focusrect` имеет значение `true` или `null` (значение по умолчанию), вокруг находящейся в фокусе кнопки появляется желтый прямоугольник. Кнопка теряет фокус в следующих случаях:

- Пользователь переходит к другой кнопке, текстовому полю или клипу с помощью клавиши Tab.
- *Selection.setFocus( )* программно перемещает фокус на другой объект. (Доступ к новому объекту, находящемуся в фокусе, в обработчике *onKillFocus( )* осуществляется через `newFocus`.)
- Пользователь перемещает курсор мыши.

В первых двух случаях выполняется *onKillFocus( )*. В последнем случае из-за ошибки во Flash Player 6 *onKillFocus( )* не выполняется, хотя должен. Если свойство `tabEnabled` кнопки имеет значение `false`, пользователь не может переместить фокус на кнопку, но это можно сделать программно с помощью *Selection.setFocus( )*. Для того чтобы перехватывать все события фокуса централизованно, а не для каждого экземпляра, применяется обработчик *Selection.onSetFocus( )*.

Пример

В следующем примере вводятся функции обратного вызова – обработчики событий *onSetFocus( )* и *onKillFocus( )* для кнопки `cancel_btn`. Кроме того создается текстовое

поле, в котором отображается состояние фокуса. Наконец, фокус устанавливается на cancel_btn. Чтобы проверить работу этого кода, создайте две кнопки, одна из которых будет называться «cancel_btn». Затем запустите фильм в автономном проигрывателе и нажмите клавишу Tab (*не касаясь мыши!*). Если тестирование происходит в броузере, то перед нажатием Tab надо щелкнуть по области воспроизведения фильма.

```
this.createTextField("status_txt", 1, 0, 0, 300, 20);
status_txt.border = true;

cancel_btn.onKillFocus = function () {
  // Обработать событие...
  status_txt.text = ("killfocus detected for cancel_btn");
};

cancel_btn.onSetFocus = function () {
  // Обработать событие...
  status_txt.text = ("setfocus detected for cancel_btn");
};

Selection.setFocus(cancel_btn);
```

См. также

Button.onSetFocus( ), *Selection.onSetFocus( )*, *Selection.setFocus( )*, *MovieClip.onKillFocus( )*, *TextField.onKillFocus( )*; глава 10

Button.onPress(), обработчик события

Flash 6; on (press) поддерживается с Flash 2

происходит при щелчке кнопкой мыши, когда курсор находится над кнопкой Flash

```
theButton.onPress()
```

```
on (press) {
  инструкции
}
```

Описание

Обработчик события *onPress( )* выполняется, если нажать главную кнопку мыши во время нахождения курсора мыши в активной области. Нажатие других кнопок мыши (правой, средней, скроллинга и т. д.) не обнаруживается. Обратите внимание на двухступенчатость «щелчка кнопкой мыши»: кнопка мыши сначала нажимается (*press*), а затем освобождается (*release*). События нажатия кнопки подходят для работы переключателей или стрельбы в играх, но следует дать возможность пользователю изменить свое решение до того, как кнопка будет освобождена, использовав для этого событие *release*.

Пример

```
cancel_btn.onPress = function () {
  // Обработать событие...
  trace("press detected");
};
```

См. также

Button.onRelease( ), *MovieClip.onPress( )*; глава 10

Button.onRelease(), обработчик события

Flash 6; on (release) поддерживается с Flash 2

происходит при нажатии и последующем освобождении кнопки мыши во время нахождения курсора над кнопкой Flash

```
theButton.onRelease()
```

```
on (release) {
  инструкции
}
```

Описание

Обработчик события *onRelease( )* выполняется, если обнаружена такая последовательность событий:

1. Курсор мыши помещен в активную область кнопки `theButton`.
2. Главная кнопка мыши нажата во время нахождения курсора мыши в активной области `theButton` (в этот момент происходит событие *onPress( )*).
3. Главная кнопка мыши освобождается, когда курсор мыши по-прежнему находится в активной области `theButton` (в этот момент происходит событие *onRelease( )*).

Применяя в логике обработки событие освобождения вместо события нажатия, можно дать пользователю возможность убрать курсор с кнопки после нажатия кнопки мыши и тем самым отменить свое действие. Обработчик *onRelease( )* применяется чаще других обработчиков событий кнопок во Flash. С его помощью программируется реакция на классический щелчок кнопкой мыши. Обратите внимание, что свойство `trackAsMenu` может изменить поведение *onRelease( )*, как описывается в `Button.trackAsMenu`.

Пример

В следующем коде приводятся два обработчика *onRelease( )*. При щелчке по `ok_btn` головка воспроизведения родительского клипа кнопки направляется в кадр с меткой `begin`. При щелчке по `cancel_btn` головка воспроизведения направляется в кадр с меткой `quit`.

```
ok_btn.onRelease = function () {
  this._parent.gotoAndStop("begin");
}

cancel_btn.onRelease = function () {
  this._parent.gotoAndStop("quit");
}
```

См. также *Button.onPress( )*, *Button*.`trackAsMenu`, *MovieClip.onRelease( )*; глава 10

Button.onReleaseOutside(), обработчик события

Flash 6; on(releaseOutside) поддерживается с Flash 2

происходит при щелчке над кнопкой с последующим ее освобождением после увода курсора с кнопки

```
theButton.onReleaseOutside()
```

```
on (releaseOutside) {
  инструкции
}
```

Описание

Обычно событие *onReleaseOutside( )* указывает на то, что пользователь изменил свое решение после щелчка по кнопке, уведя курсор в сторону перед освобождением кнопки мыши. Обработчик события *onReleaseOutside( )* выполняется при обнаружении следующей последовательности событий:

1. Курсор мыши находится над активной областью `theButton`.
2. Левая кнопка мыши нажата и удерживается (происходит событие *onPress( )*).
3. Курсор мыши выводится из активной области `theButton` (происходит событие *onDragOut( )*).
4. Левая кнопка мыши освобождается, когда курсор больше не находится в активной области `theButton`.

Обнаружение событий *onReleaseOutside( )* требуется редко, поскольку они означают, что пользователь решил не выполнять никаких действий.

Пример

```
cancel_btn.onReleaseOutside = function () {
  trace("releaseOutside detected");
}
```

См. также *Button.onRelease( )*, *MovieClip.onReleaseOutside( )*; глава 10

Button.onRollOut(), обработчик события

Flash 6; on(rollOut) поддерживается с Flash 2

происходит при уходе курсора с кнопки (при ненажатой кнопке мыши)

```
theButton.onRollOut()
```

```
on (rollOut) {
  инструкции
}
```

Описание

Обработчик события *onRollOut( )* составляет пару обработчику *onRollOver( )*; он выполняется, когда курсор мыши уводится из активной области кнопки при ненажатой кнопке мыши. Как и *onRollOver( )*, обработчик *onRollOut( )* обычно не применяется для показа выделенного состояния кнопки, поскольку изменение внешнего вида кнопки задается непосредственно в среде разработки. Создавать выделенные состояния кнопок следует при помощи соответственно названных кадров *Up*, *Over* и *Down* на временной диаграмме кнопки.

Пример

```
cancel_btn.onRollOut = function () {
  trace("rollOut detected");
};
```

См. также

Button.onDragOut( ), *Button.onRollOver( )*, *MovieClip.onRollOut( )*; глава 10

Button.onRollOver(), обработчик события

Flash 6; on(rollOver) поддерживается с Flash 2

происходит при появлении курсора над кнопкой (при ненажатой кнопке мыши) или при получении кнопкой фокуса ввода с клавиатуры

```
theButton.onRollOver()
```

```
on (rollOver) {
  инструкции
}
```

Описание

Обработчик события *onRollOver( )* выполняется, когда курсор входит в активную область кнопки и кнопка мыши при этом не нажата. Он также выполняется, когда `theButton` получает фокус ввода с клавиатуры, как описано в статье, посвященной обработчику *Button.onSetFocus( )*. Событие *rollOver* не требуется для отображения видимого состояния кнопки, поскольку изменение внешнего вида кнопки задается непосредственно в среде разработки, а не с помощью сценариев. Создавать выделенные состояния кнопок следует при помощи соответственно названных кадров *Up*, *Over* и *Down* на временной диаграмме кнопки.

Событие *rollOver* предоставляет удобный способ получения текста, выделенного в текстовом поле. Подробнее см. в описании *TextField.replaceSel( )* и объекта *Selection*.

Пример

```
cancel_btn.onRollOver = function () {
  trace("rollOver detected");
};
```

См. также *Button.onDragOut( )*, *Button.onRollOut( )*, *Button.onSetFocus( )*, *MovieClip.onRollOver( )*; глава 10

Button.onSetFocus(), обработчик события

Flash 6

функция обратного вызова, выполняемая при получении кнопкой фокуса ввода

```
theButton.onSetFocus(oldFocus)
```

Аргументы

oldFocus

Объект *TextField*, *MovieClip* или *Button*, ранее обладавший фокусом, либо `null`, если фокус не принадлежал ранее ни одному объекту.

Описание

Обработчик события *onSetFocus( )* выполняется, когда `theButton` получает фокус. Кнопка получает фокус в следующих случаях:

- Пользователь перемещается на нее с помощью клавиши Tab.
- *Selection.setFocus( )* помещает кнопку в фокус программно.

Если свойство кнопки `tabEnabled` имеет значение `false`, то пользователь не может переместить фокус на кнопку, но это можно сделать программно с помощью *Selection.setFocus( )*. В следующем коде регистрируется функция обратного вызова для ответа на *onSetFocus( )*:

```
// Назначить функцию обратного вызова для onSetFocus() с помощью литерала
next_btn.onSetFocus = function (oldFocus) {
  trace(this + " now has focus. Old focus was " + oldFocus);
};
```

Обратите внимание, что в теле функции обратного вызова можно обращаться к next_btn с помощью ключевого слова this. Чтобы прекратить обработку события *onSetFocus( )* функцией обратного вызова, можно воспользоваться оператором *delete*:

```
// Следите, чтобы после имени функции не было оператора ()!
delete next_btn.onSetFocus;
```

Обнаружить получение фокуса независимо от того, произошло оно в результате действий с мышью или с клавиатурой, позволяет обработчик *onRollOver( )*, выполняемый в обоих случаях – при нахождении курсора мыши над *theButton* и при получении *theButton* фокуса от клавиатуры. Для того чтобы перехватывать все события фокуса централизованно, а не для каждого экземпляра, применяется обработчик *Selection.onSetFocus( )*.

Для активации кнопки, имеющей фокус ввода с клавиатуры, пользователь должен нажать клавишу Enter или пробел (это приводит к такому же результату, что и щелчок по кнопке, и заставляет выполниться обработчик *onRelease( )*).

См. также *Button.onKillFocus( )*, *Button.onRollOver( )*, *MovieClip.onSetFocus( )*, *Selection.onSetFocus( )*, *Selection.setFocus( )*, *TextField.onSetFocus( )*; глава 10

Button._parent, свойство

Flash 6

ссылка на клип, в котором находится кнопка **чтение/запись**

theButton._parent

Описание

Свойство _parent представляет собой объектную ссылку на клип, содержащий *theButton*. Так, если есть кнопка с именем go_btn в клипе с именем mainSection_mc, то go_btn._parent – это ссылка на mainSection_mc.

Применение

Существует возможность присвоить в качестве значения свойства _parent другой клип (или любое другое значение), но в этом нет особого смысла – изменится только ссылка, а не физическая структура клипа и кнопок.

Пример

Свойство _parent часто применяется для управления клипом из обработчика события кнопки в зависимости от действий пользователя. Например, в следующем коде при щелчке по begin_btn начинается воспроизведение фильма, в котором находится кнопка:

```
begin_btn.onRelease = function () {
  this._parent.play();
}
```

Обратите внимание, что если прикреплять обработчики событий непосредственно к объектам (например, *on (press)*) в стиле Flash 5, то ссылка _parent не нужна, поскольку this ссылается на клип, содержащий объект. Сведения об области видимости обработчиков событий кнопок см. в главе 10.

См. также *Button.onRelease( )*, *MovieClip*._parent; глава 10

Button._rotation, свойство

Flash 6

угол поворота кнопки в градусах **чтение/запись**

theButton._rotation

Описание

Свойство _rotation имеет тип числа с плавающей точкой и задает угол поворота *theButton* относительно точки регистрации в градусах. В _rotation отражаются изменения, внесенные как на стадии разработки, так и на этапе исполнения. Подробные сведения о положительных и отрицательных, а также выходящих за допустимые пределы значениях угла поворота можно найти в описании *MovieClip*._rotation.

Пример

Следующий код поворачивает кнопку при щелчке по ней:

```
fun_btn.onRelease = function () {
  this._rotation += 10;
}
```

См. также *Button.onRelease( )*, *MovieClip*._rotation

Button.tabEnabled, свойство

Flash 6

булево значение; включает или выключает кнопку из обхода при нажатии клавиши Tab **чтение/запись**

theButton.tabEnabled

Описание

Булево свойство tabEnabled управляет вхождением *theButton* в порядок обхода текстовых полей, клипов и кнопок, находящихся в данный момент на экране, при нажатии клавиши табуляции. Порядок обхода (*tab order*) – это последовательность, в которой объекты в фильме получают фокус при нажатии клавиши Tab. Подробности см. в описании *TextField*.tabEnabled.

Применительно к кнопкам свойство tabEnabled может исключить кнопку с указанным tabIndex из пользовательского порядка обхода либо исключить любую кнопку из автоматического порядка обхода. Обычно кнопка изымается из обхода по клавише Tab, если она выключена (ее свойство enabled имеет значение false), либо если она должна быть доступна только для действий мыши, например в играх, когда прозрачные кнопки действуют как активные области мыши.

Свойство клипа tabChildren управляет включением содержащихся в клипе текстовых полей, клипов и кнопок в автоматический порядок обхода. Быстрый способ отключить обход по клавише Tab для группы кнопок состоит в том, чтобы поместить их в клип и задать для его свойства tabChildren значение false.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при

воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

Пример

Следующий код закрывает доступ с клавиатуры к `done_btn`:

```
done_btn.tabEnabled = false;
```

См. также *Button*.enabled, *Button*.tabIndex, *MovieClip*.tabChildren, *MovieClip*.tabEnabled, *TextField*.tabEnabled

Button.tabIndex, свойство

Flash 6

задает индекс кнопки в пользовательском порядке обхода при табуляции **чтение/запись**

theButton.tabIndex

Описание

Свойство `tabIndex` содержит целое число, задающее позицию *theButton* в пользовательском порядке обхода по клавише Tab. Если нет текстовых полей, клипов или кнопок, для которых задано свойство `tabIndex`, Flash определяет порядок обхода автоматически, и при нажатии клавиши Tab происходит перемещение фокуса между кнопками и текстовыми полями слева направо, затем сверху вниз. С помощью свойства `tabIndex` можно задать перемещение фокуса по клавише Tab в специальном порядке, отменяющем автоматический. Подробности см. в описании *TextField*.tabIndex.

Объекты со свойством `_visible`, равным `false`, не включаются в порядок обхода. Если хотя бы для одного объекта установлено свойство `tabIndex`, то ко всем объектам, у которых свойство `tabIndex` не установлено, нельзя получить доступ по клавише Tab.

По умолчанию кнопки включаются в порядок обхода, поэтому свойство `tabEnabled` требуется установить, только чтобы исключить кнопку из обхода по табуляции, сохранив при этом ее значение `tabIndex` (что позволяет с минимальными усилиями выключать и снова включать части интерфейса, управляемые с клавиатуры).

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

Пример

Следующий код ограничивает доступ по клавише Tab к следующим кнопкам в указанном порядке: `jump_btn`, `moveLeft_btn`, `moveRight_btn`. Порядок действует независимо от расположения кнопок на экране.

```
jump_btn.tabIndex = 1;
moveLeft_btn.tabIndex = 2;
moveRight_btn.tabIndex = 3;
```

См. также *Button*.tabEnabled, *TextField*.tabIndex, *MovieClip*.tabIndex

Button._target, свойство

Flash 6

путь к кнопке в слэш-синтаксисе **чтение/запись**

theButton._target

Описание

Свойство `_target` представляет путь к *theButton* в нотации Flash 4, использующей слэши. Например, если `go_btn` расположена на главной временной диаграмме фильма, то свойство `_target` для `go_btn` имеет значение «`/go_btn`». Для получения строки, содержащей путь к кнопке в нотации с точкой, предназначена глобальная функция *targetPath( )*. Нотация со слэшами считается устаревшей, поэтому данное свойство применяется редко и поддерживается только для совместимости со старым свойством *MovieClip.*`_target`.

См. также *targetPath( )*, *MovieClip.*`_target`

Button.trackAsMenu, свойство

Flash 3; в виде свойства появилось во Flash 6

модифицирует требования к обработчику onRelease() события кнопки, включая режим показа меню — **чтение/запись**

theButton.`trackAsMenu`

Описание

Булево свойство `trackAsMenu` модифицирует условия выполнения обработчика события кнопки `theButton` *onRelease( )*. Обычно применяется для превращения группы кнопок в меню. Если `trackAsMenu` имеет значение `undefined` (по умолчанию) или `false`, то обработчик события кнопки *onRelease( )* будет выполняться, только если кнопка мыши нажата *и* отпущена над кнопкой. Если значение `trackAsMenu` равно `true`, то освобождение кнопки мыши над кнопкой Flash всегда запускает обработчик *onRelease( )*, если только первоначально кнопка мыши не была нажата над кнопкой, свойство `trackAsMenu` которой имеет значение `false` или `undefined`. Следующие сценарии демонстрируют результат этого тонкого различия в поведении.

Рассмотрим кнопку меню «Products» с тремя кнопками подвариантов: Music, Books и Software. Следующая последовательность действий показывает, как работает меню, когда `trackAsMenu` имеет значение `false` во всех четырех кнопках:

1. Нажата кнопка Products.
2. Появляются кнопки подвариантов (прикрепленные на этапе исполнения).
3. Мышь перемещается на первый вариант, Music. Отображается состояние *Up* кнопки Music.
4. Кнопка мыши отпускается над Music. Обработчик *onRelease( )* не выполняется, поскольку кнопка мыши была нажата над Products, но отпущена над другой кнопкой – Music. Выбранный пользователем вариант не активизируется.

Нижеприведенная последовательность действий показывает, как работает меню, когда `trackAsMenu` имеет значение `true` во всех четырех кнопках:

1. Нажата кнопка Products.
2. Появляются кнопки подвариантов (прикрепленные на этапе исполнения).
3. Мышь перемещается на первый вариант, Music. Показывается состояние *Down* кнопки Music, что соответствует текущему выбору пользователя.
4. Кнопка мыши отпускается над Music. Выполняется обработчик *onRelease( )*. Активизируется выбранный пользователем вариант.

Обратите внимание, что задание `true` для `trackAsMenu` имеет побочный эффект – всегда показывается состояние кнопки *Down*, когда курсор мыши находится над кнопкой и кнопка мыши нажата (даже если первоначально кнопка мыши была нажата над другой кнопкой). Однако обработчики кнопки *onRollOver( )* и *onRollOut( )* не запускаются, пока не будет отпущена кнопка мыши.

Свойство `trackAsMenu` должно иметь одинаковое значение для всех кнопок в меню или другой группе элементов интерфейса; в противном случае могут возникнуть неприятности и путаница.

Пример

В кадре 1 фильма создайте три экземпляра кнопки с именами `b1`, `b2` и `b3`, каждый с уникальными состояниями *Up*, *Over* и *Down*. Затем прикрепите к кадру 1 приведенный ниже код и выберите пункт меню Control → Test Movie. Нажмите кнопку мыши над `b1` и отпустите ее над `b2`. Обратите внимание, что отображается состояние кнопки `b2` Down и выполняется ее обработчик *onRelease( )*.

```
b1.trackAsMenu = true;
b2.trackAsMenu = true;
b3.trackAsMenu = true;

b1.onRelease = b2.onRelease = b3.onRelease = release;

function release () {
  trace(this._name + " released");
}
```

См. также *Button*.onRelease, *Button*.onRollOver, *Button*.onRollOut

Button._url, свойство

Flash 6

сетевой адрес клипа, содержащего кнопку **чтение/запись**

theButton._url

Описание

Свойство `_url` представляет URL (Uniform Resource Locator) и является строкой с адресом *theButton*._parent в Интернете или на локальном диске. Свойство_url всегда содержит абсолютный URL, например «*http://www.moock.org/asdg/examples/someMovie.swf*», а не относительный. Подробности см. в описании *MovieClip*._url. Обратите внимание, что *theButton*._url всегда имеет такое же значение, как *theButton*._parent._url.

См. также *Button*._parent, *MovieClip*._url

Button.useHandCursor, свойство

Flash 6

булево значение; определяет, показывать ли значок ладони при нахождении курсора мыши над кнопкой **чтение/запись**

theButton.useHandCursor

Описание

Если значение булева свойства `useHandCursor` равно `false`, то отключается вывод значка курсора в виде ладони при нахождении курсора мыши над активной областью

theButton, хотя обычно этот значок отображается. Место курсора в виде ладони занимает обычный курсор – стрелка (это типичное поведение для кнопок операционной системы). В фильмах с большими прозрачными кнопками в таких интерфейсах, как слайд-шоу картинок, курсор в виде ладони часто отключается.

Обратите внимание, что, кроме значка ладони, других встроенных курсоров мыши нет. Рекомендации по созданию пользовательских значков курсора мыши см. в описании *Mouse.hide( )*.

Пример

В следующем примере отключается курсор в виде ладони для `ok_btn`:

```
ok_btn.useHandCursor = false;
```

В следующем примере отключается курсор в виде ладони для всех кнопок в фильме (если только глобальная установка не будет переопределена в кнопке с помощью ее собственного свойства `useHandCursor`):

```
Button.prototype.useHandCursor = false;
```

См. также *Mouse.hide( )*; «Всемогущая цепочка прототипов» в главе 12

Button._visible, свойство

Flash 6

определяет, показывается ли кнопка **чтение/запись**

theButton.`_visible`

Описание

Булево свойство `_visible` определяет, видна (`true`) или скрыта (`false`) кнопка *theButton*. С помощью `_visible` можно быстро скрыть кнопку. Скрытыми кнопками можно по-прежнему управлять через ActionScript – они лишь не отображаются на экране. Однако скрытая кнопка не может получить фокус или данные ввода пользователя, поэтому ее события не генерируются.

Свойство *Button*.`_visible` вполне аналогично *MovieClip*.`_visible`; все подробности см. в описании последнего свойства.

Пример

В следующем коде делается невидимой кнопка `quit_btn`:

```
quit_btn._visible = false;
```

См. также *Button*.`_alpha`, *MovieClip*.`_visible`

Button._width, свойство

Flash 6

ширина кнопки в пикселах **чтение/запись**

theButton.`_width`

Описание

Свойство `_width` имеет тип числа с плавающей точкой и содержит неотрицательное число, задающее текущую высоту *theButton* в пикселах. Если у *theButton* нет содержимого, то `_width` имеет значение 0. Свойство `_width` измеряет содержимое кнопки как расстояние между самым левым и самым правым занятыми пикселами, даже если между ними есть пустое пространство. Присваивая значение свойству `_width`, мы из-

меняем размер *theButton*, что оказывает воздействие на все ее состояния (ее кадры *Up*, *Over* и *Down*). Обратите внимание, что значение _width кнопки может изменяться при переходе из одного из этих состояний в другое.

Свойство *Button*._width вполне аналогично *MovieClip*._width; все подробности см. в описании последнего свойства.

Пример

```
// Получим ширину start_btn
trace(start_btn._width);

// Зададим ширину start_btn равной 200
start_btn._width = 200;

// Создадим текстовое поле в 10 пикселах вправо от start_btn, предполагая, что
// левый верхний угол start_btn находится в точке регистрации
start_btn._parent.createTextField("theField_txt",
                                  1,
                                  start_btn._x + start_btn._width + 10,
                                  start_btn._y,
                                  200, 20);
theField_txt.border = true;
theField_txt.text = "Click the button to start.";
```

См. также *Button*._height, *MovieClip*._width

Button._x, свойство

Flash 6

положение кнопки по горизонтали в пикселах **чтение/запись**

theButton._x

Описание

Свойство _x имеет тип числа с плавающей точкой и задает положение по горизонтали точки регистрации *theButton*. Измерение может осуществляться в одном из двух координатных пространств:

- Если *theButton* расположена на главной временной диаграмме, _x измеряется относительно *левого края сцены*.
- Если *theButton* расположена на временной диаграмме экземпляра клипа, _x измеряется относительно *точки регистрации* этого родительского экземпляра.

Свойство _x (как и все горизонтальные координаты во Flash) возрастает вправо и уменьшается влево.

Свойство *Button*._x вполне аналогично *MovieClip*._x; все подробности см. в описании последнего свойства.

Пример

```
// Поместить quit_btn на 100 пикселов правее точки регистрации ее родительского объекта
quit_btn._x = 100;
// Передвинуть кнопку еще на 10 пикселов вправо
quit_btn._x += 10;
```

См. также *Button*._y, *MovieClip*._x

Button._xmouse, свойство

Flash 6

положение курсора мыши по горизонтали относительно точки регистрации кнопки **только чтение**

theButton.`_xmouse`

Описание

Свойство `_xmouse` имеет тип числа с плавающей точкой и задает положение по горизонтали активной точки курсора мыши относительно точки регистрации *theButton*. Для получения согласованной координаты `_xmouse`, которая всегда измеряется относительно сцены, следует применять `_root._xmouse`.

См. также *Button*.`_ymouse`, *MovieClip*.`_xmouse`

Button._xscale, свойство

Flash 6

ширина кнопки в процентном отношении **чтение/запись**

theButton.`_xscale`

Описание

Свойство`_xscale` имеет тип числа с плавающей точкой и задает процентное отношение ширины *theButton* к ее первоначальной ширине. «Начальная ширина» – это ширина библиотечного символа кнопки в пикселах. Когда текущая ширина *theButton* совпадает с исходной, значение `_xscale` равняется 100. Значение `_xscale`, равное 200, удваивает первоначальную ширину *theButton*; значение `_xscale`, равное 50, устанавливает половину первоначальной ширины.

Свойство *Button*.`_xscale` вполне аналогично *MovieClip*.`_xscale`; все подробности см. в описании последнего свойства.

Пример

```
// Удвоить ширину run_btn по отношению к ширине ее библиотечного символа
run_btn._xscale = 200;
```

См. также *Button*.`_width`, *Button*.`_yscale`, *MovieClip*.`_xscale`

Button._y, свойство

Flash 6

положение кнопки по вертикали в пикселах **чтение/запись**

theButton.`_y`

Описание

Свойство `_y` имеет тип числа с плавающей точкой и задает положение по горизонтали точки регистрации *theButton*. Измерение может осуществляться в одном из двух координатных пространств:

- Если *theButton* расположена на главной временной диаграмме, `_y` измеряется относительно *верхнего края сцены*.
- Если *theButton* расположена на временной диаграмме экземпляра клипа, `_y` измеряется относительно *точки регистрации* этого родительского экземпляра.

Свойство _y (как и все вертикальные координаты во Flash) возрастает вниз и уменьшается вверх – в противоположность декартовой системе координат.

*Button.*_y аналогично *MovieClip.*_y; все подробности см. в описании последнего свойства.

Пример

```
// Поместить quit_btn прямо в точку регистрации ее родительского объекта.
quit_btn._x = 0;
quit_btn._y = 0;
```

См. также *Button.*_x, *MovieClip.*_y

Button._ymouse, свойство

Flash 6

положение курсора мыши по вертикали относительно точки регистрации кнопки **только чтение**

*theButton.*_ymouse

Описание

Свойство _ymouse имеет тип числа с плавающей точкой и задает положение по горизонтали активной точки курсора мыши относительно точки регистрации *theButton*. Для получения согласованной координаты _ymouse, которая всегда измеряется относительно сцены, следует применять _root._ymouse.

См. также *Button.*_xmouse, *MovieClip.*_ymouse

Button._yscale, свойство

Flash 6

высота кнопки в процентном отношении **чтение/запись**

*theButton.*_yscale

Описание

Свойство_yscale имеет тип числа с плавающей точкой и задает процентное отношение высоты *theButton* к ее первоначальной высоте. «Начальная высота» – эта высота библиотечного символа кнопки в пикселах. Когда текущая высота *theButton* совпадает с исходной, значение _yscale равняется 100. Значение _yscale, равное 200, удваивает первоначальную высоту *theButton*; значение _yscale, равное 50, устанавливает половину первоначальной высоты.

Свойство *Button.*_yscale аналогично *MovieClip.*_yscale; все подробности см. в описании последнего свойства.

Пример

```
// Восстановить исходные (символа библиотеки) размеры sort_btn
sort_btn._yscale = 100;
sort_btn._xscale = 100;
```

См. также *Button.*_height, *Button.*_xscale, *MovieClip.*_yscale

call(), глобальная функция

Flash 4

выполняет сценарий удаленного кадра

call(*frameLabel*)
call(*frameNumber*)

Аргументы

frameLabel

Строка, содержащая метку кадра, сценарий которого должен быть выполнен.

frameNumber

Номер кадра, сценарий которого должен быть выполнен.

Описание

Функция *call( )* выполняет сценарий, прикрепленный к кадру с меткой `frameLabel` или номером `frameNumber`. Так, следующий код выполняет сценарий кадра 20 текущей временной диаграммы:

```
call(20);
```

Во Flash 4 *call( )* применялась для создания грубого подобия многократно используемой подпрограммы (не способной принимать параметры и возвращать значение). Начиная с Flash 5 предпочтительным способом определения функции является инструкция *function*.

Обратите внимание, что в версии Flash 5 и выше при удаленном вызове сценария через *call( )* все переменные, объявленные посредством ключевого слова *var*, считаются локальными для этого выполнения сценария и прекращают свое существование по завершении сценария. Для того чтобы создать нелокальные переменные в удаленно выполняемом сценарии, опустите ключевое слово *var*:

```
var x = 10;  // Локальная переменная; удаляется после выполнения сценария
x = 10;      // Переменная временной диаграммы; сохраняется по завершении
```

Для вызова *call( )* с кадрами вне текущей временной диаграммы воспользуйтесь устаревшей функцией *tellTarget( )*. В следующем примере выполняется сценарий в кадре 10 клипа `box`:

```
tellTarget ("box") {
  call(10);
}
```

См. также Глава 9; приложение C

Capabilities, объект

Flash 6

информация о Flash Player и системе, в которой он работает

Свойства

hasAccessibility

Указывает, поддерживает ли Player облегченный доступ для пользователей с ослабленным зрением (MSAA).

hasAudio

Указывает, поддерживает ли Player воспроизведение звука.

hasAudioEncoder

Указывает, поддерживает ли система кодировку звука.

hasMP3

Указывает, поддерживает ли Player воспроизведение MP3.

hasVideoEncoder

Указывает, поддерживает ли система кодировку видео.

input

Системное устройство ввода данных пользователем, например мышь (задается словом «point»).

isDebugger

Указывает, является ли версия Player отладочной.

language

Текущий язык, установленный в операционной системе.

manufacturer

Изготовитель Player.

os Операционная система, в которой выполняется Player.

pixelAspectRatio

Отношение ширины экранного пиксела к его высоте.

screenColor

Режим цвета, поддерживаемый экраном (цветной, оттенки серого, черно-белый).

screenDPI

Количество пикселов в одном дюйме экрана.

screenResolutionX

Ширина экрана в пикселах.

screenResolutionY

Высота экрана в пикселах.

serverString

Значения всех свойств *Capabilities* в виде строки переменных в кодировке URL.

version

Версия Flash Player.

Описание

Объект *Capabilities* представляет технические характеристики Flash Player и системы, в которой он выполняется. Он хранится в виде свойства объекта *System* (то есть `System.capabilities`). Следовательно, ссылки на свойства *Capabilities* всегда начинаются с объекта *System*, например:

```
System.capabilities.propertyName
```

Например, следующий код выводит в окно Output ширину экрана:

```
trace(System.capabilities.screenResolutionX);
```

Объект *Capabilities* используется при написании кода, адаптирующего фильм к среде, в которой он воспроизводится. Он применяется преимущественно для мобильных устройств, таких как Pocket PC и сотовые телефоны. Например, запросив значение `hasAudio`, фильм, воспроизводящийся в наладонном устройстве, может узнать, поддерживает ли оно воспроизведение звука. Если `hasAudio` имеет значение `false`, фильм может показать текстовые субтитры.

У всех свойств *Capabilities* есть имена и значения сокращенного формата, воспользовавшись которыми, можно сообщить сценарию, выполняющемуся на сервере, какие возможности поддерживает проигрыватель клиента. Так, сокращенной формой свойства `screenColor` является `COL` с возможными значениями «color», «gray» и «bw». Свойство `capabilities.serverString` объединяет все пары имя/значение в сокращенном формате в строку в кодировке URL, которую можно передать на сервер. Прочтя эту информацию, сервер может отправить в ответ фильм или данные, соответствующие среде воспроизведения. Подробности см. в описании `capabilities.serverString`.

Информацию о разработке фильмов Flash для специальных устройств можно найти по следующим адресам:

http://www.macromedia.com/software/flashplayer/resources/devices/
http://www.flashenabled.com/mobile/

Список устройств, на которых может выполняться Flash Player, есть на странице:

http://www.macromedia.com/software/flashplayer/resources/devices/supported_devices/

Применение

Хотя свойства *Capabilities* доступны для записи, практического смысла изменять значения, присвоенные им системой, нет. Тем самым была бы скрыта содержащаяся в них информация. Присваивания значений любым свойствам, кроме `serverString`, не отражаются на значении `serverString`.

См. также Объект *System*, объект *Stage*

capabilities.hasAccessibility, свойство

Flash 6

указывает, поддерживает ли Player Microsoft Active Accessibility (MSAA) **чтение/запись**

`System.capabilities.hasAccessibility`

Описание

Во Flash Player 6 булево свойство `hasAccessibility` имеет значение `true`, если Player поддерживает облегченный доступ (accessibility) для людей с нарушениями зрения. На момент написания данной книги только ActiveX-версия Flash Player 6 для Windows поддерживала облегченный доступ (через Microsoft Active Accessibility, MSAA). MSAA – это API от Microsoft, предоставляющий элементы интерфейса пользователя в приложении средствам для чтения экрана и другим вспомогательным программам, таким как Window-Eyes GW Micro (*http://www.gwmicro.com*).

Фильмы, взаимодействующие со вспомогательными средствами доступа, должны убедиться, что как `hasAccessibility`, так и *Accessibility.isActive( )* возвращают `true`, прежде чем посылать какую-либо информацию об облегченном доступе. Свойство `hasAccessibility` сообщает о том, что Player поддерживает облегченный доступ, а метод *Accessibility.isActive( )* сообщает о том, что «читалка» экрана фактически работает и поддерживает связь с Flash Player.

Дополнительные сведения о MSAA есть по адресу:

http://msdn.microsoft.com/library/en-us/msaa/msaastart_9w2t.asp

Подробнее облегченный доступ рассмотрен в описании объекта *Accessibility* .

Соответствующей строкой сервера для hasAccessibility является ACC, с возможными значениями t (true) и f (false). Подробности см. в описании capabilities.serverString.

См. также Объект *Accessibility*

capabilities.hasAudio, свойство

Flash 6

указывает, поддерживает ли Player воспроизведение звука **чтение/запись**

System.capabilities.hasAudio

Описание

Булево свойство hasAudio имеет значение true для Flash Player, поддерживающих воспроизведение звука, и значение false для проигрывателей, не способных воспроизводить звук. Звук поддерживается всеми настольными версиями Flash, но не всеми версиями для наладонных устройств и приставок. Основываясь на значении этого свойства, фильмы могут предоставлять альтернативный контент, например выводить субтитры, если воспроизведение звука невозможно.

Обратите внимание, что hasAudio указывает только на то, что Flash Player поддерживает воспроизведение звука, но оно не означает, что в компьютере фактически реализована поддержка звука (то есть имеется звуковая карта, включены громкоговорители, установлена достаточная громкость звучания и т. п.).

Соответствующей строкой сервера для hasAudio является A с возможными значениями t (true) и f (false). Подробности см. в описании capabilities.serverString.

См. также capabilities.hasAudioEncoder, capabilities.hasMP3

capabilities.hasAudioEncoder, свойство

Flash 6

указывает, поддерживает ли система кодировку звука **чтение/запись**

System.capabilities.hasAudioEncoder

Описание

Булево свойство hasAudioEncoder указывает, может ли проигрыватель Flash кодировать звуковой поток (например, от микрофона), что может потребоваться для передачи на сервер, такой как Macromedia Flash Communication Server MX (Comm Server). В настольной версии Flash Player 6 hasAudioEncoder всегда имеет значение true, указывающее на возможность кодировки звука. Однако это не гарантирует наличие необходимого оборудования.

Соответствующей строкой сервера для hasAudioEncoder является AE с возможными значениями t (true) и f (false). Подробности см. в описании capabilities.serverString.

См. также capabilities.hasAudio, capabilities.hasVideoEncoder

capabilities.hasMP3, свойство

Flash 6

указывает, поддерживает ли Player воспроизведение MP3 **чтение/запись**

System.capabilities.hasMP3

Описание

Булево свойство hasMP3 имеет значение true для тех проигрывателей Flash, которые могут воспроизводить звуковые файлы MP3, и значение false для тех, в которых отсутствует поддержка MP3. Обычно с его помощью определяют тип загружаемого на этапе исполнения звукового файла. Например, если фильм обнаруживает (с помощью свойства hasAudio), что система поддерживает воспроизведение звука, но поддержка MP3 отсутствует, он может загрузить обычный звук ADPCM.

Соответствующей строкой сервера для hasMP3 является MP3 с возможными значениями t (true) и f (false). Подробности см. в описании capabilities.serverString.

См. также capabilities.hasAudio

capabilities.hasVideoEncoder, свойство

Flash 6

указывает, может ли система кодировать видео **чтение/запись**

System.capabilities.hasVideoEncoder

Описание

Булево свойство hasVideoEncoder указывает, может ли Player кодировать видеопоток (например, от веб-камеры), что может потребоваться для передачи на сервер, такой как Macromedia Flash Communication Server MX (Comm Server). В версии Flash Player 6 для настольных компьютеров hasVideoEncoder всегда имеет значение true, указывающее на возможность кодировки изображения. Однако это не гарантирует наличие необходимого оборудования (например, веб-камеры).

Соответствующей строкой сервера для hasVideoEncoder является VE с возможными значениями t (true) и f (false). Подробности см. в описании capabilities.serverString.

См. также capabilities.hasAudioEncoder

capabilities.input, свойство

Flash 6

устройство для ввода данных пользователем **чтение/запись**

System.capabilities.input

Описание

Свойство input имеет строковый тип и указывает устройство ввода, предоставляемое аппаратурой системы. Для устройств типа мыши это тип «point». Если станут доступны другие устройства ввода, такие как перо (stylus) или кнопка управления курсором (cursor pad) на сотовом телефоне, будут определены их имена. Свойство input позволяет фильму переключиться, скажем, в особый режим ввода только с клавиатуры, который будет значительно более ограниченным, чем обычный режим ввода мышью.

Соответствующей строкой сервера для input является I с возможными значениями point и другими названиями определенных устройств. Подробности см. в описании capabilities.serverString.

См. также Объект *Mouse*

capabilities.isDebugger, свойство

Flash 6

указывает, является ли версия Player отладочной **чтение/запись**

`System.capabilities.isDebugger`

Описание

Булево свойство `isDebugger` указывает, является ли версия проигрывателя релизом (`false`) или отладочной (`true`). Отладочная версия используется вместе со средой разработки Flash для отладки фильмов Flash. Обычно это свойство не требуется разработчикам Flash.

Соответствующей строкой сервера для `isDebugger` является `DEB` с возможными значениями `t` (true) и `f` (false). Подробности см. в описании `capabilities.serverString`.

capabilities.language, свойство

Flash 6

язык, установленный в операционной системе **чтение/запись**

`System.capabilities.language`

Описание

Свойство `language` имеет строковый тип и содержит двухбуквенный код языка, определенного в операционной системе для текста. Используемые кросс-платформенные коды представляют собой подмножество стандарта ISO-639-1 «Коды для представления названий языков» (см. *http://lcweb.loc.gov/standards/iso639-2/englangn.html*). Например, «en» обозначает английский язык, а «fr» – французский. В некоторых случаях `language` содержит также двузначный код страны, указывающий местный вариант языка в соответствии со стандартом ISO 3166 (см. *http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html*). Код языка, если он есть, отделяется от кода страны дефисом. Например, «en-UK» обозначает британский вариант английского языка.

Проверив значение `language`, фильм может предоставить текст на языке, принятом в системе пользователя. Элементы на разных языках могут быть непосредственно встроены в один фильм или загружаться динамически в виде переменных, XML или фильмов целиком.

Соответствующей строкой сервера для `language` является `LAN` с возможными значениями, перечисленными в табл. C.4. Подробности см. в описании `capabilities.serverString`.

Таблица C.4. Коды языков Flash

Язык	Код языка в ISO 639-1	Код страны в ISO 3166
Английский (США)	en	US
Английский (Великобритания)	en	UK
Французский	fr	
Корейский	ko	
Японский	ja	
Шведский	sv	
Немецкий	de	

Язык	Код языка в ISO 639-1	Код страны в ISO 3166
Испанский	es	
Итальянский	it	
Упрощенный китайский	zh	CN
Традиционный китайский	zh	TW
Португальский	pt	
Польский	pl	
Венгерский	hu	
Чешский	cs	
Турецкий	tr	
Финский	fi	
Датский	da	
Норвежский	no	
Голландский	nl	
Русский	ru	
Прочие/неизвестные	xu	

Ошибки реализации

Код страны для английского языка (UK и US) неправильно устанавливается всеми версиями Flash Player 6. Для обеих стран `System.capabilities.language` возвращает «en» и не содержит код страны.

Пример

Следующий код загружает английскую или французскую версии фильма в зависимости от значения текущего языка:

```
if (System.capabilities.language.substring(0,2) == "en") {
  trace("English language");
  content_mc.loadMovie("news-english.swf");
} else if (System.capabilities.language == "fr") {
  trace("French language");
  content_mc.loadMovie("news-french.swf");
}
```

См. также «Кодировка символов» в главе 4

capabilities.manufacturer, свойство

Flash 6

указывает изготовителя Flash Player **чтение/запись**

`System.capabilities.manufacturer`

Описание

Свойство `manufacturer` имеет строковый тип и указывает название организации, изготовившей Flash Player. Сторонние разработчики могут приобрести лицензию на исходный код Macromedia Flash Player и перенести проигрыватель на другие платфор-

мы или устройства. Разработчики перенесенных на другие платформы или самостоятельно изготовленных проигрывателей могут предоставить свои значения свойства manufacturer. Для настольных версий Flash Player значениями могут быть только «Macromedia Windows» и «Macromedia Macintosh». Не следует полагаться на свойство manufacturer при получении информации об операционной системе или версии, для этого лучше предпочесть capabilities.os и capabilities.version.

Свойство manufacturer может быть использовано для обеспечения особой поддержки специальных возможностей, реализованных конкретным изготовителем проигрывателя.

Дополнительную информацию можно найти на странице:

http://www.macromedia.com/software/flash/open/licensing/

Соответствующей строкой сервера для manufacturer является M со значениями, определяемыми отдельными изготовителями. Подробности см. в описании capabilities.serverString.

См. также capabilities.os, capabilities.version

capabilities.os, свойство

Flash 6

указывает операционную систему, в которой выполняется Flash Player **чтение/запись**

System.capabilities.os

Описание

Свойство os имеет строковый тип и указывает название операционной системы, в которой выполняется Flash Player. Оно предоставляет более точную информацию, чем функция *getVersion( )*, которая не делает различий между разновидностями ОС (например, Windows 2000 и Windows 98). Текущие значения os в Windows следующие:

Windows XP
Windows 2000
Windows NT
Windows 98/ME
Windows 95
Windows CE

На компьютерах Макинтош os возвращает "MacOS *x*.*y*.*z*", где *x* – главный номер версии, *y* – дополнительный номер версии и *z* – номер версии через точку. Например:

MacOS 9.2.1
MacOS 10.1.4

Соответствующей строкой сервера для os является OS со значениями, указанными выше для Windows или Mac OS либо установленными отдельными изготовителями. Подробности см. в описании capabilities.serverString.

Ошибки реализации

В документации Macromedia по Flash MX ошибочно сказано, что по умолчанию значением os служит пустая строка. В действительности по умолчанию это та ОС, под которой выполняется Flash Player.

См. также capabilities.version, capabilities.manufacturer

capabilities.pixelAspectRatio, свойство

Flash 6

отношение ширины экранного пиксела к его высоте **чтение/запись**

System.capabilities.pixelAspectRatio

Описание

Свойство pixelAspectRatio имеет тип числа с плавающей точкой и указывает на пропорции отдельного экранного пиксела. Это отношение ширины пиксела к его высоте. Например, если ширина экранного пиксела составляет половину его высоты, то pixelAspectRatio возвращает 0,5 (1/2). В большинстве настольных компьютеров пикселы квадратные, поэтому pixelAspectRatio равно 1.

Свойство pixelAspectRatio позволяет фильму компенсировать возможные искажения, вызываемые соотношением геометрических размеров пикселов в устройстве воспроизведения. Например, круг с постоянным радиусом окажется эллипсом в устройствах, где отношение геометрических размеров пиксела не равно 1. Если ширина пикселов устройства вдвое больше их высоты, то можно загрузить альтернативный вариант фильма, ширина которого вдвое меньше, чтобы в итоге получить правильное отображение. На момент написания этой книги pixelAspectRatio всегда возвращало 1, и процедура разработки для устройств с неквадратными пикселами не была определена. Всегда тестируйте фильм на предполагаемом устройстве отображения, чтобы гарантировать корректное воспроизведение при соотношении геометрических размеров, отличном от 1. Macromedia будет предоставлять дополнительную информацию по мере ее появления.

Соответствующей строкой сервера для pixelAspectRatio является AR со значением в виде числа с плавающей точкой, например 1.0.

Ошибки реализации

В документации Macromedia по Flash MX ошибочно сказано, что строкой сервера для pixelAspectRatio является PAR. В действительности в строке сервера указывается AR.

См. также

capabilities.screenDPI, capabilities.screenResolutionX, capabilities.screenResolutionY

capabilities.screenColor, свойство

Flash 6

цветовой режим, поддерживаемый экраном (цветной, полутоновый или черно-белый) **чтение/запись**

System.capabilities.screenColor

Описание

Свойство screenColor имеет строковый тип и указывает цветовой режим, поддерживаемый устройством, на котором выполняется Flash Player. Это позволяет фильмам со слабым градиентом цветов переключаться на высококонтрастную цветовую схему при воспроизведении в системе с полутоновым или черно-белым изображением. Допустимыми значениями свойства – «color» (поддерживается более двух цветов), «gray» (полутоновое изображение) и «bw» (черно-белый или двухцветный дисплей).

Соответствующей строкой сервера для screenColor является COL с возможными значениями «color», «gray» и «bw». Подробности см. в описании capabilities.serverString.

См. также

Класс *Color*

capabilities.screenDPI, свойство

Flash 6

количество пикселов в одном дюйме экрана, обычно 72 dpi **чтение/запись**

`System.capabilities.screenDPI`

Описание

Свойство `screenDPI` имеет целочисленный тип и указывает количество точек (пикселов), помещающихся в одном дюйме экрана. Типичные мониторы настольных компьютеров показывают 72 пиксела в одном дюйме. Небольшие наладонные устройства могут отображать в одном дюйме значительно меньшее или большее количество пикселов. Можно считать, что чем выше `screenDPI`, тем более мелкие элементы будут видны в фильме.

Два экрана могут иметь одинаковое разрешение по горизонтали и вертикали, но фактические размеры фильма могут сильно различаться в зависимости от физических измерений экрана и количества пикселов в дюйме. Допустим, что у нас есть два экрана с разрешением 100×100. Если один экран представляет собой квадрат со стороной 3 дюйма (50 пикселов на дюйм), а другой – квадрат со стороной 5 дюймов (20 пикселов на дюйм), то шрифт высотой 10 пикселов будет на первом экране иметь высоту 1/5 дюйма, а на втором экране – полдюйма. Из-за такого различия экранов текст можно будет легко прочесть на одном экране, но на другом он окажется неразборчивым. Проверяя значения свойств `screenDPI`, `screenResolutionX` и `screenResolutionY`, фильм может масштабировать свое содержимое, чтобы оно отчетливо отображалось на данном дисплее.

Соответствующей строкой сервера для `screenDPI` является `DP` с любыми целочисленными значениями. Подробности см. в описании `capabilities.serverString`.

Применение

Специалисты по печати увидят некоторую неправильность в употреблении акронима DPI (*dots per inch*). Канонически «DPI» относится к печатной странице, тогда как к экрану применяется «PPI» (*pixels per inch*). Однако с этой неточностью придется примириться. Обратите внимание, что количество пикселов относится к одному линейному, а не квадратному дюйму.

См. также

`capabilities.pixelAspectRatio`, `capabilities.screenResolutionX`, `capabilities.screenResolutionY`, `Stage.scaleMode`

capabilities.screenResolutionX, свойство

Flash 6

ширина экрана в пикселах **чтение/запись**

`System.capabilities.screenResolutionX`

Описание

Свойство `screenResolutionX` представляет собой целое число, равное общей ширине в пикселах экрана, на котором отображается Flash Player. Оно аналогично свойству `Screen.width` JavaScript. С помощью `screenResolutionX` можно разместить в центре экрана всплывающее окно броузера или сообщить пользователю, что разрешение его экрана недостаточно для воспроизведения фильма, как показано ниже в примере.

Если известны физические размеры экрана, то с помощью screenResolutionX и screenResolutionY в сочетании со screenDPI можно вычислить фактический размер элемента фильма. Это удобно для масштабирования фильма таким образом, чтобы он имел одинаковый размер (в пикселах) на любом дисплее, как описывается в capabilities.screenDPI.

Для получения ширины сцены фильма (а не всего экрана монитора) применяется свойство Stage.width.

Соответствующей строкой сервера для screenResolutionX является R со значением, содержащим ширину и высоту экрана в формате «*ШИРИНА×ВЫСОТА*» (например, «1600×1200»).

Пример

Следующий код выводит предупреждение, если разрешение экрана оказывается недостаточным:

```
// Задать необходимое разрешение
minResolutionX = 800;
// Если это требование не удовлетворяется,...
if (System.capabilities.screenResolutionX < minResolutionX) {
  // ...вывести предупредительное сообщение
  this.createTextField("warning_txt", 1, 300, 300, 250, 30);
  this.warning_txt.border = true;
  this.warning_txt.text = "Фильм требует ширины экрана 800 пикселов.";

  // Показать выключенный кадр
  this.gotoAndStop("movieDisabled");
}
```

Ошибки реализации

В первом выпуске документации Macromedia по Flash MX это свойство ошибочно было указано как System.capabilities.screenResolution.x. Точки перед конечным x быть не должно, а буква X должна быть прописной.

См. также

capabilities.pixelAspectRatio, capabilities.screenDPI, capabilities.screenResolutionY, Stage.width

capabilities.screenResolutionY, свойство

Flash 6

высота экрана в пикселах **чтение/запись**

System.capabilities.screenResolutionY

Описание

Свойство screenResolutionY представляет собой целое число, равное общей высоте в пикселах экрана, на котором отображается Flash Player. Оно аналогично свойству Screen.height JavaScript. Для получения высоты сцены фильма (а не всего экрана монитора) применяется свойство Stage.height.

Соответствующей строкой сервера для screenResolutionY является R, со значением, содержащим ширину и высоту экрана в формате «*ШИРИНА×ВЫСОТА*» (например, «800×600»). См. пример проверки разрешения экрана в описании capabilities.screenResolutionX.

Ошибки реализации

В первом выпуске документации Macromedia по Flash MX это свойство ошибочно было указано как `System.capabilities.screenResolution.y`. Точки перед конечным `y` быть не должно, а буква `Y` должна быть заглавной.

См. также

`capabilities.pixelAspectRatio`, `capabilities.screenDPI`, `capabilities.screenResolutionX`, `Stage.height`

capabilities.serverString, свойство

Flash 6

значения всех свойств объекта Capabilities в виде строки переменных в кодировке URL — **чтение/запись**

`System.capabilities.serverString`

Описание

Свойство `serverString` имеет тип строки и содержит перечень имен переменных и их значений, соответствующих именам свойств объекта *Capabilities* и их значений, в кодировке URL.

Например, в моей системе значение `serverString` следующее:

```
A=t&MP3=t&AE=t&VE=t&ACC=f&DEB=t&V=WIN%206%2C0%2C21%2C0&M=Macromedia Windows&
R=1600x1200&DP=72&COL=color&AR=1.0&I=point&OS=Windows XP&L=en
```

Имя переменной и значение для каждого свойства можно найти в описании этого свойства в справочнике. Свойство `serverString` должно помочь разработчику организовать получение с сервера данных на основании спецификаций проигрывателя на стороне клиента. Эта строка может быть использована при загрузке внешнего XML, переменных и даже других фильмов. Например, простой фильм *loadStub.swf* может выступать в качестве модуля загрузки для последующего содержимого. Фильм *loadStub.swf* должен передать команду *loadMovie( )* сценарию на сервере и дописать значение свойства `serverString` к запросу GET:

```
content_mc.loadMovie("content.swf?" + System.capabilities.serverString);
```

В ответ на это сервер должен отправить фильм, соответствующий окружению на стороне клиента.

Применение

Свойства объекта *Capabilities* доступны по записи, но выполненные для них присваивания не отражаются на значении `serverString`.

См. также

MovieClip.loadMovie( ), класс *LoadVars*, класс *XML*

capabilities.version, свойство

Flash 6

версия Flash Player — **чтение/запись**

`System.capabilities.version`

Описание

Свойство `version` имеет строковый тип и содержит информацию о версии и платформе для Flash Player, в который загружен текущий фильм. Во Flash 6 это свойство следу-

ет предпочесть функции *getVersion( )*, хотя в обоих случаях возвращается одинаковое значение. А во Flash 5 свойство capabilites.version недоступно, поэтому в фильмах, предназначенных для воспроизведения с помощью этого проигрывателя, надо использовать *getVersion( )*. Аналогично в фильмах для Flash Player 4 следует применять $version.

Хранящаяся в свойстве version строка имеет формат:

```
platform majorVersion,minorVersion,buildNumber,patch
```

Здесь *platform* — это код, идентифицирующий платформу ("WIN", "MAC" или "UNIX"), за которым следуют основной номер версии, дополнительный номер версии и номер сборки (или «ревизии»). Последний элемент, patch, обычно равен 0. Например:

```
WIN 5,0,30,0   // Версия 5.0, сборка 30 (5.0r30) под Windows
MAC 5,0,41,0   // Версия 5.0, сборка 41 (5.0r41) на Macintosh
UNIX 4,0,12,0  // Версия 4.0, сборка 12 (4.0r12) под Unix
```

Более подробное описание текущей операционной системы можно плучить при помощи свойства capabilities.os.

В режиме Test Movie version сообщает номер версии *проигрывателя*, встроенного в среду разработки (не совпадающий с номером версии самой среды разработки). Так, в среду разработки Flash 5 встроен проигрыватель версии 5.0 r30, поэтому version будет иметь значение:

```
WIN 5,0,30,0
```

или:

```
MAC 5,0,30,0
```

Каждый раз, когда появляется новая основная или дополнительная версия среды разработки, нумерация buildNumber начинается с 0. Однако в типичном цикле разработки инструментальной среды Flash сборки Flash Player часто появляются до выхода окончательной версии среды разработки. Поэтому номер сборки для первой новой основной версии проигрывателя обычно больше 0. Например, Flash Player 5 был впервые официально выпущен как Build 30. Flash Player 6 был официально выпущен как Build 23. Значение version для фильма, воспроизводимого в первом релизе версии Flash Player 6 под Windows, будет равно WIN 6,0,23,0.

Обычно нас интересуют только платформа, основной номер версии и номер сборки проигрывателя. Выделить нужную нам часть строки version можно с помощью средств обработки строк, описанных в главе 4, а можно создать пользовательский объект, в котором все компоненты строки version будут представлены в виде свойств, как показано в примере ниже.

Когда фильм встраивается в броузер, JavaScript и VBScript предоставляют дополнительные средства для определения версии, типа броузера и автоматической переадресации страниц. О том, как с помощью JavaScript и VBScript определить наличие Flash Player и его версии, подробно рассказано на странице:

http://www.moock.org/webdesign/flash/detection/moockfpi

Соответствующей строкой сервера для version является V с возможными значениями, описанными выше.

Пример

Следующий код извлекает различные части строки `version` и сохраняет их как свойства объекта, что позволяет легко их получить:

```
// Расщепить строку version на нужные участки.
var version = System.capabilities.version;
var firstSpace = version.indexOf(" ");
var tempString = version.substring(firstSpace + 1, version.length);
var tempArray = tempString.split(",");

// Присвоить различные части строки getVersion() нашему объекту.
// Обратите внимание на преобразование частей номера версии в целые числа.
var thePlayer = new Object();
thePlayer.platform = version.substring(0,firstSpace);
thePlayer.majorVersion = parseInt(tempArray[0]);
thePlayer.minorVersion = parseInt(tempArray[1]);
thePlayer.build = parseInt(tempArray[2]);
thePlayer.patch = parseInt(tempArray[3]);

// Воспользуемся нашим объектом для выполнения кода, специфического для номера версии.
if ((thePlayer.majorVersion == 6) && (thePlayer.build <= 23))  {
  this.createTextField("warning_txt", 1, 300, 300, 250, 30);
  this.warning_txt.border = true;
  this.warning_txt.text = "Please upgrade your Player to avoid streaming bugs.";
}
```

Ошибки реализации

В документации Macromedia по Flash MX ошибочно сказано, что строка сервера для `version` называется `VER` (на самом деле это `V`) и что значение `version` представляет собой целое число (на самом деле это строка).

См. также `capabilities.os`, *getVersion( )*, `$version`

clearInterval(), глобальная функция

Flash 6

прекращает периодическое выполнение функции или метода

`clearInterval(`*intervalID*`)`

Аргументы

intervalID

Целочисленный идентификатор интервала, возвращенный предыдущим вызовом *setInterval( )*.

Описание

Функции *setInterval( )* и *clearInterval( )* предназначены для создания и уничтожения *интервалов*, которые периодически выполняют заданную функцию или метод объекта. Интервал создается при вызове функции *setInterval( )*, возвращающей уникальный идентификатор, который можно позднее передать функции *clearInterval( )*, чтобы прекратить действие этого интервала. Действие интервала можно прекратить как внутри, так и снаружи вызываемой им функции.

Примеры и подробности см. в описании *setInterval( )*.

Применение

В ActionScript нет функции *setTimeout( )* в стиле JavaScript. Для моделирования *setTimeout( )* создайте функцию интервала, которая сбрасывает вызов себя самой при первом выполнении, как показано в примере для *setInterval( )*.

См. также

setInterval( )

Color, класс

Flash 5

управление значениями цветов клипа

Конструктор

`new Color(target)`

Аргументы

target

Строка или ссылка, задающая путь к клипу или уровню документа, цвет которых будет управляться новым объектом (ссылки в строковом контексте преобразуются в пути).

Методы

getRGB( )

Извлекает текущие значения смещений для Red, Green и Blue.

getTransform( )

Извлекает текущие значения смещений и процентную величину для Red, Green, Blue и Alpha.

setRGB( )

Присваивает новые значения смещений Red, Green и Blue, уменьшая процентные значения до 0.

setTransform( )

Присваивает новые значения смещений и/или процентную величину для Red, Green, Blue и Alpha.

Описание

Объекты *Color* применяются для программного управления цветом и прозрачностью клипа или главного фильма. После создания объекта класса *Color* для некоторого `target` можно вызывать методы этого объекта, чтобы воздействовать на цвет и прозрачность `target`. Это означает, что не существует свойства `color`, позволяющего манипулировать цветом клипа; вместо этого создается объект *Color*, воздействующий на нужный клип. Допустим, что есть экземпляр клипа с именем `ball`, который надо сделать красным. Сначала создается объект Color, в качестве `target` которого указывается `ball`, и записывается в переменную `ballColor`. После этого с помощью *ballColor.setRGB( )* присваивается красный цвет экземпляру `ball`, например:

```
var ballColor = new Color("ball");
ballColor.setRGB(0xFF0000);  // Передать setRGB() шестнадцатеричное значение для красного цвета
```

Пример кода, помещающего метод установки цвета непосредственно в класс *MovieClip*, можно найти в описании метода *Color.setRGB( )*.

Рассмотренный пример показывает, как управлять цветом в простых приложениях. Для осуществления более сложных сценариев требуются дополнительные знания о представлении цвета во Flash. Каждый цвет, воспроизводимый в клипе, определяют четыре отдельные составляющие: Red, Green, Blue и Alpha (или прозрачность). Эти четыре составляющие объединяются в различных пропорциях для создания всех цветов, видимых на экране. Количество Red, Green, Blue или Alpha в некотором цвете описывается в виде числа в диапазоне от 0 до 255. Чем больше значение Red, Green или Blue, тем больше вклад каждого из этих цветов в окончательный цвет. Помните, однако, что в компьютере цвета складываются, а не вычитаются, как в краске, поэтому большие значения приводят к более светлым, а не к более темным цветам. Если значения всех трех RGB-составляющих равны, результатом будет оттенок серого цвета. Если все они равны 0, то получается черный цвет, а если 255, то белый. Чем больше значение Alpha, тем менее прозрачен результирующий цвет. (Цвет со значением Alpha 0 полностью прозрачен, а цвет со значением Alpha 255 полностью непрозрачен.)

Например, чистый красный цвет описывается значениями:

```
Red: 255, Green: 0, Blue: 0, Alpha: 255
```

а частично прозрачный красный может иметь значения:

```
Red: 255, Green: 0, Blue: 0, Alpha: 130
```

В рамках данной книги, говоря о цветах, будем придерживаться обозначений *тройки RGB* (*RGB triplet* – *Red*, *Green*, *Blue*). ActionScript не поддерживает десятичные значения троек RGB, такие как (255, 0, 255), но поддерживает эквивалентный шестнадцатеричный формат 0x*RRGGBB*, где *RR*, *GG* и *BB* – двузначные шестнадцатеричные числа, представляющие красный, зеленый и синий цвета. Например, тройка значений RGB (R:51, G:51, B:102) эквивалентна шестнадцатеричному значению 0x333366.

Из соображений удобства далее принято также обозначение *четверки RGBA* (*Red*, *Green*, *Blue*, *Alpha*). Обратите внимание, что у клипов нет свойства color, но есть свойство _alpha, представляющее значение канала _alpha.

Изначальные значения Red, Green, Blue и Alpha для *каждого* цвета в клипе устанавливаются в среде разработки Flash с помощью панели Color Mixer. (В панели Color Mixer значение Alpha представлено как процент, а не число от 0 до 255.) Чтобы изменить *все* цвета клипа с помощью ActionScript, выполняются общие настройки (называемые *трансформациями*) составляющих цвета клипа Red, Green, Blue и Alpha. Трансформации могут быть применены на этапе разработки посредством инспектора свойств (Flash MX) или панели Effect (Flash 5).

Есть два способа установить трансформацию для каждой составляющей цвета:

- Можно установить *процентное отношение* первоначального значения составляющей равным числу в диапазоне от -100 до 100. Скажем, можно потребовать, чтобы красный цвет, используемый в клипе, составлял 80% первоначального значения. Хотя отрицательный процент как самостоятельная величина не имеет смысла, но может фигурировать в расчетах окончательной трансформации, как будет показано ниже.
- Можно указать величину *смещения* первоначального значения составляющей. Смещение представляетт собой число в диапазоне от -255 до 255. Например, можно сказать: «добавить 20 ко всем значениям синего в данном клипе» или, с помощью отрицательного числа, «вычесть 30 из всех значений синего в данном клипе».

Окончательное значение цвета в трансформированном клипе определяется путем объединения первоначальных (заданных на этапе разработки) значений составляю-

щих цвета с процентными отношениями и смещениями, установленными с помощью объекта *Color*, например:

```
R = originalRedValue   * (redTransformPercentage/100)   + redTransformOffset
G = originalGreenValue * (greenTransformPercentage/100) + greenTransformOffset
B = originalBlueValue  * (blueTransformPercentage/100)  + blueTransformOffset
A = originalAlphaValue * (alphaTransformPercentage/100) + alphaTransformOffset
```

Если не осуществлялись никакие трансформации с помощью ActionScript, то начальное *процентное отношение* трансформации для каждой составляющей по умолчанию принимается равным 100, а первоначальное *смещение* – 0.

Рассмотрим на практическом примере, как действуют преобразования цвета. Пусть в клипе есть непрозрачный красный треугольник (R:255, G:0, B:0, A:255) и непрозрачный зеленый круг (R:0, G:255, B:0, A: 255). Предположим также, что к клипу применяется общая трансформация, устанавливающая процентное значение Green равным 50, процентное значение Alpha равным 80 и смещение Blue равным 100, а все остальные смещения и процентные отношения сохраняют значения по умолчанию (0 или 100). Вот результат воздействия общей трансформации на красный треугольник:

```
R == 255 * (100/100) +   0 == 255    // Красный не изменяется
G ==   0 *  (50/100) +   0 == 0      // Зеленый уменьшается на 50%
B ==   0 * (100/100) + 100 == 100    // Синий смещается на 100
A == 255 *  (80/100) +   0 == 204    // Непрозрачность уменьшается на 80%
```

Окончательным результатом трансформации цветов для красного треугольника становится значение цвета (R:255, G:0, B:100, A:204). А вот как действует эта трансформация на зеленый круг:

```
R ==   0 * (100/100) +   0 == 0      // Красный не изменяется
G == 255 *  (50/100) +   0 == 127.5  // Зеленый уменьшается на 50%
B ==   0 * (100/100) + 100 == 100    // Синий смещается на 100
A == 255 *  (80/100) +   0 == 204    // Непрозрачность уменьшается на 80%
```

Окончательно в результате трансформации зеленый круг имеет значение цвета (R:0, G:127,5, B:100, A:204).

Для того чтобы применить нашу гипотетическую трансформацию цвета к реальному клипу, обратимся к рассматриваемому объекту *Color*. Универсальные значения смещения и процентного отношения цветов устанавливаются посредством методов *setRGB( )* или *setTransform( )* (примеры кода приведены в статьях справочника, посвященных этим методам). Наоборот, получить текущую трансформацию о цветах клипа позволяют методы *getRGB( )* и *getTransform( )*. Посредством методов класса *Color* можно создавать цветовые эффекты анимации, например постепенное проявление, постепенное исчезновение и окрашивание. Кроме того, изменять оттенки каждого экземпляра клипа можно независимо, поэтому класс *Color* предоставляет весьма эффективный способ создания разнообразной графики с минимальными затратами. Например, можно создать сцену с множеством воздушных шариков с помощью одного клипа, который окрашивается и изменяет оттенки мириадами способов, как показано в следующем ниже примере.

Применение

Вот некоторые особенности применения объектов *Color*:

- Изменение цвета клипа с помощью объекта *Color* прекращает действие трансформаций, примененных к клипу на этапе разработки.

- Установка значения свойства _alpha клипа воздействует на процентное отношение Alpha, отражаемое в свойстве aa объекта, который возвращается методом *getTransform( )*.
- Трансформации цвета не действуют на цвет фона фильма или клипа. Они воздействуют только на залитые цветом фигуры, помещаемые на сцену. Чтобы изменить цвет фона, создайте фон с помощью фигуры и раскрашивайте ее.
- К клипам можно применять вручную трансформации цветов в среде разработки Flash MX с помощью инспектора свойств и во Flash 5 с помощью панели Effect (Window → Panels → Effect). Все такие трансформации отражаются в свойствах объекта, возвращаемого *getTransform( )*. Среда разработки отлично служит для просмотра и выбора трансформаций цветов во время редактирования фильма.

Пример

В первом примере показано, как сгенерировать серию клипов шариков со случайной окраской, исходя из существующего клипа с именем balloon:

```
// Цикл для создания 20 копий клипа balloon
for (var i = 0; i < 20; i++) {
  // Создать копию шарика
  balloon.duplicateMovieClip("balloon" + i, i);

  // Поместить шарик на сцену
  this["balloon" + i]._x = Math.floor(Math.random() * 550);
  this["balloon" + i]._y = Math.floor(Math.random() * 400);

  // Создать объект Color для этого шарика
  balloonColor = new Color(this["balloon" + i]);
  // Присвоить шарику случайный цвет с помощью метода setRGB()
  balloonColor.setRGB(Math.floor(Math.random() * 0xFFFFFF));
}
// Объект Color нам больше не нужен, поэтому удалим его
delete balloonColor;
```

Присваивая смещениям цветов Red, Green и Blue одинаковое значение, можно достичь осветления или затемнения клипа. Например, следующий код делает myClip более темным:

```
brightness = new Color("myClip");
brightnessTransform = new Object();
brightnessTransform.rb = -30;
brightnessTransform.bb = -30;
brightnessTransform.gb = -30;
brightness.setTransform(brightnessTransform);
```

В следующем (и последнем) примере вводится новый метод *MovieClip.doMouseFade( )*, осветляющий и затемняющий клип в зависимости от положения курсора мыши:

```
MovieClip.prototype.doMouseFade = function () {
  var brightness = new Color(this);
  var brightnessTransform = new Object();

  this.onMouseMove = function () {
    var brightnessAmount = -255 + (_level0._xmouse / Stage.width) * 510;
    brightnessTransform.rb = brightnessAmount;
    brightnessTransform.bb = brightnessAmount;
    brightnessTransform.gb = brightnessAmount;
    brightness.setTransform(brightnessTransform);
```

```
      updateAfterEvent();
    }
  }

  box.doMouseFade();
```

Color.getRGB(), метод

Flash 5

возвращает текущие значения смещений для Red, Green и Blue

colorObj.getRGB()

Возвращаемое значение

Число, выражающее текущие смещения RGB для объекта, управляемого *colorObj*.

Описание

Метод *getRGB( ) возвращает* число в диапазоне от –16 777 215 до 16 777 215, представляющее текущие смещения составляющих Red, Green и Blue цвета клипа. Для получения процентных отношений цветов следует применять *getTransform( )*. Поскольку значения смещений цветов обычно лежат в диапазоне от 0 до 255, удобно работать со значением, возвращаемым *getRGB( )*, в шестнадцатеричном виде, когда смещение каждого цвета может быть представлено двузначным шестнадцатеричным числом. Для расшифровки числа, возвращаемого *getRGB( )*, надо обрабатывать его как шестнадцатеричное число вида 0x*RRGGBB*, где *RR* – это смещение красного, *GG* – смещением зеленого и *BB* – смещением синего. Например, если преобразовать число 10 092 339, возвращаемое *getRGB( )*, в шестнадцатеричное число 0x99FF33, то получатся смещения цветов (R:153, G:255, B:51). Если же *getRGB( )* возвращает число 255, то оно преобразуется в шестнадцатеричное число 0x0000FF и будут получены смещения цветов (R:0, G:0, B:255). Значение, возвращаемое *getRGB( )*, можно преобразовать в шестнадцатеричный вид с помощью *toString( )*, например:

```
// Создать объект Color
myColor = new Color("myClip");
// Установить смещение красного 255 (FF в шестнадцатеричной системе)
myColor.setRGB(0xFF0000);
// Получить смещение RGB и преобразовать в шестнадцатеричный вид
hexColor = myColor.getRGB().toString(16);
trace(hexColor);  // Выводит: ff0000
```

Шестнадцатеричные значения цветов знакомы большинству веб-разработчиков, поскольку они часто используются в тегах HTML. Вот, например, как с помощью шестнадцатеричного числа задается цвет фона страницы (равные значения красного и синего вместе дают розовый):

```
<BODY BGCOLOR="#FF00FF">
```

Шестнадцатеричные коды цвета в тегах HTML – это, в сущности, тот же формат, с которым работают *getRGB( )* и *setRGB( )*. Однако не обязательно использовать для интерпретации значения, возвращаемого *getRGB( )*, шестнадцатеричные числа: можно извлекать отдельные значения смещений красного, зеленого и синего цветов из значения, возвращаемого *getRGB( )*, с помощью поразрядных операторов:

```
var rgb = myColorObject.getRGB();
var red = (rgb >> 16) & 0xFF;   // Выделить смещение красного и присвоить red
var green = (rgb >> 8) & 0xFF;  // Выделить смещение зеленого и присвоить green
var blue  = rgb & 0xFF;         // Выделить смещение синего и присвоить blue
```

Поместив значения смещений в переменные (по одной для каждого первичного цвета), можно работать с ними как с отдельными десятичными числами. Однако если потребуется применить изменения значений смещений к объекту *Color*, то придется снова объединить смещения в одно число, как показано в описании метода *setRGB( )*.

Применение

Методы *getRGB( )* и *setRGB( )* удобны при необходимости непосредственного присваивания клипу новых цветов без учета исходных цветов клипа. Методы *getTransform( )* и *setTransform( )* лучше подходят для изменения составляющих RGB в трансформации цветов клипа относительно его исходных цветов.

Значения смещений легче всего прочесть с помощью метода *getTransform( )*, возвращающего значения каждой составляющей в отдельности, а не одним числом, содержащим все три составляющие, как это делает *getRGB( )*. Это особенно относится к установке отрицательных смещений с помощью *setTransform( )* из-за способа представления отрицательных чисел в двоичном виде.

Пример

```
// Создать новый объект Color для клипа с именем box
boxColor = new Color("box");
// Задать новое смещение RGB для box
boxColor.setRGB(0x333366);
// Получить смещение RGB для box
trace(boxColor.getRGB());      // Выводит: 3355494
```

См. также *Color.getTransform( )*, *Color.setRGB( )*

Color.getTransform(), метод

Flash 5

извлекает как смещения, так и процентные отношения для составляющих Red, Green, Blue и Alpha цвета клипа

colorObj.getTransform()

Возвращаемое значение

Объект, в свойствах которого содержатся значения трансформаций цвета целевого клипа *colorObj*.

Описание

Метод *getTransform( )*[1] возвращает объект со свойствами, которые сообщают, какие трансформации применены в данный момент к целевому клипу объекта *Color*. Это означает, что метод возвращает несколько отдельных значений, для удобства заключенных в объект. Возвращаемый объект представляет собой экземпляр общего класса *Object* – это не объект *Color*. Названия свойств и их значения для возвращаемого объекта описаны в табл. C.5.

---

[1] Здесь есть ошибка реализации, которая может встретиться при разработке объектно-ориентированных приложений. Объект, возвращаемый функцией *getTransform( )*, не наследует никаких свойств и методов класса *Object* и вообще не имеет таких свойств, как __proto__ и constructor. Аналогичная ошибка встречается в объектах, возвращаемых функциями *TextField.getTextExtent( )*, *Color.getTransform( )*, *Sound.getTransform( )*. – *Примеч. науч. ред.*

Таблица С.5. Свойства объекта, возвращаемого методом getTransform( )

Имя свойства	Значение свойства	Описание свойства
ra	−100 ... 100	Процентное отношение трансформации Red
rb	−255 ... 255	Величина смещения Red
ga	−100 ... 100	Процентное отношение трансформации Green
gb	−255 ... 255	Величина смещения Green
ba	−100 ... 100	Процентное отношение трансформации Blue
bb	−255 ... 255	Величина смещения Blue
aa	−100 ... 100	Процентное отношение трансформации Alpha
ab	−255 ... 255	Величина смещения Alpha

Применение

Обратите внимание, что в табл. С.5 и процентное отношение, и смещение могут быть отрицательными. Хотя взятое отдельно отрицательное процентное отношение не имеет смысла, оно может участвовать в расчете RGB-составляющих цвета, которые всегда находятся в диапазоне 0–255. Значения, выходящие за диапазон, усекаются до допустимых величин. Описание расчета, производимого для определения итоговых составляющих RGB и Alpha, приводится в описании класса *Color*.

Пример

Метод *getTransform( )* в сочетании с *setTransform( )* может быть использован для того, чтобы отдельно изменять составляющие Red, Green, Blue и Alpha трансформации цвета. В следующем коде, например, настраиваются компоненты Red и Alpha клипа с именем box:

```
// Создать новый объект Color для клипа с именем box
boxColor = new Color("box");

// Запомнить объект, возвращенный getTransform(), в boxTransform
boxTransform = boxColor.getTransform();

// Теперь изменить некоторые свойства объекта трансформации
boxTransform.rb = 200;      // Задать смещение Red равным 200
boxTransform.aa = 60;       // Задать процент Alpha равным 60

// Применить новую трансформацию к box через boxColor
boxColor.setTransform(boxTransform);
```

См. также

Color.setTransform( )

Color.setRGB(), метод

Flash 5

присваивает новые значения смещений для Red, Green и Blue

colorObj.setRGB(*offset*);

Аргументы

offset

Число в диапазоне от 0 до 16 777 215 (0xFFFFFF), представляющее новые смещения RGB для клипа, которым управляет *colorObj*. Может быть десятичным или шест-

надцатеричным целым числом. Числа, не входящие в разрешенный диапазон, преобразуются в числа, принадлежащие допустимому диапазону (по правилам дополнения до двух в двоичных обозначениях). По этой причине *setRGB( )* нельзя использовать для установки отрицательных смещений (а *setTransform( )* можно).

Описание

Метод *setRGB( )* присваивает новые значения смещений трансформации для RGB-составляющих цвета клипа. Новое значение *offset* легче всего задать как шестизначное шестнадцатеричное число вида 0x*RRGGBB*, где *RR*, *GG* и *BB* — двузначные числа в диапазоне от 00 до FF, соответствующие компонентам Red, Green и Blue. Например, тройка RGB (51, 51, 102) эквивалентна шестнадцатеричному значению:

```
0x333366
```

Следовательно, чтобы назначить RGB-смещение серого цвета для клипа с именем menu, можно выполнить:

```
var menuColor = new Color("menu");
menuColor.setRGB(0x999999);
```

Веб-разработчикам, привыкшим к шестизначным шестнадцатеричным значениям цветов в HTML, будет легко использовать *setRGB( )* и предыдущий шестнадцатеричный формат. Учебник для начинающих по десятичным, восьмеричным и шестнадцатеричным числам можно найти на *http://www.moock.org/asdg/technotes/basePrimer*.

Обратите внимание, что помимо установки смещений *setRGB( )* также автоматически устанавливает процентные отношения Red, Green и Blue трансформации цветов клипа в 0, то есть изменения цвета, осуществляемые *setRGB( )*, действуют как прямое назначение цветов (а не корректировка первоначальных цветов клипа). Для настройки цвета клипа относительно его исходных цветов следует применять метод *setTransform( )*.

Пример

Вот удобный способ генерирования числа для метода *setRGB( )*. Пользовательская функция *combineRGB( )* сдвигает числа *red* и *green* в нужную позицию в 24-разрядном числе, а затем объединяет значения *red*, *green* и *blue* с помощью поразрядного оператора ИЛИ (|). Результат используется для присвоения цвета клипу box. Дополнительные сведения о поразрядных операциях можно найти по адресу *http://www.moock.org/asdg/technotes/bitwise*.

```
function combineRGB (red, green, blue) {
  // Объединить значения цветов в одно число
  var RGB = (red<<16) | (green<<8) | blue;
  return RGB;
}
// Создать объект Color
var boxColor = new Color("box");
// Установить цвет box согласно RGB-тройке (R:201, G:160, B:21)
boxColor.setRGB(combineRGB(201, 160, 21));
```

В следующем примере во все клипы добавляется удобный метод *setRGB( )*. Он автоматически создает объект *Color*, а затем вызывает метод *Color.setRGB( )*:

```
MovieClip.prototype.setRGB = function (colorValue) {
  new Color(this).setRGB(colorValue);
};

// Использование:
theClip_mc.setRGB(0xFF0000);  // Задать для theClip_mc красный цвет
```

См. также *Color.getRGB( ), Color.setTransform( )*

Color.setTransform(), метод

Flash 5

присваивает новые значения смещений и/или процентных отношений для Red, Green, Blue и Alpha

colorObj.setTransform(*transformObject*)

Аргументы

transformObject

Объект, свойства которого содержат новые значения трансформации цвета для клипа, которым управляет *colorObj*.

Описание

Метод *setTransform( )* предоставляет точный контроль над процентными отношениями и смещением составляющих Red, Green, Blue и Alpha цвета клипа. Для того чтобы работать с ним, необходимо сначала создать объект с рядом предопределенных свойств. Трансформация, которую требуется применить к объекту *Color*, описывается через эти свойства, перечисленные в табл. C.5 для метода *getTransform( )* (разница в том, что *setTransform( )* присваивает им значения, а не читает их).

Когда объект со свойствами, описанными в табл. C.5, создан, он передается методу *setTransform( )*. Значения свойств *transformObject* становятся новыми значениями смещения и процентного отношения трансформации *colorObj* и, следовательно, клипа, которым управляет *colorObj*. После этого окончательные значения цветов клипа определяются в соответствии с расчетом, приведенным в описании класса *Color*.

Текущие смещения и процентные отношения некоторого объекта *Color* можно узнать, вызвав метод *getTransform( )*.

Пример

```
// Создать новый объект Color для клипа box
boxColor = new Color("box");

// Создать новый общий объект (не типа Color) и сохранить его в boxTransform
boxTransform = new Object();

// Присвоить требуемые значения свойствам boxTransform(ra, rb и т. д.)
// для получения нужных значений трансформации
boxTransform.ra = 50;    // Процентное отношение Red
boxTransform.rb = 0;     // Смещение Red
boxTransform.ga = 100;   // Процентное отношение Green
boxTransform.gb = 25;    // Смещение Green
boxTransform.ba = 100;   // Процентное отношение Blue
boxTransform.bb = 0;     // Смещение Blue
boxTransform.aa = 40;    // Процентное отношение Alpha
boxTransform.ab = 0;     // Смещение Alpha

// Объект трансформации готов, передадим его в setTransform()
boxColor.setTransform(boxTransform);
```

Описанный подход к созданию объекта трансформации довольно трудоемок. Легче создать новый объект трансформации с помощью метода *getTransform( )*, например:

```
// Создать новый объект Color для клипа box
boxColor = new Color("box");
```

```
// Применить getTransform() к boxColor, чтобы получить объект трансформации
boxTransform = boxColor.getTransform();

// Теперь изменить только нужные свойства boxTransform, оставив прочие без изменений
boxTransform.rb = 51;   // Смещение Red
boxTransform.aa = 40;   // Процентное отношение Alpha

// Использовать объект трансформации в методе setTransform()
boxColor.setTransform(boxTransform);
```

См. также *Color.getTransform( )*

Date(), глобальная функция

Flash 5

строка, представляющая текущие дату и время

```
Date()
```

Возвращаемое значение

Строка, содержащая текущие дату и время.

Описание

Функция *Date( )* возвращает текстовую строку, описывающую текущие дату и время для местного часового пояса. Строка содержит также разницу с GMT (разность в часах между местным временем и временем по Гринвичу).

Не путайте глобальную функцию *Date( )* с конструктором класса *Date( )*. Функция *Date( )* возвращает дату в стандартном формате короткой строки. Ее удобно читать, но не очень удобно использовать в программе, когда требуется осуществлять действия над датами и временем. Для таких целей лучше применять объекты класса *Date*, позволяющие без труда обращаться отдельно к году, месяцу, числу и времени.

Пример

Функция *Date( )* позволяет минимальными усилиями поместить текущие дату и время в текстовое поле:

```
someMovieClip.createTextField("output_txt", 1, 0, 0, 200, 20);
output_txt.text = Date();

// Присваивает myTextField строку следующего формата:
// "Mon Aug 28 16:23:09 GMT-0400 2000"
```

Упражнение для читателя: модифицируйте предыдущий пример с помощью *String.substring( )*, чтобы выводились значения даты и времени (включая часы и минуты), но отсутствовали секунды и смещение относительно GMT. См. пример в описании класса *Date*, демонстрирующий гораздо более гибкий подход.

См. также Класс *Date*, *Date.UTC( )*

Date, класс

Flash 5

текущее время и структурированная поддержка информации в дате

Конструктор

```
new Date()
new Date(milliseconds)
new Date(year, month, day, hours, minutes, seconds, ms)
```

Аргументы

milliseconds

Количество миллисекунд между новой датой и полуночью 1 января 1970 года UTC (Coordinated Universal Time, универсальное время, сродни GMT[1], – отрицательное, если до этой даты, и положительное, если после). Необходимая настройка местного часового пояса осуществляется после определения даты по UTC. Например, задав аргумент `milliseconds` равным 1000 для восточного стандартного времени (Eastern Standard Time), получим дату на 1 секунду позднее полуночи 1 января 1970 года по UTC, что соответствует 7:00:01 пополудни 31 декабря 1969 года по EST (на 5 часов отстающему от UTC).

year

Целое число, задающее год. Необходимо при использовании формата конструктора `year, ... ms`. Если `year` состоит из одной или двух цифр, они рассматриваются как число лет после 1900 года (например, `year`, равный 11, всегда относится к 1911 году, а не к 2011). Для того чтобы задать год после 2000, укажите четыре цифры (например, 2010, а не 10). Год из трех цифр рассматривается как предшествующий 1000 года н. э. Во Flash 5 расчет не вполне надежен, если `year` отрицателен или меньше 800. Во Flash 6 отрицательный год задает количество лет перед 1900. Для задания дат до 1000 года н. э. более надежен формат конструктора `milliseconds`.

month

Целое число, задающее месяц, от 0 (январь) до 11 (декабрь), а не от 1 до 12. Необходим при использовании формата конструктора `year, ... ms`. Месяцы, выходящие за этот диапазон, переносятся в следующий или предыдущий год. Например, `month`, равный 13, рассматривается как февраль следующего года.

day

Необязательное целое число, задающее число месяца, от 1 до 31. Если число не указано, берется значение по умолчанию 1. Числа, выходящие за этот диапазон, переносятся в следующий или предыдущий месяцы. Например, 31 сентября рассматривается как 1 октября, а 0 сентября – как 31 августа.

hours

Необязательное целое число, задающее час суток, от 0 (полночь) до 23. Если час не указан, берется значение по умолчанию 0. Обозначения A.M. и P.M. (до и после полудня) не поддерживаются, поэтому используйте 24-часовой формат. Часы, выходящие за этот диапазон, переносятся в следующие или предыдущие сутки. Например, `hour`, равный 25, рассматривается как 1 час ночи следующих суток.

minutes

Необязательное целое число, задающее минуты, от 0 до 59. Если не задано, по умолчанию принимается значение 0. Минуты, выходящие за этот диапазон, переносятся в следующий или предыдущий час. Например, 60 минут рассматривается как первая минута следующего часа.

seconds

Необязательное целое число, задающее секунды, от 0 до 59. Если не задано, по умолчанию принимается значение 0. Секунды, выходящие за этот диапазон, пе-

---

[1] Greenwich Mean Time – международный стандарт. В UTC измерения времени основаны на показаниях атомных часов, в GMT – на астрономических наблюдениях. – *Примеч. ред.*

реносятся в следующую или предыдущую минуту. Например, –1 секунда рассматривается как последняя секунда предыдущей минуты.

ms

Необязательное целое число, задающее миллисекунды, от 0 до 999. Если не задано, по умолчанию принимается значение 0. Миллисекунды, выходящие за этот диапазон, переносятся в следующую или предыдущую секунду. Например `ms` со значением 1005 рассматривается как 5 миллисекунд в следующей секунде.

Методы класса

UTC( )

Позволяет получить количество миллисекунд между 1 января 1970 года и указанной датой по универсальному времени.

Методы

getDate( )

Получить число месяца, от 1 до 31.

getDay( )

Получить день недели как число от 0 (воскресенье) до 6 (суббота).

getFullYear( )

Получить год в формате четырех цифр.

getHours( )

Получить час из указанного объекта *Date* от 0 до 23.

getMilliseconds( )

Получить миллисекунды из указанного объекта *Date*, от 0 до 999.

getMinutes( )

Получить минуты из указанного объекта *Date*, от 0 до 59.

getMonth( )

Получить месяц года как число от 0 (январь) до 11 (декабрь).

getSeconds( )

Получить секунды из указанного объекта *Date*, от 0 до 59.

getTime( )

Получить дату во внутреннем формате (то есть как число миллисекунд между датой и 1 января 1970 года).

getTimezoneOffset( )

Получить разницу в минутах между UTC и местным временем.

getUTCDate( )

Получить число месяца по универсальному времени, от 1 до 31.

getUTCDay( )

Получить день недели по универсальному времени, от 0 (воскресенье) до 6 (суббота).

getUTCFullYear( )

Получить год в формате четырех цифр по универсальному времени.

getUTCHours( )

Получить час суток по универсальному времени, от 0 до 23.

getUTCMilliseconds( )

Получить миллисекунды по универсальному времени, от 0 до 999.

getUTCMinutes( )

Получить минуты по универсальному времени, от 0 до 59.

getUTCMonth( )

Получить месяц года по универсальному времени, от 0 (январь) до 11 (декабрь).

getUTCSeconds( )

Получить время в секундах по UTC, от 0 до 59.

getYear( )

Получить значение года относительно 1900.

setDate( )

Задать число месяца.

setFullYear( )

Задать год в формате четырех цифр.

setHours( )

Задать час суток.

setMilliseconds( )

Задать миллисекунды.

setMinutes( )

Задать минуты.

setMonth( )

Задать месяц года.

setSeconds( )

Задать секунды.

setTime( )

Задать дату во внутреннем формате (то есть как количество миллисекунд между датой и 1 января 1970 года).

setUTCDate( )

Задать число месяца по универсальному времени.

setUTCFullYear( )

Задать год в формате четырех цифр по универсальному времени.

setUTCHours( )

Задать час суток по универсальному времени.

setUTCMilliseconds( )

Задать миллисекунды по универсальному времени.

setUTCMinutes( )

Задать минуты по универсальному времени.

setUTCMonth( )

Задать месяц года по универсальному времени.

setUTCSeconds( )

Задать секунды по универсальному времени.

setYear( )

Задать год в формате четырех цифр или, для XX столетия, двух цифр.

toString( )

Удобная для чтения строка, представляющая дату.

valueOf( )

Количество миллисекунд между датой и полуночью 1 января 1970 года по универсальному времени.

Описание

Объекты класса *Date* выступают в качестве механизма для определения текущих времени и даты и сохранения произвольных дат и моментов времени в структурированном формате.

В ActionScript конкретная дата представляется как количество миллисекунд до или после полуночи 1 января 1970 года. Если это число положительное, значит, дата следует за полуночью 1 января 1970 года, а если отрицательное, то дата предшествует полуночи 1 января 1970 года. Например, если значение даты равно 10 000, то оно соответствует времени 12:00:10, 1 января 1970 года. Если значение даты равно -10 000, то оно соответствует времени 11:59:50, 31 декабря 1969 года.

Однако обычно не приходится беспокоиться о вычислении количества миллисекунд до какой-то даты: заботу об этом берет на себя интерпретатор ActionScript. При создании объекта *Date* просто описывается дата как год, месяц, число, час, минуты, секунды и миллисекунды. После этого интерпретатор преобразует данный момент времени во внутренний формат «миллисекунд после 1970». Можно также попросить интерпретатор создать новый объект *Date* исходя из текущего времени. После создания объекта *Date* можно использовать методы класса *Date* для получения и задания года, месяца, числа, часа, минут, секунд и миллисекунд даты.

Есть три способа создать новый объект *Date*:

- Вызвать конструктор *Date( )* без аргументов. В результате в новый объект *Date* записывается в текущее время.
- Вызвать конструктор *Date( )* с одним числовым аргументом – количеством миллисекунд между полуночью 1 января 1970 года и создаваемой датой.
- Вызвать конструктор *Date( )* с числовыми аргументами в количестве от двух до семи, соответствующими году и месяцу (обязательными) и (необязательными) днем, часом, минутами, секундами и миллисекундами создаваемой даты.

Поскольку даты внутренне хранятся как одно число, у объектов класса *Date* нет свойств, которые можно читать или записывать. Но есть методы, применяемые для доступа к различным компонентам даты в понятном человеку виде (то есть в более удобных единицах). Например, так определяется месяц в конкретном объекте *Date*:

```
myMonth = myDate.getMonth();
```

Нельзя писать:

```
myMonth = myDate.month;  // Нет такого свойства! Следует применять методы.
```

Многие методы *Date* содержат буквы «UTC» (Coordinated Universal Time, универсальное время). В большинстве случаев UTC употребляют как синоним более привычного Гринвичского времени (Greenwich Mean Time, GMT) – времени, измеренного на гринвичском меридиане. UTC – это новый стандарт, менее неоднозначный, чем

GMT, в разное время имевший разный смысл. Методы UTC позволяют работать со временными моментами непосредственно в универсальном времени без преобразований между часовыми поясами. Эквивалентные методы без использования UTC генерируют значения исходя из местных часовых поясов.

Применение

Все даты и моменты времени, в том числе по UTC, определяются согласно установкам операционной системы, в которой выполняется фильм Flash (а не веб-сервера), и учитывают региональные смещения, а в некоторых случаях – переход на летнее время. Поэтому точность времени и даты определяется системой пользователя.

Если *Date( )* используется как глобальная функция – без ключевого слова `new`, – создается строка, содержащая текущее время в таком же формате, который используется *myDate.toString( )*.

В ActionScript нет встроенных средств для преобразования строки в объект *Date*, которые имеются в JavaScript.

Пример

Даты можно складывать и вычитать для получения суммарного времени или истекшего времени. Допустим, что наш друг Грэм решил отправиться в путешествие продолжительностью чуть меньше года. Пока он отсутствует, нам надо отслеживать, сколько дней его уже нет и сколько дней осталось ждать его возвращения. Такой отсчет ведет следующий код:

```
// Записать текущее время в now
var now = new Date();

// Грэм уезжает 7 сентября 2000 года (напомним, месяцы нумеруются с 0,
// поэтому сентябрю соответствует 8, а не 9)
var departs = new Date(2000,8,7);

// Грэм возвращается 15 августа 2001 года
var returns = new Date(2001,7,15);

// Преобразовать время в миллисекунды для облегчения сравнения.
// Узнать, сколько миллисекунд прошло между двумя моментами времени.
var gone = now.getTime() - departs.getTime();

// Поделить разницу между моментом отъезда и текущим временем на количество
// миллисекунд в сутках. В результате узнаем, сколько дней мы ждем.
var numDaysGone = Math.floor(gone/86400000);

// Таким же способом узнаем, сколько дней осталось ждать.
// Но на этот раз выполним округление вверх, а не вниз (с помощью функции ceil()).
var left = returns.getTime() - now.getTime();
var numDaysLeft = Math.ceil(left/86400000);

// Вывести полученные количества дней в текстовых полях.
this.createTextField("goneOutput_txt", 1, 0, 0, 200, 20);
this.createTextField("leftOutput_txt", 2, 0, 40, 200, 20);
goneOutput_txt.text = "gone for: " + numDaysGone + " days";
leftOutput_txt.text = "returns in: " + numDaysLeft + " days";
```

При добавлении или вычитании суток из имеющейся даты обычно для удобства количество миллисекунд в сутках (86 400 000) присваивается переменной. Следующий код добавляет одни сутки к текущей дате:

```
oneDay = 86400000;
now = new Date();
tomorrow = new Date(now.getTime() + oneDay);

// Можно добавить одни сутки к дате в now и так
now.setTime(now.getTime() + oneDay);
```

Чтобы использовать собственный формат для вывода даты, необходимо преобразовать значения, возвращаемые *getDate( )*, *getDay( )*, *getHours( )* и т. д., в нужные строки, как показано в следующем примере:

```
// Функция принимает объект Date и возвращает строку в формате:
// Saturday December 16
function formatDate(theDate) {
  var months = ["January", "February", "March", "April", "May", "June", "July", "August",
                "September", "October", "November", "December"];
  var days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"];
  var dateString = days[theDate.getDay()] + " "
                   + months[theDate.getMonth()] + " "
                   + theDate.getDate();
  return dateString;
}

now = new Date();
trace("Today is " + formatDate(now));
```

В следующем примере показано, как преобразовать значение по 24-часовой шкале, возвращаемое *getHours( )*, в 12-часовой циферблат с AM и PM:

```
// Функция принимает объект Date и возвращает строку в формате:
// "2:04PM"
function formatTime(theDate) {
  var hour = theDate.getHours();
  var minute = theDate.getMinutes() > 9 ?
               theDate.getMinutes() : "0" + theDate.getMinutes();
  if (hour > 12) {
    var timeString = (hour - 12) + ":" + minute + "PM";
  } else {
    var timeString = hour + ":" + minute + "AM";
  }
  return timeString;
}

now = new Date();
trace("The time is " + formatTime(now));
```

Пример применения класса *Date* для создания аналоговых часов приведен в разделе «Создание часов с помощью клипов» главы 13. Отметим, что программирование поведения, зависящего от времени, например 10-секундной паузы, часто легче осуществить с помощью глобальных функций *getTimer( )* и *setInterval( )*.

См. также *Date( )*, *Date.UTC( )*, *getTimer( )*

Date.getDate(), метод

Flash 5

число месяца

date.getDate()

Возвращаемое значение

Целое число от 1 до 31, представляющее число месяца для *date*.

Date.getDay(), метод

Flash 5

день недели

date.getDay()

Возвращаемое значение

Целое число от 0 (воскресенье) до 6 (суббота), представляющее день недели для *date*.

Пример

Следующий код загружает файл *.swf*, соответствующий текущему дню недели, в клип welcomeHeader (семь файлов *.swf* названы *sun.swf*, *mon.swf* и т. д.):

```
now = new Date();
today = now.getDay();
days = ["sun", "mon", "tue", "wed", "thu", "fri", "sat"];
welcomeHeader.loadMovie(days[today] + ".swf");
```

Date.getFullYear(), метод

Flash 5

год в формате четырех цифр

date.getFullYear()

Возвращаемое значение

Четырехзначное целое, представляющее год для *date*, например 1999 или 2000.

См. также

Date.getYear( )

Date.getHours(), метод

Flash 5

час суток

date.getHours()

Возвращаемое значение

Целое число от 0 (полночь) до 23 (11 вечера), представляющее час суток для *date*. Обозначения A.M. и P.M. не поддерживаются, но могут быть созданы вручную, как показано в примерах к классу *Date*.

Date.getMilliseconds(), метод

Flash 5

миллисекунды даты

date.getMilliseconds()

Возвращаемое значение

Целое число от 0 до 999, представляющее миллисекунды в *date*. Обратите внимание, что это не миллисекунды после 1970 года (см. *Date.getTime( )*), а дробная часть секунд, содержащаяся в данном объекте *Date*.

См. также

Date.getTime( )

Date.getMinutes(), метод

Flash 5

минуты даты

date.getMinutes()

Возвращаемое значение

Целое число от 0 до 59, представляющее минуты часа в *date*.

Date.getMonth(), метод

Flash 5

месяц года

date.getMonth()

Возвращаемое значение

Целое число от 0 (январь) до 11 (декабрь), а не от 1 до 12, представляющее месяц года в *date*.

Применение

Помните, что 1 – это не январь! Значения, возвращаемые *getMonth()*, начинаются с 0, а не с 1.

Пример

Следующий код преобразует число, возвращаемое *getMonth()*, в читаемое сокращенное название месяца:

```
var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sept", "Oct", "Nov", "Dec"];
myDateObj = new Date();
trace ("The month is " + months[myDateObj.getMonth()]);
```

Date.getSeconds(), метод

Flash 5

секунды даты

date.getSeconds()

Возвращаемое значение

Целое число от 0 до 59, представляющее секунды минуты в *date*.

Date.getTime(), метод

Flash 5

количество миллисекунд между 1 января 1970 года и временем в объекте даты

date.getTime()

Возвращаемое значение

Целое число, выражающее количество миллисекунд между временем в *date* и полуночью 1 января 1970 года. Положительное для дат после 1 января 1970 года и отрицательное для дат, предшествующих 1 января 1970.

Описание

Внутренне все даты представляются одним числом – количеством миллисекунд между временем *date* и полуночью 1 января 1970 года. Метод *getTime()* предоставляет

доступ к этому количеству миллисекунд, которое можно использовать для создания других дат или вычисления промежутка времени между двумя датами.

Пример

Пусть в кадре 10 фильма есть такой код:

```
time1 = new Date();
```

Следующий код поместим в 20-й кадр фильма:

```
time2 = new Date();
```

Промежуток времени, истекший между 10-м и 20-м кадрами фильма, в миллисекундах, можно определить так:

```
elapsedTime = time2.getTime() - time1.getTime();
```

Заметим, что глобальная функция *getTimer( )* тоже дает доступ к величине истекшего времени в фильме.

См. также *Date.setTime( )*, *getTimer( )*, *setInterval( )*

Date.getTimezoneOffset(), метод

Flash 5

разница в минутах между местным временем и UTC (известным еще как GMT)

date.getTimezoneOffset()

Возвращаемое значение

Целое число, представляющее текущую разницу в *минутах* между местным часовым поясом и фактическим временем UTC (Гринвичского меридиана). Положительное, если местное время отстает от UTC; отрицательное, если местное время опережает UTC. Содержит поправки для летнего времени в зависимости от дня в году.

Пример

При вызове по восточному поясному времени (EDT) в сезон летнего энергосбережения следующий код возвращает 240 (240 минут в 4 часах):

```
myDate = new Date();
trace(myDate.getTimezoneOffset());  // Выводит: 240
```

Однако при вызове по EDT не в сезон летнего энергосбережения тот же код возвратит 300 (300 минут в 5 часах), что представляет действительное смещение стандартного восточного времени (EST) относительно UTC.

Date.getUTCDate(), метод

Flash 5

число месяца (время UTC)

date.getUTCDate()

Возвращаемое значение

Целое от 1 до 31, представляющее число месяца для *date*, где *date* рассматривается как время UTC.

Date.getUTCDay(), метод

Flash 5

день недели (время UTC)

date.getUTCDay()

Возвращаемое значение

Целое число от 0 (воскресенье) до 6 (суббота), представляющее день недели для *date*, где *date* рассматривается как время UTC.

Date.getUTCFullYear(), метод

Flash 5

год в четырехзначном формате (время UTC)

date.getUTCFullYear()

Возвращаемое значение

Четырехзначное целое, представляющее год для *date*, где *date* рассматривается как время UTC, например 1999 или 2000.

Date.getUTCHours(), метод

Flash 5

час суток (время UTC)

date.getUTCHours()

Возвращаемое значение

Целое число от 0 (полночь) до 23 (11 вечера), представляющее час суток для *date*, где *date* рассматривается как время UTC.

Date.getUTCMilliseconds(), метод

Flash 5

миллисекунды даты (время UTC)

date.getUTCMilliseconds()

Возвращаемое значение

Целое число от 0 до 999, представляющее миллисекунды в *date*, где *date* рассматривается как время UTC.

Date.getUTCMinutes(), метод

Flash 5

минуты даты (время UTC)

date.getUTCMinutes()

Возвращаемое значение

Целое число от 0 до 59, представляющее минуты часа в *date*, где *date* рассматривается как время UTC.

Date.getUTCMonth(), метод

Flash 5

месяц года (время UTC)

date.getUTCMonth()

Возвращаемое значение

Целое число от 0 (январь) до 11 (декабрь), а не от 1 до 12, представляющее месяц года *date*, где *date* рассматривается как время UTC.

Применение

Помните, что 1 – это не январь! Значения, возвращаемые *getUTCMonth( )*, начинаются с 0, а не с 1.

Date.getUTCSeconds(), метод

Flash 5

секунды даты (время UTC)

`date.getUTCSeconds()`

Возвращаемое значение

Целое число от 0 до 59, представляющее секунды минуты *date*, где *date* рассматривается как время UTC.

Date.getYear(), метод

Flash 5

год относительно 1900

`date.getYear()`

Возвращаемое значение

Величина `date.getFullYear() - 1900`. Например, *getYear( )* от 1999 дает 99, *getYear( )* от 2001 дает 101, а *getYear( )* от 1800 дает –100. Эта функция полезнее всего для дат XX века.

См. также *Date.getFullYear( )*

Date.setDate(), метод

Flash 5

задает число месяца

`date.setDate(day)`

Аргументы

day Целое число от 1 до 31, представляющее новое число месяца для *date*. Если задать число, превосходящее количество дней в текущем месяце, номер месяца будет соответствующим образом увеличен. Например, если текущий месяц сентябрь (8) и для *day* задать новое значение 31, это будет воспринято как 1 октября. *day* станет равным 1, а *month* примет значение 9 (октябрь).

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

См. также *Date.getDate( )*, *Date.setMonth( )*

Date.setFullYear(), метод

Flash 5

задает век и год в четырехзначном формате

`date.setFullYear(year, month, day)`

Аргументы

year Четырехзначное целое, представляющее новый год для *date*, например 1999 или 2000.

month Необязательное целое число от 0 (январь) до 11 (декабрь), а не от 1 до 12, представляющее новый месяц года *date*. Если не задано, принимается равным 0.

day Необязательное целое число от 1 до 31, представляющее новое число месяца *date*. Если не задано, принимается равным 1.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

См. также *Date.getFullYear( ), Date.setYear( )*

Date.setHours(), метод

Flash 5

час суток

date.setHours(*hour*)

Аргументы

hour Целое число от 0 (полночь) до 23 (11 вечера), задающее новый час суток для *date*.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

См. также *Date.getHours( )*

Date.setMilliseconds(), метод

Flash 5

задает миллисекунды даты

date.setMilliseconds(*ms*)

Аргументы

ms Целое число от 0 до 999, представляющее новое значение миллисекунд в *date*, а не количество миллисекунд после 1970. Значения свыше 999 и меньше 0 переносятся в секунды *date*.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

См. также *Date.getMilliseconds( ), Date.setTime( )*

Date.setMinutes(), метод

Flash 5

задает минуты даты

date.setMinutes(*minute*)

Аргументы

minute

Целое число от 0 до 59, представляющее новое значение минут часа *date*. Значения свыше 59 или меньше 0 переносятся в часы *date*.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

См. также *Date.getMinutes( )*

Date.setMonth(), метод

Flash 5

задает месяц года

`date.setMonth(month)`

Аргументы

month

Целое число от 0 (январь) до 11 (декабрь), а не от 1 до 12, представляющее новый месяц года в `date`. Значения свыше 11 или меньше 0 переносятся в год `date`.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

Применение

Помните, что 1 – это не январь! Значения месяцев начинаются с 0, а не с 1.

См. также *Date.getMonth( )*

Date.setSeconds(), метод

Flash 5

задает секунды даты

`date.setSeconds(second)`

Аргументы

second

Целое число от 0 до 59, представляющее количество секунд минуты `date`. Значения свыше 59 и ниже 0 переносятся в минуты `date`.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

См. также *Date.getSeconds( )*

Date.setTime(), метод

Flash 5

задает новое значение даты исходя из количества миллисекунд между 1 января 1970 года и новой датой

`date.setTime(milliseconds)`

Аргументы

milliseconds

Целое число, представляющее количество миллисекунд между новой требуемой датой и полуночью 1 января 1970 года. Положительно для дат после 1 января 1970 года, отрицательно для предшествующих дат.

Возвращаемое значение

Значение *milliseconds*.

Описание

Внутренне все даты представляются как количество миллисекунд между временем даты и полуночью 1 января 1970 года. Метод *setTime( )* задает новую дату с использованием внутреннего представления в миллисекундах. Установка даты с помощью миллисекунд от 1970 года часто удобна, когда определяются разности между несколькими датами с помощью *getTime( )*.

Пример

Применяя совместно *setTime( )* и *getTime( )*, можно изменять время в имеющейся дате, добавляя или вычитая миллисекунды. Например, прибавим к дате один час:

```
now = new Date();
now.setTime(now.getTime() + 3600000);
```

А теперь прибавим один день:

```
now = new Date();
now.setTime(now.getTime() + 86400000);
```

Чтобы код легче читался, создают переменные, представляющие количество миллисекунд в часе и количество миллисекунд в сутках:

```
oneDay = 86400000;
oneHour = 3600000;
now = new Date();
// Вычесть одни сутки и три часа.
now.setTime(now.getTime() - oneDay - (3 * oneHour));
```

См. также *Date.getTime( )*, *Date.setMilliseconds( )*, *Date.UTC( )*, *getTimer( )*

Date.setUTCDate(), метод

Flash 5

задает число месяца (время UTC)

date.setUTCDate(*day*)

Аргументы

day

Целое число от 1 до 31, представляющее новое число месяца для *date* по времени UTC. Если задать число месяца, превосходящее количество дней в текущем месяце, месяц соответственно увеличивается. Например, если текущий номер месяца равен 8 (сентябрь), а в качестве нового значения *day* задано 31, то получится 1 октября. *day* примет значение 1, а *month* станет равным 9 (октябрь).

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года по времени UTC.

Date.setUTCFullYear(), метод

Flash 5

задает век и год в формате четырехзначного числа (время UTC)

date.setUTCFullYear(*year*)

Аргументы

year

Четырехзначное целое число, представляющее новый год для `date` по времени UTC, например 1999 или 2000.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года по времени UTC.

Date.setUTCHours(), метод

Flash 5

задает час суток (время UTC)

`date.setUTCHours(hour)`

Аргументы

hour

Целое число от 0 (полночь) до 23 (11 вечера), представляющее новый час суток для `date` по времени UTC.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года по времени UTC.

Date.setUTCMilliseconds(), метод

Flash 5

задает миллисекунды для даты (время UTC)

`date.setUTCMilliseconds(ms)`

Аргументы

ms

Целое число от 0 до 999, представляющее новое значение миллисекунд `date` по времени UTC.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года по времени UTC.

Date.setUTCMinutes(), метод

Flash 5

задает значение минут для даты (время UTC)

`date.setUTCMinutes(minute)`

Аргументы

minute

Целое число от 0 до 59, представляющее новое значение минут для даты `date` по времени UTC.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года по времени UTC.

Date.setUTCMonth(), метод

Flash 5

задает месяц года (время UTC)

`date.setUTCMonth(month)`

Аргументы

month

Целое число от 0 (январь) до 11 (декабрь), а не от 1 до 12, представляющее новый месяц года *date* по времени UTC.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года по времени UTC.

Применение

Помните, что 1 – это не январь! Значения месяцев начинаются с 0, а не с 1.

Date.setUTCSeconds(), метод

Flash 5

задает значение секунд даты (время UTC)

`date.setUTCSeconds(second)`

Аргументы

second

Целое число от 0 до 59, представляющее новое значение секунд минуты *date* по времени UTC.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года по времени UTC.

Date.setYear(), метод

Flash 5

задает год относительно 1900

`date.setYear(year, month, day)`

Аргументы

year

Обязательное целое число, задающее новый год *date*. Если передаются одна или две цифры, год считается относящимся к XX веку (1900-е годы). Например, 1 задает год 1901, а 99 задает год 1999. Год из трех цифр считается относящимся к периоду до 1000 года н. э. Четыре цифры используются для задания 2000 года и далее. Отрицательное число задает количество лет до н. э., а не до 1900 года.

month

Необязательное целое число от 0 (январь) до 11 (декабрь), а не от 1 до 12, представляющее новый месяц года *date*.

day

Необязательное целое число от 1 до 31, представляющее новое число месяца *date*.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

Описание

setYear( ) идентичен *setFullYear( )*, за исключением того, что год из одной или двух цифр интерпретируется относительно 1900, тогда как *setFullYear( )* интерпретирует их относительно 0 года н. э.

См. также *Date.getYear( )*, *Date.setFullYear( )*

Date.toString(), метод — Flash 5

удобная для чтения строка, представляющая дату

date.toString()

Возвращаемое значение

Строка, выводящая время и дату *date* в удобном для чтения виде, включая разницу с UTC (GMT). Для создания специальных представлений даты следует применять методы, извлекающие число, час, минуты и т. д., и преобразовывать эти значения в собственные строки, как было показано выше в примере для класса *Date*.

Пример

```
trace (myDate.toString()); // Выводит дату в формате: Wed Sep 15 12:11:33 GMT-0400 1999
```

Date.UTC(), метод класса — Flash 5

возвращает количество миллисекунд между 1 января 1970 года и заданной датой UTC

date.UTC(*year*, *month*, *day*, *hours*, *minutes*, *seconds*, *ms*)

Аргументы

year,...ms

Ряд числовых значений, описывающих дату и время, но указанных по времени UTC, а не по местному времени. Описания всех аргументов приведены в конструкторе *Date( )*.

Возвращаемое значение

Целое число, представляющее количество миллисекунд между указанной датой и полуночью 1 января 1970 года.

Описание

Метод *Date.UTC( )* принимает такие же аргументы, как конструктор *Date( )*, но возвращает не объект с указанной датой, а число, определяющее дату во внутреннем формате миллисекунд после 1970 года. Возвращаемое число обычно используется для создания нового объекта *Date* в UTC или присваивания времени UTC существующему объекту *Date* с помощью метода *setTime( )*.

Пример

В следующем коде показано, как измерить количество миллисекунд, прошедших между полуночью 1 января 1970 года и полуночью 1 января 2000 года по UTC:

```
trace(Date.UTC(2000, 0) + " milliseconds passed between 1970 and 2000.");
```

```
// Выводит: "946684800000 milliseconds passed between 1970 and 2000."
```

Теперь применим эти истекшие миллисекунды для создания объекта *Date* с датой UTC:

```
nowUTC = new Date(Date.UTC(2000, 0));
```

Если этот код вызвать в EST (стандартном восточном времени), которое на 5 часов отстает от UTC, то `nowUTC` покажет местное время 7 часов вечера 31 декабря 1999 года. Если получить час суток с помощью не учитывающего UTC метода *getHours( )*, получится местное время 19 часов:

```
trace(nowUTC.getHours());  // Выводит: 19
```

Но если получить час с помощью метода *getUTCHours( )*, учитывающего UTC, получим правильное время по UTC, 0 (полночь на 24-часовом циферблате):

```
trace(nowUTC.getUTCHours());  // Выводит: 0
```

См. также *Date( )*, *Date.setTime( )*, класс *Date*

Date.valueOf(), метод

Flash 5

количество миллисекунд между 1 января 1970 года и временем объекта Date

date.`valueOf()`

Возвращаемое значение

Целое число, представляющее количество миллисекунд между временем объекта *Date* и полуночью 1 января 1970 года. Положительно после 1 января 1970 года, отрицательно до этой даты.

Описание

На практике *Date.valueOf( )* эквивалентен *Date.getTime( )*.

См. также *Date.toString( )*

delete, оператор

Flash 5

удаляет содержимое переменной, свойства или элемента массива

`delete` *identifier*

Операнд

identifier

Удаляемый идентификатор.

Возвращаемое значение

Булево значение `true`, если *identifier* был удален; `false`, если нет (из-за отсутствия *identifier* или невозможности удалить его содержимое).

Описание

Оператор *delete* удаляет из сценария объект, свойство объекта, элемент массива или переменную.

См. также «Оператор *delete*» в главе 5

duplicateMovieClip(), глобальная функция

Flash 4

создает копию клипа

`duplicateMovieClip(target, newName, depth)`

Аргументы

target

Строка, указывающая путь к клипу, который надо дублировать (называемому также порождающим *клипом* (*seed clip*)). Ссылаться на вложенные клипы можно с помощью синтаксиса точки, например *duplicateMovieClip("_root.myClip", "myClip2", 0)*. В строковом контексте ссылка на клип преобразуется в путь, поэтому `target` может также быть ссылкой на объект клипа, как в *duplicateMovieClip(myClip, "myClip2", 0)*.

newName

Строка, которая станет именем экземпляра дублируемого клипа. Она должна соответствовать правилам создания идентификаторов, приведенным в разделе «Идентификаторы» главы 15.

depth

Целое число в интервале от –16 384 до 1 048 575 включительно, указывающее глубину внутри родительского для `target` объекта, на которую должен быть помещен продублированный клип. Клипы с более высоким значением глубины закрывают те, у которых значение глубины меньше, поэтому клип с глубиной 6 закрывает клип с глубиной 5, закрывающий, в свою очередь, клип с глубиной 4, и т. д. Глубина клипов, превосходящая –1, зарезервирована для динамически создаваемого содержимого. Поэтому обычно метод *duplicateMovieClip( )* следует применять с глубиной от 0 до 1 048 575. Однако для создания динамического содержимого, которое должно отображаться в `target` ниже любого содержимого, созданного на этапе разработки (то есть в качестве фона этапа исполнения), задавайте глубину –16 384. Глубины от –16 383 до –1 включительно зарезервированы для содержимого, создаваемого на этапе разработки, и не должны использоваться с *duplicateMovieClip( )*. Важные подробности см. в главе 13 в разделе «Порядок размещения фильмов и экземпляров в стеке».

Описание

Глобальная функция *duplicateMovieClip( )* предоставляет один из способов создать новый клип во время воспроизведения фильма (другие способы дают *MovieClip.attachMovie( )*, *MovieClip.createEmptyMovieClip( )* и *MovieClip. duplicateMovieClip( )*). Функция *duplicateMovieClip( )* создает точную копию `target` и помещает ее в стек клипов родительского клипа `target` на уровень `depth`. Дубликат клипа начинает воспроизводиться с кадра 1, каким бы ни был кадр `target` во время дублирования.

Дублированный клип наследует все преобразования (поворот, масштаб и пр.), применявшиеся к `target`, но не наследует его переменные временной диаграммы `target`. Функции обратного вызова, прикрепленные к свойствам-обработчикам событий `target` (например, *onEnterFrame( )*), не копируются в продублированный клип. Однако обработчики событий *onClipEvent( )* в стиле Flash 5 копируются в дубликат. Глобальная функция *duplicateMovieClip( )* доступна также в виде метода клипа, однако в таком формате не используется аргумент `target`.

Удалить продублированный клип можно с помощью *removeMovieClip( )* (важные дополнительные подробности см. в описании глобальной функции *removeMovieClip( )*).

Ошибки

В глобальной форме *duplicateMovieClip( )* отсутствуют следующие возможности *MovieClip. duplicateMovieClip( )*, появившиеся во Flash 6:

- В глобальной форме отсутствует поддержка параметра `initObject` (используемого для инициализации свойств вновь созданного клипа).
- Глобальная форма не возвращает ссылку на продублированный клип.

Пример

```
// Скопировать клип ball_mc и дать копии имя ball2_mc
duplicateMovieClip(ball_mc, "ball2_mc", 0);
// Переместить созданный клип ball2_mc, чтобы его можно было увидеть
ball2_mc._x += 100;
```

См. также

MovieClip.duplicateMovieClip( ), *removeMovieClip( )*; «Создание экземпляров с помощью *duplicateMovieClip( )*», «Добавление в стек клипов, созданных с помощью *duplicateMovieClip( )*» и «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13.

#endinitclip, прагма

Flash 6

завершает блок кода инициализации клипа

```
#endinitclip
```

Описание

Прагма `#endinitclip` завершает специальный блок кода (начинающийся с `#initclip`), инициализирующий символ клипа, прежде чем на его основе будут созданы какие-либо экземпляры. Она может находиться только в кадре 1 символа клипа (расположение в другом месте вызывает ошибку компиляции), и ей должна предшествовать прагма `#initclip`. Все подробности находятся в описании `#initclip`.

См. также

`#initclip`

escape(), глобальная функция

Flash 5; во Flash 6 введена поддержка UTF-8

кодирует строку для надежной передачи по сети

escape(*string*)

Аргументы

string

Строка (или выражение, возвращающее строку), которую необходимо закодировать.

Возвращаемое значение

Закодированная версия *string*.

Описание

Функция *escape( )* создает на основе переданной ей строки новую закодированную строку. Вместо любых символов строки, отличных от цифр и базовых латинских

букв от A до Z и от a до z без акцентов, в новую строку записываются шестнадцатеричные escape-последовательности. Во Flash Player 6 замещающие шестнадцатеричные escape-последовательности имеют следующий формат:

- Если `System.useCodepage` имеет значение `false` (по умолчанию), то символ заменяется одной или несколькими шестнадцатеричными escape-последовательностями вида `%xx`, представляющими код символа в UTF-8. Символы Unicode в диапазоне 0–127 кодируются в виде `%xx` (один байт), символы в диапазоне 128–2047 кодируются как `%xx%xx` (два байта), а символы с 2048 и выше, кодируются как `%xx%xx%xx` (три байта).
- Если `System.useCodepage` имеет значение `true` и язык системы не является азиатским двухбайтовым, то символ заменяется шестнадцатеричной escape-последовательностью `%xx`, где `xx` – шестнадцатеричная кодовая последовательность для этого символа. Кодовая позиция определяется согласно кодовой странице системы (на большинстве западных компьютеров это обычно Latin 1). Учтите, что на Макинтошах системная кодовая страница называется Mac Roman, но Flash использует вместо нее Latin 1.
- Если `System.useCodepage` имеет значение `true` и язык системы является азиатским двухбайтовым, то символ заменяется шестнадцатеричной escape-последовательностью `%xx` или `%xx%xx` в соответствии с двухбайтовой кодировкой символов, поддерживаемой системой (например, Shift-JIS).

Flash Player 5 действует так, как если бы `System.useCodepage` имела значение `true` (ранние версии не поддерживали *escape( )*).

Функция *escape( )* иногда применяется при отправке информации из фильма Flash приложениям на сервере или записи cookies в броузере. Для раскодирования закодированной строки применяется глобальная функция *unescape( )*.

Применение

Учтите, что для символов, расположенных выше кодовой позиции Unicode U+007F (десятичное 127), escape-последовательность UTF-8 не совпадает с числом кодовой позиции Unicode! Например, кодовой позицией для символа й является U+00E9 (десятичное 233), но последовательностью UTF-8 будет %C3%A9. Сведения о загрузке закодированного текста из внешних источников приведены в описании класса *LoadVars*.

Пример

```
trace(escape("6-5 is 4"));  // Выводит: 6%2D5%20is%204  (- is %2D, space is %20)
trace(escape("resume"));    // Выводит: resum%C3%A9
```

См. также *unescape( )*; приложение B

eval(), глобальная функция

Flash 4; недопустима в левой части присваивания во Flash 6

преобразовать строку в идентификатор

`eval(`*`stringExpression`*`)`

Аргументы

stringExpression

Строковое выражение, совпадающее с некоторым идентификатором. Может быть квалифицированным (например, «_root.speed») или неквалифицированным (например, «speed»).

Возвращаемые значения

Значение переменной, представленной *stringExpression*, или ссылка на объект, клип или функцию, представленные *stringExpression*. Если *stringExpression* не представляет существующей идентификатор, то возвращается значение undefined.

Описание

Функция *eval( )* предоставляет способ динамического создания идентификатора по строке текста. Она ищет переменную, определяемую *stringExpression*, и возвращает ее значение. Например, воспользуемся значением, возвращаемым *eval( )*, чтобы присвоить значение переменной currentName:

```
var name1 = "Kathy";
var count = 1;
// Найти name1 и возвратить ее значение, "Kathy", присвоив переменной currentName
var currentName = eval("name" + count);
```

А вот пример управления клипом с динамически создаваемым именем star1:

```
var count = 1;
eval("star" + count)._x += 100;     // Переместить star1 вправо на 100 пикселов
```

Во Flash 4 и 5 (но не Flash 6) можно использовать вызов *eval( )* вместо идентификатора в левой части выражения присваивания, например:

```
var count = 4;
eval("name" + count) = "Simone";    // Присваивает name1 значение "Simone" во Flash 4 и 5
```

Однако во Flash 6 *eval( )* не может находиться в левой части присваивания (в соответствии со спецификацией ECMA-262). Для динамического создания левой части в выражении присваивания следует применять оператор обращения к свойству [] или инструкцию *set*:

```
// Использование []
var count = 4;
this["name" + count] = "Simone";      // Присваивает name4 значение "Simone"
trace (name4);

// Использование set
var count = 4;
set("name" + count, "Simone");        // Тоже присваивает name4 значение "Simone"
trace (name4);
```

Обратите внимание, что, в отличие от своего родственника JavaScript, ActionScript не позволяет компилировать и выполнять с помощью *eval( )* произвольные блоки кода. Полная поддержка *eval( )* потребовала бы включить компилятор ActionScript в проигрыватель, из-за чего размер последнего слишком бы возрос.

Применение

Начиная с Flash 5 *eval( )* редко требуется для динамического доступа к переменным. При работе с несколькими элементами, содержащими данные, лучше использовать массивы и объекты.

Пример

Функция *eval( )* часто применяется для преобразования строкового свойства *MovieClip.*_droptarget или значения, возвращаемого *Selection.getFocus( )*, в ссылку на объект. В следующем примере предполагается, что есть ряд клипов с лицами мультпер-

сонажей. Когда пользователь опускает клип food на одно из этих лиц, с посредством свойства _droptarget извлекается путь к этому лицу, а затем *eval( )* возвращает ссылку на объект, соответствующий этому лицу. Наконец, в кадр отправляется лицо, рот которого набит едой:

```
// Преобразовать строку _droptarget в ссылку
theFace = eval(food._droptarget);
// Выбрать нужный клип лица с помощью полученной ссылки
theFace.gotoAndStop("full");
```

См. также «Выполнение кода в строке с помощью *eval( )*» в главе 4

_focusrect, глобальное свойство

Flash 4

выделенное состояние, используемое для кнопок, активизируемых с клавиатуры — **чтение/запись**

_focusrect

Описание

Когда курсор мыши находится во Flash над кнопкой, показывается содержимое состояния кнопки *Over*. Кнопки могут получать фокус ввода с клавиатуры и при нажатии пользователем клавиши Tab. Когда кнопка получает фокус ввода с клавиатуры, Flash помещает вокруг этой кнопки желтый прямоугольник, что не всегда желательно по эстетическим соображениям. Отключить показ этого желтого прямоугольника можно с помощью глобального свойства _focusrect, например:

```
_focusrect = false;
```

Если _focusrect имеет значение false, Flash выводит состояние *Over* для кнопок, получивших фокус ввода с клавиатуры, или кадр с меткой _over для клипов, получивших фокус. Когда _focusrect имеет значение true (по умолчанию), Flash выводит желтый прямоугольник дополнительно к отображению состояния *Over* (или кадра _over).

Обратите также внимание, что _focusrect можно устанавливать для отдельных кнопок или экземпляров клипов с помощью *Button*._focusrect и *MovieClip*._focusrect.

См. также

Button._focusrect, *MovieClip*.focusEnabled, *MovieClip*._focusrect, *MovieClip*.tabEnabled, *Selection.getFocus( )*, *Selection.setFocus( )*

fscommand(), глобальная функция

Flash 3 и выше (усилена во Flash 5 включением trapallkeys)

отправляет сообщение автономному проигрывателю или хост-приложению (обычно веб-броузеру)

fscommand(*command*, *parameters*)

Аргументы

command

Строка, передаваемая хост-приложению, часто имя функции JavaScript.

parameters

Строка, передаваемая хост-приложению, часто аргумент функции, указанной в *command*.

Описание

С помощью функции *fscommand( )* фильм Flash может связываться с автономным проигрывателем или хост-приложением проигрывателя – средой, в которой выполняется Flash Player (обычно веб-броузером, но это может быть также Macromedia Director или приложение, способное содержать элементы ActiveX). Функция *fscommand( )* обычно используется для:

- Отправки одной из ограниченного множества встроенных команд автономному проигрывателю Flash.
- Отправки команд в веб-броузер интерпретатору JavaScript или VBScript.
- Связи с Lingo в фильме Macromedia Director.

В случае применения с автономным проигрывателем *fscommand( )* принимает одну из встроенных пар *команда/аргумент*, как показано в табл. С.6. Обратите внимание, что во Flash 6 следует предпочесть использовать свойства `Stage.scaleMode` и `Stage.showMenu`, а не вызов *fscommand( )* с командами `"allowscale"` и `"showmenu"`.

Таблица С.6. Пары команда/аргумент в автономном проигрывателе

Команда	Аргумент	Описание
`"allowscale"`	`"true"` или `"false"`	Аргумент `"false"` предотвращает увеличение или сжатие содержимого фильма в зависимости от размера окна проигрывателя. `"allowscale"` часто используется в сочетании с `"fullscreen"`, чтобы создать автономный проигрыватель, занимающий весь экран при сохранении исходного размера фильма. См. также `Stage.scaleMode`
`"exec"`	`"application_name"`	Запускает внешнее приложение (действует только в автономном проигрывателе). Путь к приложению задается в строке *application_name*. Во Flash MX приложение *должно* находиться в папке с именем "*fscommand*" в каталоге, где находится файл *.swf* (если файл *.swf* находится в */demo/*, то запускаемое приложение должно находиться в */demo/fscommand/*). Например, для запуска someApp.exe применяется команда "*fscommand("exec", "someApp.exe")*" без указания имени вложенной папки */fscommand/*. Попытки выполнить «exec», когда исполняемый модуль находится в другом каталоге, будут безрезультатными. Во Flash 5 запускаемое приложение может находиться в любом месте диска, с которого был загружен фильм. Детали формирования путей для Flash 5 Projector см. в *http://www.macromedia.com/support/flash/ts/documents/fscommand_projectors.htm*
`"fullscreen"`	`"true"` или `"false"`	Если аргумент равен `"true"`, то окно проигрывателя раскрывается полностью, занимая весь экран
`"quit"`	нет	Закрывает фильм и выходит из автономного плеера
`"showmenu"`	`"true"` или `"false"`	Аргумент `"false"` подавляет вывод управляющих элементов в контекстном меню проигрывателя, оставляя минимум опций. Доступ к контекстному меню – щелчок правой кнопкой мыши в Windows и щелчок кнопкой мыши при нажатой клавише Ctrl на Mac. Во Flash 6 предпочтительнее пользоваться `Stage.showMenu`

Команда	Аргумент	Описание
"trapallkeys"	"true" или "false"	Если аргумент равен "true", то все нажатия клавиш – даже клавиш сокращенного набора – посылаются фильму Flash. "trapallkeys" применяется для отключения управляющих клавиш в автономном проигрывателе (например, Ctrl-F или Command-F для полноэкранного режима, Ctrl-Q или Command-Q для завершения работы, Esc для остановки/выхода из полноэкранного режима и т. д.). Добавлена к *fscommand* во Flash 5

При использовании в файле *.swf*, воспроизводимом в броузере, функция *fscommand( )* вызывает конкретную функцию JavaScript (в Netscape) или VBScript (в Internet Explorer) на странице HTML, содержащей фильм. Имя этой специальной функции имеет общий вид *movieID_DoFSCommand*, где *movieID* является именем, указанным в атрибуте ID тега <OBJECT> (в Internet Explorer) или атрибуте EMBED тега <NAME> (в Netscape). В приведенном ниже примере тег <OBJECT> имеет значением атрибута ID "testmovie", поэтому Internet Explorer выполняет функцию VBScript *testmovie_DoFSCommand( )*, когда Flash Player вызывает *fscommand( )*.

При вызове *movieID_DoFSCommand( )* значения command и parameters в *fscommand( )* передаются в качестве аргументов функции *movieID_DoFSCommand( )*. Если на странице HTML, содержащей фильм, отсутствует функция *movieID_DoFSCommand()*, то происходит отказ *fscommand( )* без каких-либо сообщений. Подробный рассказ о том, как написать функцию *movieID_DoFSCommand()* на JavaScript или VBScript, выходит за рамки этой книги, но образец такой функции есть в приводимом примере.

Вызовы *fscommand( )* не будут действовать, если на странице HTML тег <OBJECT> не содержит атрибута ID или тег <EMBED> не содержит атрибута NAME (последний не следует путать с не относящимися к делу атрибутами NAME в тегах <OBJECT>). По умолчанию эти атрибуты не включаются в страницу, но их легко добавить в среде разработки, выбрав пункт меню File → Publish Settings → HTML → Template → Flash with FSCommand. Образцы тегов <OBJECT> и <EMBED> есть в примере. Обратите внимание, что для работы *fscommand( )* с Netscape атрибут SWLIVECONNECT тега HTML <EMBED> должен иметь значение "true" (и не забудьте атрибут NAME, который тоже необходим), как показано в приводимом примере.

Применение

Связь с броузером через *fscommand( ) невозможна* в следующих конфигурациях системы:

- Internet Explorer под Macintosh OS
- Любая версия Netscape 6 ниже 6.2 (*fscommand( )* в Netscape 6.2 и более поздних требует как минимум Flash Player 6.0.40.0)
- Любой броузер на Macintosh серии 68K
- Любой броузер под Windows 3.1

Заметим, что *fscommand( )* не всегда дает лучший способ связи с фильмом Director из Flash. Предпочтительным средством коммуникации с Macromedia Director является функция *getURL( )* с протоколом event: или lingo:. За подробностями обратитесь к описанию функции *getURL( )* или следующим техническим замечаниям Macromedia:

http://www.macromedia.com/support/director/ts/documents/flash_xtra_sending_events.htm

http://www.macromedia.com/support/director/ts/documents/flash_xtra_lingo.htm
http://www.macromedia.com/support/director/ts/documents/flash_tips.htm

Пример

Выход из автономного проигрывателя (проектора):

```
fscommand("quit");
```

Запуск автономного проигрывателя в полноэкранном режиме:

```
fscommand("fullscreen", "true");
```

Запуск автономного проигрывателя в полноэкранном режиме с сохранением размера исходного фильма:

```
fscommand("fullscreen", "true");
fscommand("allowscale", "false");
```

Информацию о запуске фильма в полноэкранном окне веб-броузера можно получить по адресу:

http://www.moock.org/webdesign/flash/launchwindow/fullscreen.

В следующем коде показана страница HTML с командами JavaScript и VBScript, необходимыми для ответа на простой вызов *fscommand()* в фильме. Обратите внимание, что функция VBScript просто вызывает функцию JavaScript; это позволяет работать как в Internet Explorer, так и в Netscape с помощью одной функции JavaScript. Обратите внимание на атрибуты `ID`, `NAME` и `SWLIVECONNECT`, выделенные полужирным шрифтом.

```
<HTML>
<HEAD>
<TITLE>fscommand demo</TITLE>

<SCRIPT LANGUAGE="JavaScript">
<!--
function testmovie_DoFSCommand(command, args) {
  alert("Here's the Flash message " + command + ", " + args);
}
//-->
</SCRIPT>

<SCRIPT LANGUAGE="VBScript">
<!--
Sub testmovie_FSCommand(ByVal command, ByVal args)
    call testmovie_DoFSCommand(command, args)
end sub
//-->
</SCRIPT>

</HEAD>

<BODY BGCOLOR="#FFFFFF">

<OBJECT
  ID="testmovie"
  CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  WIDTH="100%"
  HEIGHT="100%"
  CODEBASE="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab">
```

```
    <PARAM NAME="MOVIE" VALUE="flash-to-javascript.swf">

    <EMBED
      NAME="testmovie"
      SRC="flash-to-javascript.swf"
      WIDTH="100%"
      HEIGHT="100%"
      SWLIVECONNECT="true"
      PLUGINSPAGE="http://www.macromedia.com/go/flashplayer/"
    </EMBED>
  </OBJECT>

  </BODY>
  </HTML>
```

Предшествующую функцию JavaScript *testmovie_DoFSCommand( )* из фильма *flash-to javascript.swf*, можно вызвать так:

```
fscommand("hello", "world");
```

См. также

Дополнительные сведения об *fscommand( )* и управлении Flash с помощью JavaScript можно найти по адресам:

http://www.moock.org/webdesign/flash/fscommand
http://www.macromedia.com/support/flash/publishexport/scriptingwithflash

Function, класс

Flash 6

объектно-ориентированное представление функций ActionScript

Свойства

prototype	Объект, задающий методы и свойства для класса.

Методы

apply( )	Вызвать функцию как метод объекта, передав параметры в массиве.
call( )	Вызвать функцию как метод объекта.
toString( )	Возвращает строковое представление объекта функции.

Описание

В ActionScript каждая функция, как определенная пользователем, так и встроенная, представляет собой экземпляр класса *Function*. Будучи объектом, каждая функция может обладать свойствами и методами, и каждую функцию можно сохранять, передавать и сравнивать как данные ActionScript. Хотя представление функций в виде объектов некоторым программистам может показаться непривычным (и, возможно, странным), но это важная составляющая реализации объектно-ориентированного программирования в ECMA-262. В ActionScript каждая функция может, в принципе, быть использована в качестве конструктора класса. Поэтому каждая функция определяет свойство `prototype`, в котором задаются методы и свойства, наследуемые экземплярами этого класса. Кроме того, каждая функция может быть использована в качестве метода. Поэтому каждую функцию можно сохранить в качестве свойства объекта (определив ее в качестве метода для этого объекта). Это и другие применения функций в системе наследования ActionScript обсуждаются в главе 12.

Функции были объектами уже во Flash 5, но только во Flash 6 класс *Function* появился в ActionScript формально (с введением методов *apply( )* и *call( )*, предоставивших специальные средства вызова функций как методов произвольных объектов).

Некоторые команды, первоначально реализованные во Flash 4 (например, *getURL( )* и *trace( )*), действуют как функции, но фактически скомпилированы в собственные байт-коды, не являются членами класса *Function* и не обладают свойством `prototype`.

Общие сведения о создании функций и их применении в ActionScript можно найти в главе 9.

Применение

Обратите внимание, что в ActionScript новые объекты *Function* нельзя создавать посредством конструктора *Function*. Все функции должны определяться с помощью предложений *function* или литералов функций. Например, следующий код допустим в JavaScript, но не в ActionScript:

```
var divide = new Function ("numerator", "divisor", "return numerator/divisor;");
```

См. также Объект *Arguments*; глава 9, глава 12

Function.apply(), метод

Flash 6

вызвать функцию как метод объекта, передав параметры в массиве

theFunction.apply(*thisObj, parametersArray*)

Аргументы

thisObj

Объект, для которого *theFunction* вызывается как метод.

parametersArray

Массив значений, передаваемых в качестве аргументов *theFunction*.

Возвращаемое значение

Значение, возвращаемое *theFunction*.

Описание

Метод *apply( )* вызывает функцию *theFunction* как метод объекта *thisObj* и передает значения массива *parametersArray* в *theFunction* как аргументы. Внутри *theFunction* значением ключевого слова `this` является ссылка на *thisObj*. Метод возвращает значение, возвращаемое функцией *theFunction*. Для вызова *theFunction* без указания ссылки на объект в качестве значения *thisObj* нужно задать `null`. Чтобы передать в функцию *theFunction* список аргументов, разделенных запятыми, а не массив значений *parametersArray*, используется *Function.call( )*.

Метод *apply( )* обычно применяется, чтобы изменить значение ключевого слова `this` во время выполнения функции или передать функции массив аргументов вместо списка значений, разделенных запятыми.

Следующий код демонстрирует базовый синтаксис *apply( )*:

```
// Создать функцию
function square (x) {
  return x*x;
}
```

```
// Вызвать функцию как метод _root, передав ему литерал массива,
// содержащий единственный аргумент 10.
tenSquared = square.apply(_root, [10]);
```

Практического смысла в этом примере немного: проще было бы непосредственно вызвать функцию в виде `_root.square(10);`. Метод *apply( )* оказывается удобным при необходимости динамически указывать объект, к которому должен быть применен метод, например, когда желательно использовать один метод с многими разными объектами, не относящимися к одному классу и не имеющими общего надкласса. Допустим, что имеются методы *getWidth( )* и *moveClipTo( )*, которые требуется применить к прямоугольным и круглым объектам. Вот код, который создает общий метод *Rectangle.invokeAsMethod( )*, вызывающий *getWidth( )*, *moveClipTo( )* или любую другую функцию в качестве метода объекта-прямоугольника:

```
// Класс Rect
function Rect (width, height, x, y) {
  this.width = width;
  this.height = height;
  this.x = x;
  this.y = y;
}

// Метод для вызова любой переданной func как метода объекта Rect.
// Обратите внимание, что func - ссылка на функцию, а не строка.
Rect.prototype.invokeAsMethod = function (func) {
  // Вызвать func как метод Rect, передав ему любые заданные аргументы.
  func.apply(this, arguments.slice(1));
};

// Простая общая функция getWidth, не принимающая аргументов
function getWidth () {
  trace("width: " + this.width);
  return this.width;
}

// Несколько более сложный пример, moveClipTo, которая принимает аргументы
function moveClipTo (newX, newY) {
  this.x = newX;
  this.y = newY;
}

// Создать новый объект Rect
box = new Rect(15, 20);
// Вызвать getWidth для box
box.invokeAsMethod(getWidth);  // Displays: width: 15
// Вызвать moveClipTo для box, передав аргументы newX и newY arguments
box.invokeAsMethod(moveClipTo, 100, 200);
// Проверить, что получилось...
trace("box x: " + box.x);  // Выводит: box x: 100
trace("box y: " + box.y);  // Выводит: box y: 200
```

В таком приеме нет необходимости, если можно логично назначить объектам общий класс или надкласс. В нашем примере с прямоугольником вместо использования *apply( )* лучше было бы создать надкласс геометрической фигуры *Shape* с методами *moveClipTo( )* и *getWidth( )*, которые наследовались бы как `Rect`, так и `Circle`. Более обоснованное применение метода *apply( )* см. в классе *CountDown* следующего примера.

Пример

Ниже показан класс *CountDown*, вызывающий заданный метод объекта с некоторой задержкой. В нем применяется *apply( )*, что обеспечивает вызов метода с нужным объектом и передачу методу аргументов в виде массива.

```
/*
 * CountDown класс
 *  Описание: Вызывает метод после паузы в delay миллисекунд.
 * Параметры: obj  Объект, метод которого будет вызван.
 *           meth  Строка с именем вызываемого метода.
 *          delay  Задержка (в миллисекундах) перед вызовом метода.
 * arg4...argn  Аргументы, передаваемые методу meth.
 */
function CountDown (obj, meth, delay) {
  // Запомнить длительность паузы перед вызовом meth.
  this.delay = delay;
  // Сохранить ссылку на метод.
  this.callback = obj[meth];
  // Сохранить ссылку на объект.
  this.obj = obj;
  // Сохранить дополнительные аргументы. Мы передадим их потом meth.
  this.args = arguments.slice(3);
  // Задать пустой intervalID для setInterval().
  this.intervalID = -1;
  // Начать отсчет времени
  this.start();
}

/*
 * Метод: start()
 */
CountDown.prototype.start = function () {
  // Выполнить вызов метода через this.delay миллисекунд.
  this.intervalID = setInterval(this, "invokeCallback", this.delay);
};

/*
 * Метод: stop()
 */
CountDown.prototype.stop = function () {
  // Сбросить интервал, чтобы не вызывать метод повторно.
  clearInterval(this.intervalID);
  this.intervalID = -1;
};

/*
 * Метод: invokeCallback()
 */
CountDown.prototype.invokeCallback = function () {
  // Время вышло! Прекратить отсчет.
  this.stop();
  // Вызвать callback как метод this.obj, передав
  // this.args для использования в качестве аргументов метода.
  this.callback.apply(this.obj, this.args);
};

// ОБРАЗЕЦ ПРИМЕНЕНИЯ
// Создать объект countdown, который вызовет
```

```
// _root.output("hello", "world") через 1000 миллисекунд.
count = new CountDown(_root, "output", 1000, "hello", "world");

// Вызываемый метод
function output (arg1, arg2) {
  trace("output: " + arg1 + ", " + arg2);
}
```

См. также *Function.call( )*, объект *Arguments*; глава 12

Function.call(), метод

Flash 6

вызвает функцию в качестве метода объекта

theFunction.call(*thisObj, param1, param2,...paramn*)

Аргументы

thisObj

Объект, для которого *theFunction* вызывается как метод.

param1, ...*paramn*

Список значений, передаваемых в качестве аргументов *theFunction*.

Возвращаемое значение

Значение, возвращаемое *theFunction*.

Описание

Метод *call( )* вызывает функцию *theFunction* как метод объекта *thisObj* и передает значения *param1*, *param2*, ...*paramn* в *theFunction* как аргументы. Функционально он совпадает с методом *apply( )* за исключением того, что аргументы *theFunction* задаются в виде списка значений, разделяемых запятыми, а не как массив. Подробнее см. в *Function.apply( )*.

Пример

```
function moveClipTo (newX, newY) {
  this._x = newX;
  this._y = newY;
}
moveClipTo.call(ball_mc, 100, 200);  // Перемещает ball_mc в точку 100, 200
```

См. также *Function.apply( )*, объект *Arguments*; глава 12

Function.prototype, свойство

Flash 5

Объект, задающий методы и свойства класса **чтение/запись**

theFunction.prototype

Описание

Свойство prototype представляет собой сердцевину системы наследования классов и объектов в ActionScript. По умолчанию это объект общего типа, готовый к тому, что ему будут назначены свойства и методы, которые станут доступными объектам, создаваемым с помощью *theFunction*. Как всякий общий объект, *theFunction*.prototype начинается со встроенных свойств __proto__ и constructor. ActionScript автоматически обес-

печивает ссылку *theFunction*.prototype.constructor на *theFunction* (во всех других случаях свойство конструктора объекта изначально является ссылкой на собственную функцию конструктора, которой для объекта общего типа будет *Object*, а не *theFunction*).

Объект prototype можно также заменить новым объектом, что обычно происходит при задании надкласса для класса.

Сам класс *Function* также является функцией, поэтому он также определяет свойство prototype, с помощью которого можно добавлять методы и свойства ко всем функциям. Например, следующий код добавляет во все функции методы *getName( )* и *setName( )*, а также переопределяет существующий по умолчанию метод *Function.toString( )*:

```
Function.prototype.getName = function () {
  return this.funcName;
};
Function.prototype.setName = function (name) {
  this.funcName = name;
};
Function.prototype.toString = Function.prototype.getName;

// Применение:
function squareNum (x) {
  return x * x;
}

squareNum.setName("squareNum");
trace(squareNum);  // Выводит: squareNum (вместо [type Function])
```

Полное описание роли свойства prototype в объектно-ориентированном программировании ActionScript см. в главе 12. Среди опытных разработчиков свойство prototype служит предметом оживленной дискуссии по поводу назначения надклассов. См. раздел «Более сложные вопросы стандартного назначения надклассов» в главе 12.

См. также *Object*.constructor, *Object*.__proto__; глава 12

Function.toString(), метод

Flash 5

возвращает строковое представление объекта функции

theFunction.toString()

Возвращаемое значение

Строковое представление объекта функции.

Описание

Метод *toString( )* возвращает строку, описывающую *theFunction*. По умолчанию все функции возвращают строку «[type Function]». Проверка значения, возвращаемого *toString( )*, может оказаться полезной при отладке, поскольку позволяет отличить объекты функций от других объектов. Операторы *instanceof* и *typeof* также могут быть использованы для определения типа данных объекта *Function*. Например:

```
function moveClipTo (newX, newY) {
  this._x = newX;
  this._y = newY;
}
trace(moveClipTo.toString());           // Выводит: [type Function]
trace(typeof moveClipTo);               // Выводит: function
trace(moveClipTo instanceof Function);  // Выводит: true
```

Чтобы сделать значение, возвращаемое *toString( )*, более информативным, можно переопределить метод *toString( )* для функции:

```
moveClipTo.toString = function () {
  return "moveClipTo function";
};

trace(moveClipTo.toString());  // Выводит: moveClipTo function
```

См. также *instanceof*, *Object.toString( )*, *typeof*

getProperty(), глобальная функция

Flash 4; во Flash 5 устарела

возвращает значение свойства клипа

getProperty(*movieClip*, *property*)

Аргументы

movieClip

Выражение, возвращающее строку, которая указывает путь к клипу. Во Flash 5 это может быть и ссылка на клип, потому что ссылки на клип в строковом контексте преобразуются в пути.

property

Имя извлекаемого встроенного свойства. Должно быть идентификатором, а не строкой (то есть _x, а не "_x").

Возвращаемое значение

Значение *property* для *movieClip*.

Описание

Функция *getProperty( )* извлекает значение какого-либо встроенного свойства клипа. Хотя *getProperty( )* представляла единственный способ доступа к свойствам объекта во Flash 4, во Flash 5 и выше предпочтительным инструментом доступа к свойствам служат операторы «точка» и [].

Пример

В каждом из следующих вызовов *getProperty( )* извлекается значение свойства _x клипа с именем ball на главной временной диаграмме:

```
getProperty(ball, _x);              // Относительная ссылка на клип
getProperty(_root.ball, _x);        // Абсолютная ссылка на клип
getProperty("ball", _x);            // Относительный путь в строке
getProperty("_root.ball", _x);      // Путь в строке через точку
getProperty("/ball", _x);           // Путь в строке через косую черту
```

В следующем коде показан доступ к тому же свойству с помощью предпочтительных операторов «точка» и []:

```
ball._x;
_root.ball._x;
ball["_x"];
_root["ball"]["_x"];
```

См. также *setProperty( )*; «„Объектность“ клипов» в главе 13

getTimer(), глобальная функция

Flash 4

определить продолжительность работы проигрывателя в миллисекундах

```
getTimer()
```

Возвращаемое значение

Количество миллисекунд, истекших с момента запуска проигрывателя.

Описание

Функция *getTimer( )* показывает продолжительность работы проигрывателя в миллисекундах. Функция *getTimer( )* возвращает единственное значение для каждого экземпляра проигрывателя во всех загруженных в него фильмах. При загрузке нового файла *.swf* это значение не сбрасывается. До версии Flash 6 существовала практика многократного вызова *getTimer( )* для управления выполнением блока кода во времени или добавления в фильм функций, зависящих от времени, скажем, обратного отсчета в видеоигре. Во Flash 6 выполнение кода по таймеру следует организовывать с помощью *setInterval( )*.

Следующий код демонстрирует функцию таймера *doCountdown( )*, выводящую на экран количество оставшихся секунд. Более сложный класс *CountDown*, использующий *setInterval( )*, см. в описании *Function.apply( )*.

```
function doCountdown (countAmount) {
  // Зафиксировать начальное время в миллисекундах
  var startTime = getTimer();
  // Инициализировать переменную для хранения длительности истекшего времени
  var elapsed = 0;
  // Создать текстовое поле для показа оставшегося времени в секундах
  this.createTextField("counterOutput_txt", 1, 0, 0, 200, 20);

  // В каждом следующем кадре...
  this.onEnterFrame = function () {
    // ...проверить, сколько времени прошло
    elapsed = getTimer() - startTime;
    // Если истекло меньше времени, чем установлено...
    if (elapsed < countAmount * 1000) {
      // ...показать в текстовом поле, сколько секунд осталось
      this.counterOutput_txt.text = countAmount - Math.floor(elapsed / 1000);
    } else {
      // ...если время истекло, сообщить об этом пользователю
      this.counterOutput_txt.text = "Time's Up!";
      // Прекратить проверку времени
      delete this.onEnterFrame;
    }
  }
}

// Применение:
doCountDown(60);
```

Пример

Продолжительность воспроизведения фильма в целых секундах, а не миллисекундах можно узнать, разделив значение, возвращенное *getTimer( )*, на 1000 и убрав дробную часть с помощью *Math.floor( )*, *Math.round( )* или *Math.ceil( )*. Например:

```
numSeconds = Math.floor(getTimer()/1000);
```

Следующий код циклически воспроизводит два кадра, пока продолжительность воспроизведения фильма не превысит 10 секунд, после чего цикл прекращается:

```
now = getTimer();
if (now > 10000) {
  play();
} else {
  gotoAndPlay(_currentframe - 1);
}
```

Следующий код то же самое выполняет с помощью *setInterval( )*:

```
stop();
intID = setInterval(function () { play(); clearInterval(intID); }, 1000);
```

См. также *clearInterval( )*, *Date( )*, класс *Date*, *setInterval( )*

getURL(), глобальная функция

Flash 2; усилена во Flash 4 добавлением параметра method

загружает документ в броузер, выполняет сценарии на стороне сервера или генерирует события Macromedia Director

```
getURL(url)
getURL(url, window)
getURL(url, window, method)
```

Аргументы

url

Строка, задающая абсолютное или относительное местонахождение документа, который надо загрузить, или внешнего сценария, который надо выполнить, или команды (в зависимости от протокола, как показано в табл. C.7).

window

Необязательная строка, задающая имя окна броузера или фрейма, в который должен быть загружен документ. Может быть пользовательским именем или одним из четырех предустановленных: `"_blank"`, `"_parent"`, `"_self"` или `"_top"`.

method

Необязательная строка, задающая метод, с помощью которого переменные текущей временной диаграммы пересылаются во внешний сценарий: `"GET"` или `"POST"`. Этот параметр должен быть литералом, а не переменной или выражением. Автономная версия Flash Player в Windows всегда использует метод `"GET"` независимо от того, что указано в *method*. Поддержка GET и POST подробно рассмотрена в приложении F.

Описание

Функция *getURL( )* применяется для:

- Загрузки документа (обычно веб-страницы) во фрейм или окно броузера.
- Выполнения сценария на стороне сервера и получения результата в фрейм или окно броузера.
- Выполнения кода JavaScript в веб-броузере.
- Генерирования событий от элементов Flash, импортированных в Director как спрайты.

Для того чтобы загрузить документ в текущее окно или фрейм, просто укажите URL документа (аргумент *url*), не передавая аргументы *window* или *method*. Flash поддерживает абсолютные URL (такие, в которых указаны протокол, например «*http:*», а также имя сервера или аппаратного устройства) и относительные URL (те, которые заданы относительно текущего адреса):

```
getURL("http://www.moock.org/");                  // Абсолютный URL веб-страницы
getURL("file:///C:/WINDOWS/Desktop/index.html");  // Абсолютный URL локального файла
getURL("/whatever/index.html");                   // Относительный URL,
                                                  // предполагается протокол http
```

Чтобы загрузить документ в окно или фрейм с некоторым именем, укажите это имя в качестве аргумента *window*. Например:

```
getURL("http://www.moock.org/", "contentFrame");  // Загрузить в именованный фрейм
getURL("http://www.moock.org/", "remoteWin");     // Загрузить в именованное окно
```

Чтобы заменить набор фреймов, в котором содержится текущий фильм, дайте аргументу *window* значение "_parent", например:

```
getURL("http://www.moock.org/", "_parent");
```

Чтобы заменить загружаемым документом *все* наборы фреймов на веб-странице, дайте аргументу *window* значение "_top", например:

```
getURL("http://www.moock.org/", "_top");
```

Чтобы открыть загружаемый документ в новом анонимном окне броузера, дайте аргументу *window* значение "_blank", например:

```
getURL("http://www.moock.org/", "_blank");
```

Обратите внимание, что открытие нового окна с помощью "_blank" лишает нас всякого контроля над его внешним видом (размером, конфигурацией панелей управления, местоположением и т. д.). Чтобы открывать настраиваемые окна с помощью *getURL( )*, надо вызвать на содержащей фильм странице функцию JavaScript. Приемы открытия окон с помощью JavaScript описаны на странице

http://www.moock.org/webdesign/flash

До появления Flash 6 функция *getURL( )* применялась для пересылки переменных удаленным приложениям или сценариям на сервере и получения результатов в броузере. Во Flash 6 для этого следует предпочесть метод *LoadVars.send( )*. Для того чтобы передать переменные текущей временной диаграммы во внешний сценарий с помощью *getURL( )*, надо указать имя сценария в качестве аргумента *url* и задать "GET" или "POST" в качестве значения аргумента *method*. Например:

```
getURL("http://www.someserver.com/cgi-bin/search.pl", "resultsFrame", "GET");
```

При вызове в качестве метода клипа *getURL( )* посылает удаленному сценарию переменные временной диаграммы этого клипа, например:

```
// Передает переменные myClip в search.pl
myClip.getURL("http://www.server.com/cgi-bin/search.pl", "resultsFrame", "GET");
```

Результаты выполнения сценария появятся в окне или фрейме, указанных в аргументе *window* (который необходимо задавать при передаче переменных).

Для того чтобы загрузить результаты выполнения сценария в текущие фрейм или окно, надо задать "_self" в качестве значения аргумента *window*:

```
getURL("http://www.someserver.com/cgi-bin/search.pl", "_self", "GET");
```

Если аргумент *method* имеет значение "GET", то переменные временной диаграммы текущего клипа посылаются в виде строки запроса, присоединенной к URL сценария в запросе HTTP GET. Строки запросов состоят из пар имя/значение, разделяемых амперсандами (&). Например:

```
http://www.someserver.com/cgi-bin/search.pl?term=javascript&scope=entiresite
```

Если аргумент *method* равен "POST", переменные временной диаграммы текущего клипа посылаются сценарию в виде отдельного блока данных после заголовка запроса HTTP POST (так же, как в обычной форме HTML, использующей метод POST). Обратите внимание, что "POST" недоступен в автономном Flash Player под Windows.

Многие веб-серверы ограничивают максимальную допустимую длину URL величиной от 255 до 1024 символов, поэтому при передаче больших объемов данных следует применять "POST", а не "GET".

Обратите внимание, что любые данные, возвращаемые сценарием, который вызван *getURL( )*, выводятся как обычное веб-содержимое в броузере, а не во Flash. Чтобы получить результаты выполнения сценария во Flash, используйте *loadVariables( )* или лучше класс *LoadVars*. О загрузке файлов *.swf* см. описание функции *loadMovie( )*.

Функция *getURL( )* может применяться не только с http:, но и с другими протоколами, как показано в табл. C.7.

Таблица C.7. Протоколы, поддерживаемые getURL

Протокол	Формат	Назначение
event	«event: *eventName params*»	Послать событие в Director, если в качестве элемента Flash выступает спрайт Director
file	«file:///driveSpec/folder/filename»	Обратиться к локальному файлу
ftp	«ftp://server.domain.com/folder/filename»	Обратиться к удаленному файлу по FTP (протокол передачи файлов)
http	«http://server.domain.com/folder/filename»	Обратиться к удаленному файлу по HTTP (протокол передачи гипертекста)
javascript	«javascript: *command*»	Выполнить в броузере команду JavaScript
lingo	«lingo: *command*»	Выполнить команду Lingo, если элемент Flash является спрайтом Director
print	«print:», *"targetClip"*	Печать указанного клипа. Во Flash 5 и выше следует использовать функцию *print( )*
vbscript	«vbscript: *command*»	Выполнить в броузере команду VBScript
mailto	«mailto:user@somedomain.com»	Отправить сообщение через почтовую программу, действующую по умолчанию в системе пользователя
telnet	«telnet://domain.com:8888/»	Подключение к domain.com по telnet на порту 8888

Как видно из табл. C.7, если элемент Flash импортируется в файл Macromedia Director, с помощью *getURL( )* можно генерировать события Lingo или выполнять команды Lingo. (Lingo – это язык сценариев в Director, родственный ActionScript.) Например, можно добавить событие кадра такого вида:

```
getURL ("event: eventName params");
```

что приведет к вызову в Director обработчика события Lingo с именем *on eventName*. Ниже показана команда *getURL( )*, которая генерирует событие с именем "myEvent" и передает ему строку "A". Обратите внимание на преобразование символа " с помощью \":

```
getURL ("event: myEvent \"A\"");  // Отправить событие в Director
```

Вот обработчик события Lingo *sprite*, который должен быть прикреплен к спрайту элемента Flash в Director, чтобы получить событие. Заметим, что спрайт в Director примерно соответствует экземпляру клипа во Flash; ключевое слово Lingo *put* родственно команде ActionScript *trace( )*, а && в Lingo является оператором конкатенации строк:

```
on myEvent msg
  put "The message received from Flash was " && msg
end
```

Можно также запустить Lingo из спрайта Flash в Director с помощью ключевого слова "lingo:":

```
getURL ("lingo: beep");  // Пусть Director даст звуковой сигнал
```

Использование элементов Flash в Macromedia Director требует надлежащей версии подключаемого к Director модуля под названием *Flash Asset Xtra*, бесплатно распространяемого Macromedia. Director 8.5 содержит Flash Asset Xtra для файлов *.swf* Flash 5, но не может импортировать файлы *.swf* Flash 6. Обновленный модуль Flash 6 (Flash MX) Asset Xtra поставляется с Director MX (то есть Director 9), и его также можно взять с сайта Macromedia. Flash MX Asset Xtra действует также с Shockwave 8.5.1, но требует версии Mac OS 8.6 или более поздней либо Windows 98 или более поздней.

Наконец, с помощью *getURL( )* можно выполнять код JavaScript. Вызовем простую команду *alert* JavaScript с помощью *getURL( )*:

```
getURL ("javascript: alert('hello world');");
```

Заметим, что код JavaScript не выполняется из URL в Internet Explorer 4.5 на Макинтошах.

Пример

Вот код стандартной кнопки со ссылкой на веб-страницу:

```
link_btn.onRelease = function () {
  getURL("http://www.macromedia.com/");
}
```

Ошибки реализации

Internet Explorer 4.5 (IE 4.5) и более старые версии для Макинтошей не поддерживают метод "POST" в вызове *getURL( )*. Для обслуживания этих броузеров следует применять "GET" вместо "POST" (что влечет ограничения на длину, о которых говорилось выше).

В большинстве броузеров относительные ссылки в *getURL( )* разрешаются относительно файла HTML, содержащего файл *.swf*. В IE 4.5 и более старых версиях для компьютеров Макинтош относительные ссылки разрешаются относительно расположения файла *.swf*, а не файла HTML, что приводит к ошибкам, если они находятся в разных каталогах. Для того чтобы обойти это препятствие, располагайте файлы *.swf* и *.html* в одном каталоге или указывайте при вызове *getURL( )* абсолютные URL:

```
getURL ("http://www.someserver.com/")
```

См. также

loadMovie( ), *loadVariables( )*, класс *LoadVars*, *MovieClip.getURL( )*, *MovieClip.loadVariables( )*; «<A> (якорь или гипертекстовая ссылка)» в приложении E

getVersion(), глобальная функция — Flash 5

возращает платформу и номер версии Flash Player

```
getVersion()
```

Возвращаемое значение

Строка с данными о платформе и версии для Flash Player, в котором находится текущий фильм.

Описание

Функция *getVersion( )* сообщает о платформе и версии Flash Player, в котором воспроизводится фильм. Она может применяться для условного выполнения различного кода для определенных версий Flash Player или в определенных операционных системах. Начиная с Flash 6 вместо *getVersion( )* лучше применять свойство `capabilities.version`. Однако для фильмов, воспроизводимых во Flash Player 5, надо по-прежнему использовать *getVersion( )*, а во Flash 4 следует работать с глобальным свойством `$version`.

Сведения о значениях, возвращаемых *getVersion( )*, и пример кода можно найти в описании `capabilities.version`.

См. также

`capabilities.version`, `$version`; приложение C

_global, объект — Flash 6

контейнер для глобальных переменных

```
_global.variableName
```

Описание

Объект *_global* позволяет создавать глобальные переменные (то есть переменные, доступные из временной диаграммы любого клипа или уровня). В отличие от JavaScript, в ActionScript нет «глобального контекста выполнения», в котором определения переменных автоматически создают глобальные переменные. Областью видимости всех нелокальных переменных, создаваемых в ActionScript, служит клип, в котором они созданы. Для того чтобы сделать переменную непосредственно доступной во всех временных диаграммах фильма, надо добавить в `_global` соответствующее свойство. Например:

```
// Добавить свойство age в объект _global. В результате будет создана
// глобальная переменная age.
```

```
_global.age = 31;
// Обращение к age из любой временной диаграммы:
trace(age);  // Выводит: 31
```

С помощью глобальных переменных можно сделать функцию, свойство или объект доступными глобально подобно встроенным классам и объектам (*Math*, *TextField*, *Mouse* и т. д.). Так, сделать глобальным класс *CountDown* можно при помощи такого кода (см. полный текст *CountDown* в описании *Function.apply( )*):

```
_global.CountDown = function (obj, meth, delay) {
  // ... Здесь находится код класса
};
```

После этого из любой временной диаграммы можно непосредственно создавать экземпляры *CountDown*:

```
theTimer = new CountDown(game, "timesUp", 10000);
```

Глобальные переменные не становятся свойствами всех объектов. Например, нельзя обращаться к глобальной переменной `age` как к свойству объекта *Date*:

```
_global.age = 31;  // Создать глобальную переменную
now = new Date();  // Создать объект Date
trace(now.age);    // age не является свойством now. Выводит: undefined
```

Аналогично к глобальным переменным нельзя обращаться в клипах с помощью точечной нотации. Например, к глобальной переменной `password` нельзя обращаться как к `form_mc.password` (где `form_mc` — клип), а на временной диаграмме `form_mc` нельзя обращаться к `password` в виде `this.password`. Доступ к глобальной переменной `password` может быть осуществлен только в виде `_global.password` или прямо как `password` (без квалифицирующей ссылки на объект) из текущей временной диаграммы.

Присваивание значения переменной временной диаграммы с таким же именем, как у глобальной переменной, перекрывает (но не изменяет!) значение глобальной переменной. Например, если в клипе с именем `player1_mc` переменной `age` присваивается значение 10, то внутри `player1_mc` переменная `age` будет содержать 10:

```
// КОД НА ВРЕМЕННОЙ ДИАГРАММЕ player1_mc
_global.age = 31;  // Создать глобальную переменную с именем age
var age = 10;      // Создать переменную временной диаграммы с таким же именем age
trace(age);        // Выводит: 10
```

Однако в любой другой временной диаграмме глобальная переменная `age` будет попрежнему содержать 31. Фактически, если удалить переменную временной диаграммы `age` из `player1_mc`, то ссылки на `age` будут осуществлять обращение к глобальной переменной:

```
delete age;
trace(age);        // Выводит: 31
```

Для того чтобы присвоить или изменить значение глобальной переменной, нужна полностью квалифицированная ссылка на свойство `_global`, например:

```
_global.age = 21;  // Эта модификация age видна в каждой временной диаграмме,
                   // в которой не задано собственное значение для age.
```

Подобным же образом локальные переменные функций перекрывают глобальные:

```
_global.age = 21;  // Присвоить значение глобальной переменной
```

```
function showAge () {
  var age = 5;     // Присвоить значение локальной переменной
  trace(age);
}
showAge();         // Выводит: 5
                   // Локальная переменная перекрывает глобальную
trace(age);        // Выводит: 21
                   // Глобальная переменная продолжает существовать вне функции
```

Дополнительные сведения о перекрытии переменных приведены в разделе «Функции и цепочка областей видимости» главы 9. Для удаления глобальной переменной применяется оператор *delete*, например:

```
delete _global.age;  // Удаляет age. (Ссылка на _global обязательна.)
```

При работе с глобальными переменными следите за тем, чтобы их имена были уникальными, особенно если фильмы будут использоваться другими разработчиками или загружаться в их фильмы. С этой целью можно задавать для переменных префиксы в виде доменного имени, например:

```
_global.moockorgGravity = 9.8;
```

Либо, чтобы облегчить чтение, создайте объект, в котором будут храниться все глобальные переменные:

```
_global.moockorg = new Object();
_global.moockorg.gravity = 9.8;
```

Разработчики на Java, возможно, предпочтут такой код:

```
// Создать пространство имен.
if (!_global.org) {
  _global.org = new Object();
}
if (!_global.org.moock) {
  _global.org.moock = new Object();
}

// Присвоить значение переменной в пространстве имен.
_global.org.moock.gravity = 9.8;
```

Для простоты обнаружения глобальных переменных можно добавлять к их именам специальный суффикс, например `_gl`:

```
_global.moockorgGravity_gl = 9.8;
```

Ошибки реализации

Во Flash 6 нельзя непосредственно добавить в объект `_global` так называемое «свойство доступа». Подробности см. в описании *Object.addProperty( )*.

См. также Глава 2; «Цепочка областей видимости» в главе 9

gotoAndPlay(), глобальная функция — Flash 2

переместить головку воспроизведения в заданный кадр, затем воспроизвести текущий клип или фильм

```
gotoAndPlay(frameNumber)
gotoAndPlay(frameLabel)
```

```
gotoAndPlay(scene, frameNumber)
gotoAndPlay(scene, frameLabel)
```

Аргументы

frameNumber

Положительное целое число, задающее номер кадра, в который должна переместиться головка воспроизведения текущей временной диаграммы перед воспроизведением. Если `frameNumber` меньше 1 или больше количества кадров на временной диаграмме, то головка воспроизведения направляется в первый или последний кадры соответственно.

frameLabel

Строка, задающая метку кадра, в который головка воспроизведения текущей временной диаграммы должна переместиться перед воспроизведением. Если `frameLabel` не найдена, головка воспроизведения направляется в первый кадр текущей сцены.

scene

Необязательная строка, задающая имя сцены, содержащей заданные `frameNumber` или `frameLabel`. Если этот аргумент не задан, предполагается текущая сцена.

Описание

При вызове без аргумента `scene` функция *gotoAndPlay( )* отправляет головку воспроизведения текущей временной диаграммы в кадр, заданный `frameNumber` или `frameLabel`, и воспроизводит временную диаграмму с этого места. «Текущей временной диаграммой» служит клип или главный фильм, из которого вызвана функция *gotoAndPlay( )*.

При переходе в другой кадр все промежуточные кадры пропускаются, а их сценарии не выполняются.

Если в вызове функции *gotoAndPlay( )* указаны два аргумента, предполагается, что первый из них – `scene`. Если задан только один аргумент, предполагается, что это `frameNumber` или `frameLabel`, а сцена – текущая.

При вызове с аргументом `scene` функция *gotoAndPlay( )* перемещает головку воспроизведения в указанный номером или меткой кадр заданной сцены и воспроизводит эту сцену. При этом функция *gotoAndPlay( )* должна вызываться из временной диаграммы `_root`. В противном случае операция не выполняется и головка воспроизведения не перемещается в указанный кадр. Обратите внимание, что сцены выравниваются в одну временную диаграмму во время воспроизведения фильма. Это значит, что если временная диаграмма сцены 1 содержит 20 кадров, а временная диаграмма сцены 2 содержит 10 кадров, то направить головку воспроизведения в кадр 5 сцены 2 можно, вызвав `gotoAndPlay(25)`.

Не рекомендуется применять сцены при работе с фильмами, активно использующими ActionScript. В отличие от клипов, сцены не представляются объектами, и ими нельзя непосредственно управлять с помощью большинства встроенных функций. Как правило, лучше использовать метки и клипы в качестве псевдосцен на временной диаграмме вместо сцен Flash.

Глобальная функция *gotoAndPlay( )* действует только на текущую временную диаграмму. На кадры или состояние других клипов текущей временной диаграммы она

не оказывает влияния. Воспроизведение других клипов достигается вызовом отдельных команд *play( )* или *gotoAndPlay( )* для каждого клипа, включая вложенные. Чтобы применить функцию *gotoAndPlay( )* к клипу, отличному от текущего, необходимо обратиться к формату метода клипа: *MovieClip.gotoAndPlay( )*.

Ошибки реализации

Во Flash Player 5 и Flash Player 6 функция *gotoAndPlay( )* не работает при вызове из обработчика события *onClipEvent( )* с заданием строкового литерала в аргументе `frameLabel`.

Во Flash Player 4 и последующих при задании аргумента `frameLabel` не в виде литерала в любой сцене, кроме первой, головка воспроизведения перескакивает к указанной метке *плюс* столько кадров, сколько их во всех сценах перед текущей.

Обе эти ошибки можно обойти, вызвав *gotoAndPlay( )* как метод клипа – *MovieClip.gotoAndPlay( )* и задав текущий клип при помощи ключевого слова `this` – `this.`*gotoAndPlay("myLabel")* – вместо *gotoAndPlay("myLabel")*.

Пример

```
// Перейти в кадр 5 текущей временной диаграммы и воспроизвести его
gotoAndPlay(5);
// Перейти в кадр 10 сцены exitSequence и воспроизвести его
gotoAndPlay("exitSequence", 10);
// Перейти в кадр "goodbye" сцены exitSequence и воспроизвести его
gotoAndPlay("exitSequence", "goodbye");
// Внимание! Будет воспроизведен кадр "exitSequence" текущей сцены.
gotoAndPlay("exitSequence");
// Воспроизвести кадр 1 сцены exitSequence
gotoAndPlay("exitSequence", 1);
```

Упражнение. Напишите функцию, рекурсивно воспроизводящую все клипы, вложенные внутрь клипа, из которого вызывается *gotoAndPlay( )*. (Совет: воспользуйтесь циклом for-in для получения списка дочерних клипов, вложенных в каждый клип.)

См. также *gotoAndStop( )*, *MovieClip.gotoAndPlay( )*, *play( )*, *stop( )*

gotoAndStop(), глобальная функция

Flash 2

перемещает головку воспроизведения в указанный кадр и останавливает текущий клип или фильм

```
gotoAndStop(frameNumber)
gotoAndStop(frameLabel)
gotoAndStop(scene, frameNumber)
gotoAndStop(scene, frameLabel)
```

Аргументы

frameNumber

Положительное целое число, задающее номер кадра, в который должна переместиться головка воспроизведения текущей временной диаграммы. Если `frameNumber` меньше 1 или больше количества кадров на временной диаграмме, головка воспроизведения направляется в первый или последний кадры соответственно.

frameLabel

Строка, задающая метку кадра, в который должна переместиться головка воспроизведения текущей временной диаграммы. Если `frameLabel` не найдена, головка воспроизведения направляется в первый кадр текущей сцены.

scene

Необязательная строка, задающая имя сцены, содержащей заданные `frameNumber` или `frameLabel`. Если не задана, предполагается текущая сцена.

Описание

При вызове без аргумента `scene` функция *gotoAndStop( )* отправляет головку воспроизведения текущей временной диаграммы в кадр, заданный аргументами `frameNumber` или `frameLabel`. «Текущей временной диаграммой» служит фильм или клип, из которого вызвана функция *gotoAndStop( )*. Головка воспроизведения останавливается в целевом кадре: переместившись туда, она не начнет автоматически перемещаться вперед.

Если в вызове функции *gotoAndStop( )* указаны два аргумента, предполагается, что первый из них – это `scene`. Если задан только один аргумент, предполагается, что это `frameNumber` или `frameLabel`, а сцена – текущая.

При вызове с аргументом `scene` функция *gotoAndStop( )* перемещает головку воспроизведения в указанный номером или меткой кадр заданной сцены и прекращает воспроизведение. При этом функция *gotoAndStop( ) должна* вызываться из временной диаграммы `_root`, в противном случае операция не выполняется и головка воспроизведения не перемещается в указанный кадр. Обратите внимание, что сцены выравниваются в одну временную диаграмму во время воспроизведения.

Глобальная функция *gotoAndStop( )* действует только на текущую временную диаграмму. На кадры или состояние других клипов текущей временной диаграммы она не оказывает влияния. Для того чтобы перемещались головки воспроизведения других клипов, необходимо вызывать отдельные команды *gotoAndStop( )* в каждом клипе. А если надо применить функцию *gotoAndStop( )* к клипу, находящемуся вне текущего клипа, следует вызвать ее как метод клипа: *MovieClip.gotoAndStop( )*.

Ошибки реализации

Во Flash Player 5 и 6 функция *gotoAndStop( )* не работает при вызове из обработчика события *onClipEvent( )* с заданием строкового литерала в параметре `frameLabel`.

Во Flash Player 4 и последующих при задании аргумента `frameLabel` не в виде литерала в любой сцене, кроме первой, головка воспроизведения перескакивает к указанной метке *плюс* столько кадров, сколько их во всех сценах перед текущей.

Обеих этих ошибок можно избежать, вызвав *gotoAndStop( )* как метод клипа и задав посредством ключевого слова `this` текущий клип, то есть `this.`*gotoAndStop("myLabel")*, а не *gotoAndStop("myLabel")*.

Пример

```
// Перейти в кадр 5 текущей временной диаграммы и остановиться там
gotoAndStop(5);
// Перейти в кадр 20 сцены introSequence и остановиться там
gotoAndStop("introSequence", 20);
// Перейти в кадр "hello" сцены introSequence и остановиться там
gotoAndStop("introSequence", "hello")
// Внимание! Будет сделан переход в кадр "introSequence" текущей сцены
gotoAndStop("introSequence")
```

```
// Перейти в кадр 1 сцены introSequence
gotoAndStop("introSequence", 1)
```

Упражнение. Напишите пользовательскую функцию *gotoDeltaAndStop( )*, позволяющую задать относительное количество пропускаемых при переходе кадров, включая отрицательное. Например, при вызове *gotoDeltaAndStop(-5)* должен происходить переход на 5 кадров назад. (Совет: вам потребуется свойство `MovieClip._currentframe`.) Не забудьте добавить специальный код для обработки граничных состояний (то есть таких, когда относительный переход вызвал бы перемещение дальше начала или конца текущей временной диаграммы).

См. также *gotoAndPlay( )*, *MovieClip.gotoAndStop( )*, *play( )*, *stop( )*

_highquality, глобальное свойство

Flash 4; во Flash 5 заменено свойством _quality

качество вывода Flash Player **чтение/запись**

`_highquality`

Описание

Глобальное свойство `_highquality` содержит целое число от 0 до 2, следующим образом управляющее качеством вывода Flash Player:

0 Низкое качество. Не выполняется сглаживание (antialiasing) ни для растровой, ни для векторной графики.

1 Высокое качество. Сглаживается векторная графика. Растровая графика сглаживается, когда нет анимации.

2 Высшее качество. И растровая, и векторная графика сглаживается всегда.

Начиная с Flash 5 `_highquality` заменено свойством `_quality`, позволяющим устанавливать «среднее», «автоматическое низкое» и «автоматическое высокое» качество, дополнительно к «низкому», «высокому» и «высшему».

См. также `_quality`, *toggleHighQuality( )*

#include, директива

Flash 5

импорт текста из внешнего файла ActionScript

`#include` *path*

Аргументы

path Строка, задающая имя и местонахождение импортируемого файла сценария либо относительно файла *.fla*, либо как абсолютный путь (см. примеры ниже). Обратите внимание, что элементы пути должна отделять друг от друга прямая косая черта, а не обратная. Файлы сценариев должны иметь расширение *.as*.

Описание

Директива `#include` загружает в текущий сценарий текст сценария из внешнего текстового файла (предпочтительно с расширением *.as*), помещая его прямо в то место, где находится команда `#include`. Если указанного в *path* файла нет, директива генерирует ошибку, и никакой внешний текст не включается.

Операция #include выполняется на этапе компиляции, и это означает, что включаемый в фильм текст существовал во время тестирования, экспорта или публикации из среды разработки. Изменения, внесенные во внешний файл после экспорта фильма, не будут отражены в фильме. Чтобы добавить эти изменения в фильм, его надо экспортировать повторно.

Включение внешнего файла с помощью директивы #include не изменяет содержимое файла *.fla* и не облегчает просмотр содержимого импортируемого файла. Кроме того, если внешний файл не будет найден или его содержимое изменится, то при очередной компиляции файла *.fla* в файл *.swf* код может неожиданно перестать работать. Чтобы зафиксировать импорт кода из внешнего фала *.as* и разорвать связь с самим внешним файлом *.as*, надо поместить текст в текущий файл *.fla* с помощью операций копирования и вставки (и убрать директиву #include).

Обратите внимание, что директива #include выполняется всегда – даже если находится в условной инструкции. Так, следующий код не удалит *theScript.as* из фильма:

```
if (false) {
   #include "theScript.as"
 }
```

Директива #include обычно применяется для включения одного и того же блока кода в несколько сценариев или в разных проектах Flash (подобно тому, как используется внешняя библиотека с элементами). Это следует делать для централизации кода, сопровождения кода в системе контроля версий (например, CVS или Microsoft Visual Source Safe) либо при работе с внешним текстовым редактором, который вам нравится больше, чем редактор ActionScript. Это удобно и тогда, когда программист работает отдельно от, скажем, разработчика графических анимаций Flash. Внешние файлы легко держать в хранилищах кода, например в библиотеках классов или функций, независимых от текущей временной диаграммы или клипа. Менее полезными они оказываются в коде, требующем тесной интеграции с временной диаграммой Flash.

Для того чтобы использовать во включаемом файле международные символы, сохраните его в формате UTF-8 из текстового редактора и добавьте в начало файла следующую строку (обратите внимание на символы обычной косой черты и пробела после двух дефисов):

```
//!-- UTF8
```

Директива #include не поддерживает файлы в кодировке UTF-16.

Поддерживаются вложенные включения, поэтому можно создать специальный файл *.as*, позволяющий включать группу связанных между собой классов с помощью одной директивы, аналогично пакетам Java.

Текст включаемого файла участвует в проверке синтаксиса, осуществляемой по команде Check Syntax (Ctrl-T или Command-T) в меню панели Actions (вызываемом кнопкой со стрелкой в правом верхнем углу панели).

Применение

Обратите внимание, что директива #include начинается с символа (#), в ней нет скобок и она *не* должна заканчиваться точкой с запятой. Любая инструкция #include, за-

вершающаяся точкой с запятой, вызовет ошибку и не выполнит импорта внешнего сценария.

Ошибки реализации

Во Flash 5 директивы `#include` необходимо отделять друг от друга пустой строкой, которую можно разместить между самими директивами в исходном файле или в тексте включаемых файлов. Это известная ошибка, характерная только для Flash 5. Например:

```
// Так можно.
#include "file1.as"

#include "file2.as"

// Во Flash 5 так можно, только если в конце file1.as есть пустая строка.
#include "file1.as"
#include "file2.as"
```

Во Flash 5 и Flash MX, если последняя строка включаемого файла представляет собою комментарий, файлы последующих директив `#include` не будут включены (их закомментирует завершающий комментарий предыдущего включенного файла). Например:

```
// НЕВЕРНО!
doSomething();  // Этот комментарий отключит последующие включаемые файлы.
// ПРАВИЛЬНО
// Такой комментарий вполне безопасен.
doSomething();
```

Пример

Следующий код импортирует внешний файл *.as* с именем *myScript.as* в текущий файл *.fla*. Если задан относительный путь, то *myScript.as* должен быть в той же папке, что и файл *.fla*, содержащий директиву `#include`:

```
#include "myScript.as"
```

Можно построить относительный путь, содержащий подкаталог. В следующем коде предполагается, что *myScript.as* находится на один уровень ниже текущего файла *.fla* в подкаталоге с именем *includes*:

```
#include "includes/myScript.as"
```

Папка, расположенная над текущей, обозначается двумя точками. В следующем примере предполагается, что *myScript. as* находится на один уровень *выше* текущего файла *.fla*:

```
#include "../myScript.as"
```

В следующем коде предполагается, что *myScript.as* находится в каталоге *includes*, на том же уровне, что и подкаталог, содержащий текущий файл *.fla*:

```
#include "../includes/myScript.as"
```

Можно задать и абсолютный путь к любой папке, например:

```
#include "C:/WINDOWS/Desktop/myScript.as"
```

Однако абсолютные пути не одинаковы для разных платформ и могут потребовать изменения при компиляции файла *.fla* на другой машине с другими каталогами. Обратите внимание на разницу в задании буквы диска:

```
#include "C:/WINDOWS/Desktop/myScript.as"                // Windows
#include "Mac HD/Desktop folder/working/myScript.as"     // Macintosh
```

См. также «Размещение кода ActionScript во внешних файлах» в главе 16

Infinity, глобальное свойство

Flash 5

константа, представляющая бесконечно большое число **только для чтения**

Infinity

Описание

В ActionScript любое число, превышающее максимально допустимое, представляется числовой константой Infinity (хотя недопустимые в расчетах числа представляются константой NaN). Самое большое допустимое значение в ActionScript представляется числом Number.MAX_VALUE (1.79769313486231e+308). Превышение максимально допустимого числа называется переполнением и обычно вызывается математической ошибкой.

Пример

Результатом вычисления, превосходящим максимально допустимое число, является Infinity. Например:

```
Number.MAX_VALUE * 2;    // Дает Infinity
```

Infinity получается также при делении положительного числа на ноль:

```
1000 / 0;                // Дает  Infinity
```

Применение

Как правило, Infinity служит сокращенной записью для Number.POSITIVE_INFINITY.

См. также

-Infinity, *isFinite( )*, NaN, Number.POSITIVE_INFINITY; «Особые значения числового типа данных» в главе 4

–Infinity, глобальное свойство

Flash 5

константа, представляющая бесконечно большое число **только для чтения**

-Infinity

Описание

В ActionScript любое число, выходящее за границы допустимого отрицательного числа, представляется числовой константой -Infinity (недопустимые в расчетах числа представляются константой NaN). Наименьшим отрицательным значением (имеющим наибольшую абсолютную величину), допустимым в ActionScript, является -Number.MAX_VALUE, эквивалентное –1. 79769313486231e+308.

Пример

Результатом вычисления, меньшим (то есть более отрицательным), чем минимальное допустимое отрицательное число, является -Infinity. Например:

```
-Number.MAX_VALUE * 2;    // Дает -Infinity
```

-Infinity получается также при делении отрицательного числа на ноль:

```
-1000 / 0;                // Дает -Infinity
```

Применение

Обычно `-Infinity` служит сокращенной записью для `Number.NEGATIVE_INFINITY`.

См. также

`Infinity`, *isFinite( )*, `NaN`, `Number.Mav_VALUE`, `Number.NEGATIVE_INFINITY`; «Особые значения числового типа данных» в главе 4

#initclip, прагма

Flash 6

начинает блок кода инициализации клипа

`#initclip` *order*

Аргументы

order

Неотрицательное целое число, задающее очередность выполнения этого блока `#initclip` относительно других блоков `#initclip` в фильме. Блоки с меньшими значениями *order* выполняются раньше, чем блоки с большими значениями. У нескольких блоков `#initclip` может быть одинаковое значение *order*.

Описание

Прагма `#initclip` открывает специальный блок кода, позволяющий инициализировать символ клипа до создания его экземпляров. Как разъяснялось в главе 14, `#initclip` обычно применяется для создания так называемых компонентов, представляющих собой экземпляры класса *MovieClip*, которые обладают свойством независимости, благодаря которому могут легко использоваться непрограммистами. Поэтому блок кода `#initclip` обычно применяется для регистрации символа клипа в классе ActionScript (то есть создания подкласса *MovieClip*, как показано в главе 14). Блок `#initclip` может находиться только в кадре 1 символа клипа (в других местах он генерирует ошибку). Он выполняется только один раз для каждого фильма в соответствии с настройками экспорта этого символа (задаваемыми с помощью Options → Linkage в панели Library):

- Если символ не экспортируется, то блок `#initclip` выполняется непосредственно перед кадром, в котором впервые появляется экземпляр символа.
- Если символ экспортируется для ActionScript и не задан параметр «Export in first frame», то блок `#initclip`, как и в предыдущем случае, выполняется перед кадром, в котором впервые появляется экземпляр символа.
- Если символ экспортируется для ActionScript и параметр «Export in first frame» задан, то блок `#initclip` выполняется перед первым кадром фильма.

Когда выполняется блок кода `#initclip`, экземпляры связанного с ним символа клипа (даже размещенные на этапе разработки) еще не созданы, поэтому код выполняется в контексте главной временной диаграммы (`_root`). Следовательно, у кода `#initclip` нет доступа к экземплярам символа, в котором он определен. Такой режим задан намеренно, поскольку `#initclip` применяется для инициализации символа в целом, а не конкретного экземпляра, и должен выполниться прежде, чем будут созданы какие-либо экземпляры.

В самом простом виде блок `#initclip` выглядит так:

```
// КОД В КАДРЕ 1 СИМВОЛА КЛИПА
#initclip
```

```
// ...код, связывающий символ клипа с классом ActionScript
#endinitclip
```

Прагма #endinitclip закрывает блок #initclip (если #endinitclip отсутствует, Flash генерирует ошибку компиляции). Следующий код демонстрирует более полный блок #initclip, помещенный в символ клипа с именем *ballSymbol*. Этот код создает пользовательский подкласс *MovieClip* с именем *Ball* и с помощью *Object.registerClass( )* связывает *ballSymbol* (в библиотеке фильма) с классом ActionScript *Ball*. Хотя блок #initclip выполняется лишь один раз, обычно в нем определяется конструктор, который будет выполняться при каждом создании экземпляра символа. Подробную информацию о подклассах клипов можно найти в главе 14.

```
// КОД В КАДРЕ 1 СИМВОЛА КЛИПА ballSymbol
#initclip

// Создать пользовательский класс ActionScript с именем Ball.
// При выполнении #initclip определяется конструктор. Конструктор выполняется каждый раз,
// когда экземпляр ballSymbol динамически или вручную помещается на сцену,
// при этом ключевое слово this указывает на этот экземпляр.
function Ball () {
  // Обратите внимание, что this.color передается через initObj в attachMovie().
  // См. детали в MovieClip.attachMovie().
  this.setRGB(this.color);
}

// Задать MovieClip в качестве надкласса Ball
Ball.prototype = new MovieClip();

// Связать класс Ball с символом библиотеки ballSymbol.
Object.registerClass("ballSymbol", Ball);

#endinitclip
```

Когда экземпляр *ballSymbol* помещается на сцену (вручную или с помощью ActionScript), выполняется конструктор *Ball*, в котором this указывает на этот экземпляр. Экземпляр наследует свойства и методы как *Ball*, так и *MovieClip*. Например:

```
// Вызвать конструктор Ball, который задаст для redBall цвет. Обратите внимание, что
// значение 0xFF0000 для color задано в методе attachMovie() параметром initObj.
this.attachMovie("ballSymbol", "redBall", 1, { color: 0xFF0000 });
// Вызвать метод MovieClip с новым экземпляром.
redBall.play();
```

Интересно, что весь код в нашем примере #initclip можно было поместить на главную временную диаграмму. Создание подкласса *MovieClip* и связывание его с символом клипа можно выполнять не только в коде #initclip. Но вследствие того, что этот код помещен непосредственно в символ клипа, мы создаем мобильный пакет, легко перемещаемый из одного файла *.fla* в другой. Чтобы воспользоваться нашим подклассом *MovieClip*, разработчику достаточно перетащить *ballSymbol* из нашей библиотеки в свою и создавать его экземпляры обычным образом. Можно еще больше облегчить непрограммистам работу с этим клипом, определив для него параметры компонента, что позволит настраивать экземпляры с помощью инспектора свойств. Дополнительную информацию о параметрах компонентов см. в главе 16 и статье Macromedia, рассказывающей о создании компонентов, по адресу:

http://www.macromedia.com/support/flash/applications/creating_comps/

Если вернуться к нашему примеру, то мы видели, что код `#initclip` выполняется в контексте главной временной диаграммы. Поэтому функция конструктора *Ball* фактически создается на главной временной диаграмме и выглядит как `_root.Ball`. Если мы знаем, что не будем создавать подклассы *Ball*, то можем даже убрать этот конструктор с главной временной диаграммы, поместив в последней строке блока `#initclip` следующий код:

```
delete Ball;
```

Конструктор, зарегистрированный для символа клипа, внутренне сохраняется во Flash Player и будет нормально выполняться даже после удаления. Однако функцию `_root.Ball` не следует удалять, если она определяет статические свойства (например, `Ball.gravity`) или если другие символы клипов используют ее как надкласс. Более того, функция *Ball* должна быть создана раньше любых своих подклассов, иначе она не сможет выступать в качестве надкласса.

Гарантировать, что надкласс, например наш класс *Ball*, будет создан прежде своих подклассов, можно с помощью параметра *order* прагмы `#initclip`, который назначает блоку кода приоритет исполнения. Несколько блоков `#initclip` могут иметь одинаковые значения *order*, но те из них, номера которых меньше, всегда выполняются перед теми, номера которых больше. Блоки, значение *order* которых не задано, всегда выполняются раньше пронумерованных блоков. Например, подкласс *BasketBall* класса *Ball* можно создать посредством такого кода:

```
// КОД В КАДРЕ 1 СИМВОЛА КЛИПА ballSymbol
// Обратите внимание на параметр order!
#initclip 1
// Создать пользовательский класс ActionScript
function Ball () {
  this.setRGB(this.color);
}

// Задать MovieClip в качестве надкласса Ball
Ball.prototype = new MovieClip();
// Связать класс Ball с символом библиотеки ballSymbol
Object.registerClass("ballSymbol", Ball);
#endinitclip

// КОД В КАДРЕ 1 СИМВОЛА КЛИПА ballSymbol
// Обратите внимание на параметр order!
#initclip 2
// Создать пользовательский класс ActionScript
function BasketBall () {
  // Вызвать конструктор надкласса
  super();
}

// Задать Ball в качестве надкласса BasketBall
BasketBall.prototype = new Ball();
// Связать класс BasketBall с символом библиотеки basketballSymbol
Object.registerClass("basketballSymbol", BasketBall);
#endinitclip
```

Поскольку функции, определенные в блоке #initclip, имеют областью видимости _root, все ссылки на переменные, свойства и методы внутри функции должны быть ясно квалифицированы.

Например, для того чтобы создать полностью квалифицированную ссылку, надо обратиться к свойству _x текущего клипа так: this._x, а если ссылка на _x будет неявной, как в следующем коде, то она будет указывать на _root._x!

```
Ball.prototype.onEnterFrame = function () {
  // НЕПРАВИЛЬНО! Перемещает _root, а не ball!
  _x++;

  // ПРАВИЛЬНО. Перемещает ball.
  this._x++;
}
```

Повторяющиеся имена классов внутри отдельных блоков #initclip не вступают в конфликт. Например, если конструктор класса *Vehicle* создается внутри двух разных блоков #initclip, каждый клип окажется связанным с собственной версией функции, но _root.Vehicle будет указывать на ту, которая была создана последней. Это предохраняет от ошибок в коде из-за конфликтов имен в различных компонентах. Однако использование одинаковых имен классов в одном фильме не рекомендуется и считается плохой практикой. Можно рекомендовать сохранить каждый класс #initclip в отдельном пространстве имен объекта _global, таком как _global.org.moock, как описано в главе 12 и демонстрируется на протяжении главы 14.

Код в блоке #initclip может не только привязывать класс к символу клипа, хотя чаще всего решает именно эту задачу. Ничто не мешает выполнять в #initclip другие функции каждый раз, когда в фильме впервые используется некоторый символ. Директива #initclip может определять произвольный блок кода, выполняемый перед тем, как символ, содержащий этот блок #initclip, используется впервые (или в начале воспроизведения фильма, если этот символ экспортируется в первом кадре фильма).

Например, в клипе, экспортируемом для ActionScript в первом кадре, удобно поместить глобальную библиотеку. В следующем коде создается класс *Timer*, который остается доступным на протяжении всего фильма. Символ клипа – это просто контейнер для кода: экземпляр символа нигде не создается.

```
// КОД В ПЕРВОМ КАДРЕ ЭКСПОРТИРУЕМОГО КЛИПА
#initclip
_global.Timer = function () {
  // ...Здесь размещается код класса Timer...
};
#endinitclip

// КОД В ЛЮБОМ МЕСТЕ ВРЕМЕННОЙ ДИАГРАММЫ ФИЛЬМА
gameclock = new Timer();
```

Значительно подробнее о создании подклассов клипов рассказано в главе 14.

См. также

#endinitclip, *MovieClip.attachMovie( )*, *MovieClip.duplicateMovieClip( )*, *Object.registerClass( )*; глава 14

instanceof, оператор

Flash 6

определяет, является ли объект экземпляром указанного класса

object instanceof *classConstructor*

Операнды

object

Любой объект.

classConstructor

Ссылка на функцию-конструктор любого класса.

Возвращаемое значение

Булево значение true, если *object* представляет собой экземпляр *classConstructor*, в противном случае возвращается false.

Применение

В операторе *instanceof* не используются скобки (это оператор, а не функция).

Пример

```
var now = new Date();
trace(now instanceof Date);  // Выводит: true
```

См. также

typeof; подзаголовок *Constructor* в описании класса *Date*; глава 12; «Оператор instanceof» в главе 5

int(), глобальная функция

Flash 4; во Flash 5 заменена аналогичными методами Math

отбросить дробную часть числа

int(*number*)

Аргументы

number

Число или выражение, возвращающее число, обычно число с дробной (десятичной) частью.

Возвращаемое значение

Целая часть *number*.

Описание

Функция *int( )* применялась во Flash 4 в качестве метода извлечения целой части числа. Фактически она округляет положительные числа вниз и отрицательные числа вверх. Функция *int( )* работает только с числами в диапазоне от -2147483648 (-2^{31}) до 2147483647 ($-2^{31}-1$). Для чисел, выходящих за этот диапазон, она возвращает undefined. Если *number* представляет собой строку, состоящую только из чисел, то *int( )* предварительно преобразует эту строку в число. Если *number* представляет собой булево значение true, *int( )* возвращает 1. Для всех остальных нечисловых данных (включая undefined и null) *int( )* возвращает значение 0.

Применение

Функция *int( )* устарела по сравнению с более точными и стандартными методами *Math.floor( )*, *Math.ceil( )* и *Math.round( )*. Преобразовывать нечисловые данные в целые или действительные числа следует при помощи *parseInt( )* или *Number( )*.

Пример

```
int(4.5)      // Дает 4
int(-4.5)     // Дает -4
int(3.999)    // Дает 3
```

Функция *int( )* позволяет проверить, является ли число целым, путем его сравнения с результатом, возвращаемым ею:

```
if (int(x) != x) {
  trace ("Пожалуйста, введите целое число");
}
```

См. также

Math.ceil( ), *Math.floor( )*, *Math.round( )*, *Number( )*, *parseFloat( )*, *parseInt( )*

isFinite(), глобальная функция

Flash 5

проверяет число на выход за границы Infinity и (Infinity

`isFinite(number)`

Аргументы

number

Любое число или выражение, дающее числовое значение.

Возвращаемое значение

Булево значение `true`, если число находится в диапазоне от `Number.MAX_VALUE` до `-Number.MAX_VALUE` включительно, и `false`, если нет. Если *number* не относится к типу данных *number*, то оно перед выполнением *isFinite( )* конвертируется в тип *number*.

Описание

Функция *isFinite( )* проверяет, находится ли число в допустимом для ActionScript диапазоне значений. То есть что *number* — число, меньшее `Infinity` и большее `-Infinity`. Выполняйте *isFinite( )* в начале кода, для правильной работы которого требуются допустимые числа.

Пример

```
if (!isFinite(x * y)) {          // Проверить, конечно ли число
  trace ("Результат слишком велик для вывода. Попробуйте еще раз.");
}
isFinite(-2342434);              // Возвращает true
isFinite(Math.PI);               // Возвращает true
isFinite(Infinity)               // Возвращает false
isFinite(-Infinity)              // Возвращает false
isFinite(NaN)                    // Возвращает false
```

См. также

-Infinity, Infinity, *isNan( )*, Number.MAX_VALUE, Number.MIN_VALUE; «Особые значения числового типа данных» в главе 4

isNaN(), глобальная функция

Flash 5

проверить равенство с особым значением NaN

isNaN(*value*)

Аргументы

value Проверяемое выражение.

Возвращаемое значение

Булево значение true, если *value* является особым числовым значением NaN, в противном случае — false.

Описание

Проверка значения на равенство специальному числовому значению NaN осуществляется с помощью функции *isNaN( )*, поскольку NaN не дает равенства самому себе в обычной проверке. Так, выражение:

```
NaN == NaN;
```

возвращает значение false. Функцию *isNaN( )* часто применяют для проверки отсутствия в выражении математической ошибки (такой, как деление нуля на ноль) или успешности преобразования величины в допустимое число. Так как *isNaN( )* возвращает true, если выражение *не* является допустимым числовым выражением, то вместе с *isNaN( )* часто применяется оператор логического отрицания (!) (то, что *не* является нечислом, есть число). Обратите внимание, что 0/0 дает NaN, но все положительные числа при делении на 0 дают Infinity, а все отрицательные числа при делении на 0 дают -Infinity.

Пример

```
// Присвоить значение
var x = "test123";
// Перед использованием x в математическом выражении проверить, является ли оно допустимым
// числом. Это удобный прием для данных, вводимых пользователем в текстовые поля,
// которые всегда являются строками.
if (!isNaN(parseFloat(x))) {
  var y = parseFloat(x) * 2;
}
```

См. также

isFinite( ), NaN; «Особые значения числового типа данных» в главе 4

Key, объект

Flash 5

определяет состояние клавиш на клавиатуре

Key.*property*
Key.*methodName()*

Свойства

В табл. С.8. перечислены свойства объекта *Key*.

Таблица C.8. Свойства кодов клавиш объекта Key

Свойство	Эквивалентный код клавиши	Свойство	Эквивалентный код клавиши
BACKSPACE	8	INSERT	45
CAPSLOCK	20	LEFT	37
CONTROL	17	PGDN	34
DELETEKEY	46	PGUP	33
DOWN	40	RIGHT	39
END	35	SHIFT	16
ENTER	13	SPACE	32
ESCAPE	27	TAB	9
HOME	36	UP	38

Методы

addListener( )

Регистрирует объект для получения событий *onKeyUp( )* и *onKeyDown( )*

getAscii( )

Возвращает ASCII-код последней нажатой клавиши

getCode( )

Возвращает код последней нажатой клавиши

isDown( )

Проверяет, нажата ли в данный момент конкретная клавиша

isToggled( )

Проверяет, активны ли клавиши Num Lock, Caps Lock или Scroll Lock

removeListener( )

Отменяет извещение о событиях для указанного приемника

События приемника

onKeyDown( ) Происходит при нажатии клавиши

onKeyUp( ) Происходит при освобождении клавиши

Описание

Объект *Key* позволяет определить, какие клавиши нажаты в данный момент и какая клавиша была отпущена последней. Его можно применять для создания интерфейсов, управляемых клавиатурой, например для игры, в которой движение космического корабля управляется клавишами со стрелками.

Поскольку клавиатуры бывают разные, создание интерфейсов, управляемых с клавиатуры, может быть затруднено. Однако если правильно выбирать средства в сценариях, можно обеспечить одинаковые условия для всех пользователей.

Есть два общих подхода для обнаружения команд, подаваемых с клавиатуры:

- Проверить, нажата ли в данный момент некоторая клавиша, можно с помощью метода *isDown( )*. Это рекомендуется в случаях, когда требуется постоянно прово-

дить опрос (то есть многократную проверку) клавиатуры, например в видеоиграх. При опросе клавиатуры *isDown( )* проверяет ее состояние в обработчике *MovieClip.onEnterFrame( )* или по таймеру, созданному *setInterval( )*.

- Узнать, какая клавиша была отпущена последней, можно с помощью методов *getCode( )* и *getAscii( )*. Этот подход рекомендуется для типичных интерфейсов, управляемых клавиатурой, в которых выполняются конкретные операции в зависимости от нажатых клавиш. Обычно эти методы вызываются из приемников событий *onKeyDown( )* или *onKeyUp( )* либо из обработчиков событий клипа *onClipEvent(keyDown)* или *onClipEvent(keyUp)*. При этом нет необходимости постоянно проверять (то есть *опрашивать*) клавиатуру, чтобы узнать, какая клавиша была нажата последней. На практике это привело бы к ошибочному повторению одной и той же операции, даже если клавиша не была нажата повторно. Это означает, что обычно следует проверять *getCode( )* и *getAscii( )* только в приемнике или в обработчике события клипа, чтобы гарантировать вызов обработчика один и только один раз для каждого нажатия клавиши.

Метод *isDown( )* – синхронный, как и метод *isToggled( )*. Оба они сообщают о текущем состоянии клавиатуры в момент их вызова. Например, *isDown( )* возвращает `false`, если рассматриваемая клавиша была освобождена раньше, чем был вызван *isDown( )*. Напротив, методы *getAscii( )* и *getCode( )* являются асинхронными. Они сообщают о последнем нажатии клавиши, даже если она уже освобождена. Поэтому обычно не следует вызывать *getAscii( )* или *getCode( )*, если событие клавиатуры не обнаружено с помощью, скажем, приемника *onKeyDown( )*. Методы *getAscii( )* и *getCode( )* будут возвращать одно и то же значение сколь угодно долго, даже если в данный момент не нажата никакая клавиша, пока не будет нажата другая.

Так называемый виртуальный *код клавиши Windows*, или просто код клавиши (*keycode*), возвращаемый *getCode( )* и требуемый *isDown( )*, является числом, представляющим физические клавиши клавиатуры, а не символы на этих клавишах или их ASCII-коды. Клавиши можно идентифицировать по их коду даже при воспроизведении фильма в разных операционных системах или при использовании клавиатурами различных языков или раскладок символов.

На большинстве клавиатур коды клавиш от A до Z такие же, как кодовые позиции (65–90) соответствующих букв в кодировке Latin 1 (и совпадают с их ASCII-кодами). Коды клавиш 0...9 тоже совпадают со значениями этих чисел в Latin 1 и ASCII-значениями этих чисел (48–57). Коды других клавиш могут не соответствовать кодовым позициям в Latin 1, хотя коды некоторых, например клавиши Tab, совпадают со значениями в ASCII. Однако многие коды небуквенных и нецифровых клавиш доступны как свойства *Key* (см. табл. C.8). Например, не надо помнить, что код клавиши Up равен 38 – достаточно прочитать свойство `Key.UP`. Следующий код проверяет, нажата ли в данный момент клавиша «стрелка вверх»:

```
if (Key.isDown(Key.UP)) {
  trace("The up arrow is being pressed");
}
```

При работе с клавишами, которые не относятся к буквенным или цифровым и не доступны в виде свойства *Key* (например, с функциональными клавишами F1, F2 и т. д.), надежнее всего быстро создать тестовый фильм и узнать код нужной клавиши, например:

```
var keyTester = new Object();
keyTester.onKeyDown = function () {
  trace(Key.getCode());
}
Key.addListener(keyTester);
```

При проверке клавиш в режиме Test Movie не забывайте отключать клавиши сокращенного набора с помощью Control → Disable Keyboard Shortcuts.

Коды клавиш перечислены в приложении B.

Применение

События приемников *onKeyDown( )* и *onKeyUp( )* были добавлены к объекту *Key* во Flash 6. В большинстве случаев предпочтительнее пользоваться ими, а не обработчиками *MovieClip.onKeyDown( )* и *MovieClip.onKeyUp( )* и обработчиками в стиле Flash 5 *onClipEvent(keyDown)* и *onClipEvent(keyUp)*.

Однако следует иметь в виду следующие опасности:

- Во Flash 6 функция *updateAfterEvent( )*, обновляющая экран в промежутках между кадрами, может применяться только с событиями *MovieClip* для мыши и клавиш и с *setInterval( )*. Поэтому для создания приложений, в которых нажатие клавиш обновляет экран мгновенно, пользуйтесь *onClipEvent(keyDown)* и *onClipEvent(keyUp)*, которые являются прямыми аналогами *Key.onKeyDown( )* и *Key.onKeyUp( )*.
- Чтобы перехватывать нажатия клавиш, когда фокус ввода находится в некотором конкретном клипе, следует применять обработчики *MovieClip.onKeyDown( )* и *MovieClip.onKeyUp( )*. Подробности см. в их описаниях.

См. также

MovieClip.onKeyDown( ), *MovieClip.onKeyUp( )*; глава 10, приложение B

Key.addListener(), метод

Flash 6

регистрирует объект для получения событий onKeyDown() и onKeyUp()

`Key.addListener(listener)`

Аргументы

listener

Объект с определенным в нем методом *onKeyDown( )* и/или *onKeyUp( )*.

Описание

Метод *addListener( )* добавляет `listener` к списку объектов, оповещаемых при возникновении событий *onKeyDown( )* и *onKeyUp( )* объекта *Key*. Объект `listener` может принадлежать любому классу, в котором определены методы с именами *onKeyDown* и/или *onKeyUp*. Эти методы могут быть не встроенными, а определенными пользователем. Они вызываются, когда происходит соответствующее событие для *Key*. Прекратить получение событий объектом-приемником позволяет метод *Key.removeListener( )*.

На каждое событие *Key* могут независимо реагировать несколько объектов. Если в *Key* зарегистрировано несколько приемников, их методы вызываются в том порядке, в котором эти приемники регистрировались.

Общие сведения о приемниках и функциях обратного вызова приведены в главе 10. Пример кода приемника *Key* можно найти в описаниях *Key.getAscii( )* и *Key.onKeyDown( )*.

См. также

Key.getAscii( ), *Key.onKeyDown( )*, *Key.onKeyUp( )*, *Key.removeListener( )*; глава 10

Key.getAscii(), метод

Flash 5

возвращает код ASCII последней нажатой клавиши

```
Key.getAscii()
```

Возвращаемое значение

Целое число, представляющее код ASCII последней нажатой клавиши.

Описание

Метод *getAscii( )* возвращает ASCII-значение последней нажатой клавиши (обратите внимание, что он не принимает аргументы). Поскольку ASCII-значение есть не у всех клавиш, *getAscii( )* обычно применяется для определения букв и цифр из набора символов Latin 1 (используемого в западноевропейских языках). В отличие от *getCode( )*, метод *getAscii( )* различает буквы верхнего и нижнего регистра. Другое отличие от *getCode( )* состоит в невозможности различить две клавиши с одинаковым значением ASCII, например клавишу 8 на основной клавиатуре и клавишу 8 – на числовой.

Код ASCII преобразуется в символ строки с помощью метода *String.fromCharCode( )*, как показано в следующем примере.

Для обнаружения нажатия определенных клавиш по их относительному физическому положению на клавиатуре, а не значению ASCII (что желательно при крестообразном расположении четырех клавиш для управления игрой), следует применять *getCode( )*. Методы *getAscii( )* и *getCode( )* обычно используются в приемнике события *onKeyDown( )*, чтобы их вызов происходил по одному и только по одному разу для каждого нажатия клавиши.

Пример

Следующий пример показывает, как отличить прописную букву *A* от строчной.

```
if (Key.getAscii() == 65) {
  trace("Uppercase A was pressed last.");
} else if (Key.getAscii() == 97) {
  trace("Lowercase a was pressed last.");
} else {
  trace("Some other key was pressed last.");
}
```

Обратите внимание, что *getCode( )* в этом примере не позволил бы отличить прописную A от строчной a, поскольку в обоих случаях код клавиши равен 65. Чтобы переписать пример для *getCode( )*, пришлось бы проверять состояние клавиши Shift методом *isDown( )* или клавиши Caps Lock методом *isToggled( )*.

В следующем примере показано обнаружение нажатий клавиш в простой игре в «виселицу». Приемник события *onKeyDown( )* определяет нажатую клавишу и добавляет ее в список предположений пользователя. Обратите внимание, что *keyListener* – общий объект, и это демонстрирует, что любой объект, в котором определены обра-

ботчики обратного вызова *onKeyDown( )* или *onKeyUp( )*, может выступать в качестве приемника. Объекты без таких обработчиков обратного вызова не извещаются о событиях клавиатуры, даже если эти объекты добавляются в список приемников событий с помощью *Key.addListener( )*.

```
keyListener = new Object();
userGuesses = new Array();
guessNum = 0;
keyListener.onKeyDown = function () {
  var lastKey = Key.getAscii();
  userGuesses[guessNum] = String.fromCharCode(lastKey);
  guessNum++;
  trace(userGuesses);
}
Key.addListener(keyListener);
```

См. также *Key.getCode( )*, *Key.isDown( )*; приложение B

Key.getCode(), метод

Flash 5

возвращает код последней нажатой клавиши

```
Key.getCode()
```

Возвращаемое значение

Целое число, представляющее код последней нажатой клавиши.

Описание

Метод *getCode( )* возвращает код последней нажатой клавиши, являющийся произвольным числом, представляющим физическое положение клавиши на клавиатуре. В операционных системах, отличных от Windows, собственная система кодов клавиш автоматически транслируется Flash в Windows-эквивалентные, поэтому *getCode( )* предоставляет независимый от платформы способ обнаружения нажатия конкретных физических клавиш. Метод *getCode( )* может также применяться для различения клавиш с одинаковыми значениями ASCII. Например, он может отличить клавишу 8 на основной клавиатуре от клавиши 8 на числовой, на что *getAscii( )* не способен. Зато *getCode( )* не умеет различить прописные и строчные буквы (он возвращает один код для букв *A* и *a*, потому что они вводятся нажатием одной и той же клавиши). Он также не видит нажатий клавиш-модификаторов, таких как Ctrl или Shift.

Многие часто используемые значения кодов клавиш доступны как свойства объекта *Key* (см. табл. C.8.). Для того чтобы определить код конкретной клавиши, обратитесь к приложению B или создайте тестер кодов клавиш сами:

1. Создайте новый документ Flash.
2. В кадр 1 поместите такой код:

```
var keyTester = new Object();
keyTester.onKeyDown = function () {
  trace(Key.getCode());
}
Key.addListener(keyTester);
```

3. Выберите Control → Test Movie.
4. Выберите Control → Disable Keyboard Shortcuts.

5. Щелкните по сцене фильма.
6. Нажмите клавишу. Ее код появится в окне Output.

Пример

В отличие от *isDown( )*, метод *getCode( )* удобен при создании интерфейсов, в которых нажатие отдельной клавиши приводит к одному прямому результату. Методы *getAscii( )* и *getCode( )* обычно используются в приемнике события *onKeyDown( )*, чтобы их вызов происходил по одному и только по одному разу для каждого нажатия клавиши.

Например, можно позволить пользователю пропустить вводную часть фильма нажатием пробела. Когда нажимается пробел, головка воспроизведения перемещается в основной интерфейс фильма (на главной временной диаграмме), например:

```
keyListener = new Object();
keyListener.onKeyDown = function () {
  if (Key.getCode() == Key.SPACE) {
    _root.gotoAndStop("mainInterface");
  }
}
Key.addListener(keyListener);
```

См. также *Button keyPress*, *Key.getAscii( )*, *Key.isDown( )*; приложение B

Key.isDown(), метод

Flash 5

проверяет, нажата ли конкретная клавиша в данный момент

`Key.isDown(keycode)`

Аргументы

keycode

Число, представляющее код проверяемой клавиши. Может быть одной из констант объекта *Key* (например, `Key.UP`, `Key.BACKSPACE`).

Возвращаемое значение

Булево значение, указывающее, что клавиша с данным кодом *keycode* нажата (`true`) или не нажата (`false`).

Описание

Метод *isDown( )* сообщает, нажата ли в данный момент клавиша с указанным кодом *keycode*, например:

```
trace(isDown(65));    // Выводит: true, если во время теста нажата клавиша "A"
```

Метод предоставляет непосредственный доступ к состоянию клавиатуры и более всего годится для систем, требующих постоянного ввода с клавиатуры или обнаружения нажатия одновременно нескольких клавиш.

Метод *isDown( )* обладает тремя преимуществами над *getCode( )* и *getAscii( )*:

- Обнаруживает состояние модифицирующих клавиш, таких как Ctrl, Command, Shift и Option, которые не генерируют событий клавиатуры. (Для определения состояния клавиш Caps Lock, Num Lock и Scroll Lock можно также использовать *isToggled( )*.)
- Обнаруживает состояние произвольных клавиш, позволяя программистам проверять одновременное нажатие нескольких клавиш.

- Обнаруживать, нажата ли клавиша в данный момент (*getCode( )* и *getAscii( )* сообщают о том, какая клавиша была нажата последней, но в данный момент она может уже быть отпущена).

В отличие от *getCode( )* и *getAscii( )*, всегда возвращающих последнюю нажатую клавишу (даже после того, как она была отпущена), *isDown( )* принимает аргумент *keycode*, задающий клавишу, которую надо проверить. Например, проверив обе клавиши `Key.UP` и `Key.RIGHT` в двух отдельных вызовах *isDown( )*, можно установить, что космический корабль в игре должен быть перемещен по диагонали. В зависимости от расположения конкретных проверяемых клавиш максимальное количество одновременно обнаруживаемых клавиш может не превышать трех.

Пример

Метод *isDown( )* обычно применяется для создания систем, подвергаемых постоянному обновлению в каждом новом кадре. В следующем коде мы поворачиваем и продвигаем космический корабль в каждом кадре, в котором нажаты соответствующие клавиши стрелок. Обратите внимание, что если надо обнаружить одновременное нажатие двух клавиш, то следует разделить инструкции *if*. В данном примере состояние клавиши правой стрелки игнорируется, если клавиша левой стрелки тоже нажата. Но независимо от этого всегда проверяется состояние клавиши «стрелка вверх» в отдельной инструкции *if*. Работающую версию этого примера космического корабля можно найти в сетевом хранилище кода:

```
// Код в клипе космического корабля
spaceship_mc.onEnterFrame = function () {
  if (Key.isDown(Key.LEFT)) {          // Стрелка влево
    this._rotation -= 10;
  } else if (Key.isDown(Key.RIGHT)) {  // Стрелка вправо
    _rotation += 10;
  }
  if (Key.isDown(Key.UP)) {            // Стрелка вверх
    this.thrust += 10;
  }
}
```

См. также

Button keyPress, *Key.getAscii( )*, *Key.getCode( )*, *Key.isToggled( )*; приложение B

Key.isToggled(), метод

Flash 5

проверяет, включены ли клавиши Caps Lock, Num Lock или Scroll Lock

`Key.isToggled(keycode)`

Аргументы

keycode

Целочисленный код клавиши; обычно код Caps Lock (20), Num Lock (144) или Scroll Lock (145). Может быть константой `Key.CapsLock`.

Возвращаемое значение

Булево значение, показывающее, включена (`true`) клавиша с кодом *keycode* или выключена (`false`).

Описание

Метод *isToggled( )* определяет состояние специальных клавиш Caps Lock, Num Lock или Scroll Lock. В отличие от других клавиш, они имеют состояние «включено» и «выключено», указывающее, действует ли функция, которую они представляют. Значение, возвращаемое методом *isToggled( )*, сообщает о том, действует функция указанной клавиши или нет. Хотя *isToggled( )* фактически работает с любым кодом клавиши, возвращаемое им значение полезно только для специальных клавиш, действующих как переключатели. Состояние других клавиш позволяет выяснить метод *isDown( )*. Обратите внимание, что *getCode( )* и *getAscii( )* обнаруживают последнюю нажатую клавишу, которая может уже быть отпущена, а потому не отражают текущее состояние клавиши.

См. также *Key.getAscii( )*, *Key.getCode( )*, *Key.isDown( )*; приложение B

Key.onKeyDown(), событие приемника

Flash 6

происходит при нажатии клавиши

`listener.onKeyDown()`

Описание

События *onKeyDown( )* и *onKeyUp( )* представляют собой клавиатурные аналоги событий *Mouse.onMouseDown( )* и *Mouse.onMouseUp( )*. В совокупности они образуют базовые инструменты, позволяющие программировать взаимодействие с клавиатурой. Событие *onKeyDown( )* может обрабатываться приемником событий, заданным `listener`, который добавлен с помощью *addListener( )*. Событие *onKeyDown( )* генерируется при нажатии клавиш на клавиатуре, включая модифицирующие клавиши типа Shift и Ctrl. Если клавиша удерживается в нажатом состоянии, *onKeyDown( )* может происходить многократно, что зависит от настроек операционной системы и клавиатуры. В отличие от события кнопки *on(keyPress)*, событие *onKeyDown( )* возникает при нажатии любой, а не только заданной клавиши.

Обработчик события *onKeyUp( )* иногда оказывается предпочтительнее, чем *onKeyDown( )*, поскольку неопытные пользователи, например дети, имеют обыкновение держать клавишу нажатой, что может генерировать множество лишних событий *keyDown* для каждого события *keyUp*. Однако обработчик *onKeyDown( )* в целом реагирует быстрее, чем *onKeyUp( )*, поэтому его следует применять, когда важна скорость реакции.

Для того чтобы перехватить (то есть обнаружить) событие *onKeyDown( )*, сначала надо присвоить функцию обратного вызова свойству `onKeyDown` некоторого объекта, например `listener`, а затем добавить `listener` в качестве приемника с помощью *Key.addListener( )*, например:

```
// Создать общий объект
keyListener = new Object();

// Присвоить функцию свойству onKeyDown объекта
keyListener.onKeyDown = function () {
  trace("Some key was pressed!");
}

// Зарегистрировать приемник в объекте Key
Key.addListener(keyListener);
```

Реагировать на событие *onKeyDown( )* могут объекты приемников любого класса и в любом количестве.

События клавиатуры генерируются только тогда, когда Flash Player находится в фокусе ввода. Пользователь должен щелкнуть по сцене фильма, чтобы активизировать обработчики нажатий клавиш. Можно заставить пользователя щелкнуть по какой-нибудь кнопке перед входом в часть фильма, которая управляется клавиатурой. Это гарантирует получение Flash Player фокуса ввода с клавиатуры.

Как можно заметить, приведенная функция *onKeyDown( )* не сообщает о том, какая клавиша была нажата. Если ожидается ввод вообще, например нажатие пользователем любой клавиши для перехода в следующий раздел фильма, нас может не интересовать, какая именно клавиша была нажата. Но иногда требуется привязать некоторое действие к конкретной клавише. Например, может быть желательно, чтобы разные клавиши поворачивали космический корабль в различных направлениях.

Обработчики *onKeyDown( )* удобны в интерфейсах навигации, рассматривающих каждое нажатие клавиши как отдельное действие, на которое должна произойти реакция (например, нажатие на клавишу пробела влечет показ следующего слайда презентации). Если безразлично, какая клавиша была нажата, просто выполните нужный код в приемнике *onKeyDown( )*. Определить, какая клавиша вызвала событие *onKeyDown*, можно с помощью метода *Key.getAscii( )* или *Key.getCode( )* внутри условного оператора *if* .

В тех случаях, когда надо узнать текущее состояние клавиатуры, или в приложениях типа игр, требующих мгновенной непрерывной реакции на нажатия клавиш, которые могут быть одновременными, лучше применять методы *Key.isDown( )* и *Key.isToggled( )*. Обычно эти методы помещают внутрь обработчика, вызываемого многократно, что позволяет проверять состояние клавиатуры, даже если система не генерирует повторные события, пока клавиши удерживаются нажатыми. Так, состояние клавиатуры можно проверить в обработчике *MovieClip.onEnterFrame( )* или с помощью интервала, созданного *setInterval( )*.

В следующем примере создается приемник *onKeyDown( )*, который сообщает ASCII-код последней нажатой клавиши:

```
keyListener = new Object();
keyListener.onKeyDown = function () {
  lastKeyPressed = String.fromCharCode(Key.getAscii());
  trace("You pressed the '" + lastKeyPressed + "' key.");
}
Key.addListener(keyListener);
```

А так можно проверить, была ли последней нажатой клавишей «стрелка вверх»:

```
keyListener = new Object();
keyListener.onKeyDown = function () {
  // Проверить, была ли последней нажатой клавишей "стрелка вверх".
  // "Стрелка вверх" представлена свойством Key.UP.
  if (Key.getCode() == Key.UP) {
    trace("The up arrow was the last key depressed");
  }
}
Key.addListener(keyListener);
```

На практике применять *Key.getAscii( )* следует тогда, когда требуется узнать символ, соответствующий нажатой клавише, например, была ли им прописная *A* или строчная *a*. Метод *Key.getCode( )* следует предпочесть, когда надо узнать, какая физическая клавиша была нажата, например, при использовании четырех соседних клавиш для навигации, независимо от символов, представляемых клавишами. Дальнейшие подробности см. в описании объекта *Key*.

Событие *onKeyDown( )* выполняется немедленно при нажатии клавиши, независимо от кода на временной диаграмме фильма. Однако изменения в фильме, осуществленные приемником, не показываются до момента очередного обновления кадра, который определяется частотой кадров фильма. К сожалению, во Flash 6 нельзя принудительно обновить экран из приемника *onKeyDown( )* при помощи *updateAfterEvent( )*, поскольку этот метод срабатывает только при вызове из события мыши или клавиатуры клипа, либо функции обратного вызова *setInterval( )*.

Обработка специальных клавиш

Для того чтобы отключить команды контекстного меню автономного проигрывателя (Standalone Player), (Open, Close, Fullscreen и т. д.), добавьте в начало своего фильма следующую строку кода:

```
fscommand("trapallkeys", "true");
```

Эта команда также отключает выход по клавише Esc из полноэкранного режима автономного проигрывателя. Перехват клавиши Esc в проигрывателе выполнит такой код:

```
keyListener.onKeyDown = function () {
  if (Key.getCode() == Key.ESCAPE) {
    // Реакция на нажатие Escape
  }
}
```

Не во всяком броузере можно перехватить клавишу Esc. Кроме того, не существует способа отключить клавишу Alt или комбинации клавиш Alt-Tab или Ctrl-Alt-Delete в Windows.

Следующий код перехватывает нажатия клавиши Tab:

```
keyListener.onKeyDown = function () {
  if (Key.getCode() == Key.TAB) {
    trace("Tab was the last key depressed");
  }
}
```

В некоторых броузерах Flash Player 5 обнаруживает клавишу Tab только внутри события кнопки *on(keyPress)*. Следующий код успешно обнаруживает клавишу Tab во всех броузерах, использующих Flash Player 5:

```
// КОД В КНОПКЕ НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
on (keyPress "<Tab>") {
  // Присвоить значение фиктивной переменной
  foo = 0;
}

// КОД В КЛИПЕ НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
// Обратите внимание, что это событие keyUp. Событие keyDown не работает
// с Flash 5, выполняющимся в Internet Explorer.
onClipEvent (keyUp) {
  if (Key.getCode() == Key.TAB) {
```

```
      // Поместим курсор в myTextField на _level0
      Selection.setFocus("_level0.myTextField");
    }
  }
```

Во Flash Player 6 клавишу Tab уверенно обнаруживает приемник *Key*. Обычно перехват клавиши Tab выполняется для перемещения точки ввода в определенное текстовое поле формы. Однако во Flash 6 можно точно управлять поведением клавиши Tab с помощью свойств `tabIndex` и `tabEnabled` классов *TextField*, *MovieClip* и *Button*, для чего не требуются приемники *Key*. Совместимый с Flash 5 пример перехода по табуляции есть по адресу:

http://moock.org/asdg/codedepot/interface-widgets/taborder.fla

Для перехвата комбинации клавиш, такой как Ctrl-G, используйте код:

```
keyListener.onKeyDown = function () {
  if (Key.isDown(Key.CONTROL) && Key.isDown(71)) {
    // Реакция на Ctrl-G. 71 - код клавиши G.
  }
}
```

Клавишу Enter (или Return) можно перехватить так:

```
keyListener.onKeyDown = function () {
  if (Key.getCode() == Key.ENTER) {
    // Реакция на нажатие Enter (например, передача формы)
  }
}
```

Применение

Эмулировать *onKeyDown( )* во Flash 5 следует посредством события клипа *onClipEvent(keyDown)*, прикрепляемого к этому клипу на этапе разработки. Например:

```
onClipEvent (keyDown) {
  trace("Some key was pressed");
}
```

См. также

Key.addListener( ), *Key.getAscii( )*, *Key.getCode( )*, *Key.isToggled( )*, *Key.onKeyUp( )*, *MovieClip.onKeyDown( )*

Key.onKeyUp(), событие приемника

Flash 6

происходит при освобождении клавиши

listener.onKeyUp()

Описание

Событие *onKeyUp( )* генерируется, когда отпускается нажатая клавиша. Иногда оно используется при программировании игр, чтобы отключить что-нибудь, включенное ранее по событию *onKeyDown( )* – в качестве классического примера можно привести движение космического корабля. Событие *onKeyUp( )* может обрабатываться приемником событий, заданным *listener*, который добавлен с помощью *addListener( )*. С помощью события *onKeyUp( )* можно также нажатием клавиши временно модифицировать некоторый режим приложения. Например, в среде разработки Flash при нажа-

тии клавиши пробела происходит временный переход в инструмент Hand, а при освобождении этой клавиши восстанавливается прежний инструмент. Аналогичный подход позволяет скрывать или показывать какие-то элементы приложения, например временные меню.

Для того чтобы обеспечить реакцию на событие *onKeyUp( )*, надо сначала присвоить функцию обратного вызова свойству `onKeyUp` некоторого объекта, например *`listener`*, а затем добавить *`listener`* в качестве приемника с помощью *Key.addListener( )*:

```
// Создать общий объектт
keyListener = new Object();

// Присвоить функцию свойству onKeyUp объекта
keyListener.onKeyUp = function () {
  trace("A key was released!");
}

// Зарегистрировать приемник в объекте Key
Key.addListener(keyListener);
```

Реагировать на событие *onKeyUp( )* могут объекты приемников в любом количестве. Как и в случае *onKeyDown( )*, для получения полезной информации в обработчике события *onKeyUp( )* обычно применяются другие методы объекта *Key*:

```
keyListener.onKeyUp = function () {
    trace("The last key released was " + Key.getAscii());
  }
}
```

События *onKeyDown( )* и *onKeyUp( )* генерируются только один раз для каждого нажатия и освобождения клавиши. При этом совершенно не имеет значения, нажата ли все еще клавиша в данный момент. С помощью этих событий можно точно обнаруживать нажатия клавиш, даже если пользователь отпускает клавишу в промежутке между кадрами, и не проверять одну и ту же клавишу дважды, если она была нажата всего один раз. В любом случае обычно в приемниках *onKeyDown( )* или *onKeyUp( )* применяются методы *Key.getCode( )* и *Key.getAscii( )*, чтобы узнать, какая клавиша была нажата последней.

Событие *onKeyUp( )* выполняется немедленно при освобождении клавиши независимо от кода на временной диаграмме фильма. Однако изменения вида фильма, осуществленные приемником, не отображаются до момента очередного обновления кадра, который определяется частотой кадров фильма. К сожалению, во Flash 6 нельзя принудительно обновить экран из приемника *onKeyUp( )* при помощи *updateAfterEvent( )*, поскольку этот метод действует только при вызове из события мыши или клавиатуры клипа, либо функции обратного вызова *setInterval( )*.

Применение

Эмулировать *onKeyUp( )* во Flash 5 можно с помощью события клипа *on(keyUp)* класса *MovieClip*, прикрепляемого к клипу на этапе разработки. Например:

```
onClipEvent (keyUp) {
  trace("Some key was released");
}
```

См. также *Key.addListener( )*, *Key.onKeyDown( )*, *MovieClip.onKeyUp( )*

Key.removeListener(), метод

Flash 6

отменить оповещение о событиях для указанного приемника

`Key.removeListener(listener)`

Аргументы

listener

Объект, ранее зарегистрированный в качестве приемника для *Key* с помощью *Key.addListener( )*

Возвращаемое значение

Булево значение `true`, если *listener* был найден и удален, и `false`, если *listener* не был зарегистрирован в качестве приемника для *Key*. Возвращаемое значение полезно при отладке кода, который удаляет приемники.

Описание

Метод *removeListener( )* прекращает прием извещений о событиях *Key* для *listener*. Вызывать его следует только после того, как *listener* зарегистрирован в качестве приемника событий посредством *addListener( )*. Объект приемника не уничтожается и может многократно подключаться и отключаться в качестве приемника.

Пример

Приемники событий следует отключать с помощью этого метода, если в задаче, которую они решают, больше нет необходимости. Например, когда в приложении, управляемом с клавиатуры, отображается модальное диалоговое окно, ввод данных с клавиатуры должен игнорироваться, пока окно диалога не будет закрыто. Отключить ввод с клавиатуры вне окна диалога можно, если удалить приемник событий клавиатуры, установленный по умолчанию для приложения, как в следующем коде:

```
// Определение метода для открытия диалогового окна из theApp
theApp.displayModalDialog = function () {
  // Отключить обычный приемник событий клавиатуры
  Key.removeListener(this.defaultKeyListener);
  // Включить временный приемник событий клавиатуры для модального диалога
  Key.addListener(this.modalKeyListener);
  // Продолжить открытие окна диалога...
}
```

См. также *Key.addListener( )*, *Key.onKeyDown( )*, *Key.onKeyUp( )*; глава 10

_levelN, глобальное свойство

Flash 3

уровень документа во Flash Player **только для чтения**

```
_level0
_level1
_level2
…
_levelN
```

Описание

Во Flash Player для одновременного воспроизведения может быть загружено несколько файлов *.swf*. Каждый загруженный *.swf* может быть помещен в клип или на

собственный «уровень» в стеке уровней документа. В свойстве _level*n* содержится ссылка на главную временную диаграмму файла *.swf*, загруженного в проигрыватель на уровень документа *n*. Каждый уровень документа представлен нумерованным свойством, например _level0, _level1, _level2 и т. д. Файл *.swf*, находящийся на уровне с большим номером, закрывает более низкие уровни, если они занимают одну и ту же часть сцены. Первый документ, загруженный в любой Flash Player, считается имеющим _level0.

Обратите внимание, что _level*n* может не совпадать с _root, так как последний указывает на главную временную диаграмму текущего фильма, а _level*n* – на главную временную диаграмму фильма, находящегося на конкретном уровне документа. Для создания глобальных переменных, доступных с любого уровня документа, используется объект _global.

Пример

Ссылка _level*n* обычно применяется для управления фильмами на других уровнях стека документов. Например, воспроизведем фильм на уровне 2:

```
_level2.play();
```

Можно также использовать _level*n* в сочетании со ссылками на клипы для управления клипами, содержащимися в фильме, на любом уровне в стеке документов. Например:

```
_level1.orderForm.titleBar.play();
```

Ссылка _level*n* может также выступать в качестве значения аргумента *target* нескольких функций, в том числе *loadMovie( )*, *unloadMovie( )*, *loadVariables( )* и *print( )*. Если уровень еще не существует, следует задать ссылку на уровень в кавычках. При отсутствии кавычек несуществующий уровень рассматривается как undefined, и может оказаться, что команда действует в текущей временной диаграмме вместо нового неопределенного уровня. Например, при выполнении следующего кода из главной временной диаграммы _level0 фильм на _level0 будет заменен новым, если _level1 еще не определен:

```
loadMovie("myMovie.swf", _level1);
```

Если нельзя гарантировать, что этот уровень уже существует, то более безопасен следующий подход:

```
loadMovie("myMovie.swf", "_level1");  // Действует, даже если _level1 не существует
```

Однако существование _level0 гарантировано всегда, поэтому ссылаться на _level0 можно без кавычек, например:

```
startDrag(_level0, true);
```

См. также

loadMovie( ), *unloadMovie( )*, _root; «Импорт внешних фильмов» и «Порядок расположения фильмов и экземпляров в стеке» в главе 13

loadMovie(), глобальная функция

Flash 4; во Flash 6 добавлены загрузка файлов .jpg и средства защиты

загрузить в проигрыватель файл .swf

loadMovie(*url*, *target*)
loadMovie(*url*, *target*, *method*)

Аргументы

url

Строка, задающая абсолютный или относительный путь к загружаемому файлу *.swf*. или *.jpg*. Во всех URL должна использоваться обычная косая черта, а абсолютные URL должны содержать ссылку на протокол в виде *http://* или *file:///*. Относительные URL разрешаются относительно URL фильма на уровне_level0 (то есть _level0._url) и по умолчанию используют протокол *http://*.

target

Строка, задающая клип или уровень документа, в которых расположится внешний файл . *swf* или *.jpg*. Может также являться ссылкой на существующий клип или уровень документа (в строковом контексте ссылки преобразуются в пути).

method

Необязательная строка, задающая метод отправки переменных внешнему сценарию. Значениями *method* могут быть "GET" и "POST". Этот параметр должен быть литералом, а не переменной или другим выражением. Под Windows автономные версии Flash Player до версии 6 всегда используют метод "GET" независимо от заданного в *method*. Во Flash 6 поддерживаются оба метода – и "GET" и "POST". См. приложение F.

Описание

Функция *loadMovie( )* представляет собою глобальный аналог метода *MovieClip.loadMovie( )*. Обе версии импортируют во Flash Player файл фильма *.swf* или графический файл *.jpg* (JPEG), находящийся по адресу *url*, но в глобальной версии с помощью параметра *target* можно задать новое местоположение загружаемого файла, тогда как версия *MovieClip* в виде метода всегда загружает файл в вызывающий клип. Если url задает изображение в формате JPEG, то это не должен быть формат с прогрессивной разверткой.

Если *target* представляет собой строку, задающую путь к клипу, или ссылку на существующий клип, то загруженный файл *.swf* или *.jpg* будет помещен в указанный клип (вызывая удаление предшествующего содержимого). Для того чтобы загрузить фильм или графику в *текущий* клип, в качестве параметра *target* следует задать пустую строку:

```
loadMovie("myMovie.swf", "")
```

Либо можно воспользоваться версией *MovieClip.loadMovie( )* с ключевым словом this:

```
this.loadMovie("myMovie.swf");
```

Если *target* – это ссылка на существующий уровень документа (например, _level2) или строка, указывающая путь к уровню документа (например, "_level2"), то файл *.swf* помещается на указанный уровень документа. Загрузка фильма на уровень _level0 очищает проигрыватель от всего содержимого на всех уровнях и помещает новый файл *.swf* или *.jpg* на уровень _level0.

Следует учитывать, что *loadMovie( )* выполняется асинхронно: фактическое выполнение операции загрузки начнется только после того, как выполнятся все сценарии в текущем кадре (если точнее, то Flash запрашивает у операционной системы внешний объект непосредственно перед обновлением экрана). После того как начнется загрузка файла *.swf* или *.jpg* в проигрыватель, за ее ходом можно наблюдать с помощью *MovieClip.getBytesLoaded( )* и *MovieClip.*_framesloaded. Подробнее о предварительной загрузке фильмов см. соответствующие описания в данном справочнике и статье:

http://design.oreilly.com/news/action_0501.html

Следите за тем, чтобы контроль хода загрузки начинался только после фактического выполнения *loadMovie( )*. Точность показаний при контроле загрузки файла *.swf* можно обеспечить, поместив в его первый кадр флаг, который покажет, что загрузка действительно началась:

```
var loadStarted = true;
```

Затем опросите значение этой переменной вне файла *.swf*, прежде чем начинать контроль хода загрузки:

```
if (target_mc.loadStarted == true) {
  // It's safe to display accurate load progress...
}
```

При вызове *loadMovie( )* можно передавать переменные, и тогда `url` обычно оказывается адресом сценария, возвращающего файл *.swf* или *.jpg* в соответствии со значениями переданных переменных. Чтобы отправить переменные при вызове *loadMovie( )*, включается аргумент `method` (со значением `"GET"` или `"POST"`). `"GET"` посылает переменные временной диаграммы текущего клипа в виде строки запроса, присоединенной к `url` сценария. `"POST"` посылает переменные временной диаграммы текущего клипа после заголовка HTTP для запроса POST. До версии Flash 6 метод `"POST"` был недоступен в автономном Flash Player, но в автономном проигрывателе Flash Player 6 он действует. Поскольку у большинства веб-серверов длина URL ограничена 1024 символами (а в некоторых случаях всего 255 символами), для передачи больших объемов данных следует применять `"POST"`, а не `"GET"`.

Вызовы *loadMovie( )*, использующие метод `"GET"`, могут передавать переменные в загруженный фильм без вмешательства сценария, если фильм предоставляется веб-сервером, а не локальной файловой системой. Например, загрузим внешний фильм *myMovie.swf* на уровень 1 стека документов проигрывателя, передав ему переменные текущей временной диаграммы:

```
loadMovie("myMovie.swf", "_level1", "GET");
```

Переменные, передаваемые в загружаемый фильм, определяются на его главной временной диаграмме. Этот прием действует только при обработке запроса *loadMovie( )* веб-сервером. Попытки применения метода `"GET"` в *loadMovie( )* для локальных файлов вызывают сообщение об ошибке «Error opening URL».

Во Flash 6 фильм из одного домена может загрузить фильм из другого домена, но не может получить доступ к его переменным, свойствам и функциям без специального разрешения. Подробности см. в описании *System.security.allowDomain( )*.

Применение

Будьте осторожны, применяя ссылки на клипы и уровни в качестве аргумента `target` функции *loadMovie( )*. Если аргумент `target` окажется `undefined` или пустой строкой, то функция *loadMovie( )* использует в качестве `target` текущую временную диаграмму (то есть заменит текущую временную диаграмму вновь загруженным контентом). Аналогично ссылки `target`, дающие пустую строку, заставляют *loadMovie( )* работать с текущей временной диаграммой. В частности, этим вызываются проблемы при загрузке фильмов на новые незанятые уровни. Рассмотрим следующий код:

```
loadMovie("myMovie.swf", _level1);
```

Если до выполнения этой инструкции объект _level1 не существовал, то этот код загрузит файл *myMovie.swf* во временную диаграмму, содержавшую инструкцию *loadMovie( )*, а не на _level1! Это препятствие можно обойти, применив *loadMovieNum( )* вместо *loadMovie( )*. Кроме того, можно использовать строку в качестве параметра *target* для *loadMovie( )*:

```
loadMovie("myMovie.swf", "_level1");
```

В этом случае уровень будет создан, если он еще не существует (только _level0 есть по умолчанию во всех фильмах). Дополнительные сведения см. в разделе «Проблемы, связанные с перекрытием методов и глобальных функций» главы 13.

Глобальная функция *loadMovie( )* замещает версию действия *Load Movie* из Flash 4 с целевым адресом.

Пример

Вот несколько примеров применения *loadMovie( )*:

```
loadMovie("myMovie.swf", "_level1");// Поместить myMovie.swf на level 1
loadMovie("myMovie.swf", "_level0");// Поместить myMovie.swf на level 0
loadMovie("myMovie.swf", "myClip"); // Поместить myMovie.swf в myClip
// Заменить содержимое проигрывателя на coolmovie.swf, задав абсолютный путь
loadMovie("http://www.yourflashsite.com/coolmovie.swf", "_level0");
// Загрузить фильм на уровень 1 с рабочего стола Windows. Обратите внимание на протокол
// file:/// и прямой слэш.
loadMovie("file:///C:/WINDOWS/Desktop/animation.swf", "_level1");
// Загрузить образ JPEG в imageHolder_mc
loadMovie("photo.jpg", "_level0.imageHolder_mc");
```

Следующий код загружает в фильм файл *.jpg* для использования его в качестве кнопки. Сначала создается клип, выполняющий функции кнопки, затем внутри этого клипа создается еще один, который будет хранить изображение. Обратите внимание на вложенность клипов. Если прикрепить обработчик *onRelease( )* к клипу, хранящему изображение, а не к внешнему клипу кнопки, то в результате загрузки файла *.jpg* этот обработчик будет уничтожен.

```
this.createEmptyMovieClip("btn_mc", 1);
this.btn_mc.createEmptyMovieClip("imgHolder_mc", 1);
this.btn_mc.onRelease = function () {
  trace("Image clicked");
}
this.btn_mc.imgHolder_mc.loadMovie("buttonPhoto.jpg");
```

Общую утилиту загрузки граф5ики с поддержкой прокрутки, масштабирования и предварительной загрузки можно найти в сетевом хранилище кода.

См. также

loadMovieNum( ), *MovieClip.loadMovie( )*, *System.security.allowDomain( )*, *unloadMovie( )*; «Импорт внешних фильмов и графики» в главе 13

loadMovieNum(), глобальная функция

Flash 3; во Flash 6 добавлены загрузка файлов .jpg и средства защиты

загрузить внешний файл .swf на уровень документа

```
loadMovieNum(URL, level)
loadMovieNum(URL, level, method)
```

Аргументы

url

Строка, задающая абсолютный или относительный путь к загружаемому файлу *.swf*. или *.jpg*. Во всех URL должна использоваться прямая косая черта, а абсолютные URL должны содержать ссылку на протокол в виде *http://* или *file:///*. Относительные URL разрешаются относительно URL фильма на уровне `_level0` (то есть `_level0._url`) и по умолчанию используют протокол *http://*.

level

Неотрицательное целое или выражение с неотрицательным целым значением, задающее уровень документа, на котором будет размещен внешний файл *.swf* или *.jpg*.

method

Необязательная строка, указывающая метод отправки переменных во внешний сценарий. Допустимыми значениями *method* являются `"GET"` и `"POST"`. Этот параметр должен быть литералом, а не переменной или другим выражением. В автономной версии Flash Player всегда применяется метод `"GET"` независимо от указанного *method*. Под Windows автономные версии Flash Player до версии 6 всегда используют метод `"GET"` независимо от заданного в *method*. Во Flash 6 поддерживаются оба метода — `"GET"` и `"POST"`. См. приложение F.

Описание

Функция *loadMovieNum( )* почти идентична *loadMovie( )* с указанием целевого уровня, но требует, чтобы целевой уровень загрузки был задан в виде числа, а не строки. Это означает, что *loadMovieNum( )* может загружать фильмы только на уровни документов, а не в клипы. Функция *loadMovieNum( )* безопаснее в работе, чем *loadMovie( )*, поскольку предотвращает непреднамеренную замену текущего клипа или фильма. Если указанный уровень не существует, он будет создан. Если указанный уровень существует, его содержимое будет заменено новым файлом *.swf* или *.jpg*. Допускается загрузка фильма на `_level2`, даже если `_level1` не создан.

Функция *loadMovieNum( )* применяется, когда требуется динамически задавать уровень для загрузки фильма, например:

```
var x = 3;
loadMovieNum("myMovie.swf", x);
```

Того же самого можно достичь путем конкатенации строк и обычной функции *loadMovie( )*:

```
loadMovie("myMovie.swf", "_level" + x);
```

Применение

Глобальная функция *loadMovieNum( )* замещает действие *Load Movie* из Flash 3, в котором допускались только номера уровней (во Flash 4 был добавлен параметр *method*).

См. также *loadMovie( )*, *MovieClip.loadMovie( )*

loadVariables(), глобальная функция

Flash 4

получить группу внешних переменных

```
loadVariables(URL, target)
loadVariables(URL, target, method)
```

Аргументы

url

Строка, задающая путь к источнику переменных – сценарию на стороне сервера, который возвращает переменные, или текстовому файлу, содержащему переменные.

target

Строка, указывающая путь к клипу или уровню документа, где будут определены загружаемые переменные. Может также являться ссылкой на клип или уровень документа (в строковом контексте ссылки преобразуются в пути).

method

Необязательная строка, указывающая метод отправки переменных внешнему сценарию. Если она задана, то переменные текущей временной диаграммы передаются сценарию, и *target* получает загруженные переменные. Если опущена, то переменные загружаются, но не отправляются. Допустимыми значениями для *method* являются "GET" и "POST". Этот параметр должен быть литералом, а не переменной или другим выражением. Под Windows в версиях, предшествующих Flash 6, в автономной версии Flash Player всегда используется метод "GET" независимо от значения *method*.

Описание

С помощью *loadVariables( )* можно импортировать в фильм переменные из текстового файла или выполняемого на сервере приложения, например сценария Perl. Однако во Flash 6 более гибкие и мощные средства импорта и экспорта переменных предоставляются классом *LoadVars*.

Областью видимости для переменных, загружаемых через *loadVariables( )*, является клип или уровень, указанные в *target*, а типом данных всегда является *string*. Для прикрепления загруженных переменных к текущей временной диаграмме надо задать в качестве значения аргумента *target* пустую строку:

```
loadVariables("myVars.txt", "");  // Загружает переменные из myVars.txt
                                  // в текущую временную диаграмму
```

Будут ли загружаемые переменные находиться в текстовом файле или они будут созданы сценарием, они должны иметь формат, соответствующий кодировке URL (см. описание класса *LoadVars*).

Текстовый файл, используемый с *loadVariables( )*, представляет собою обычный текстовый файл, содержащий переменные в кодировке URL (см. описание класса *LoadVars( )*). Для загрузки переменных из внешнего текстового файла путь к файлу указывается как аргумент *url* в вызове функции *loadVariables( )*. Например:

```
// Загрузить переменные из myVariables.txt в главную временную диаграмму фильма
loadVariables("myVariables.txt", "_root");
```

Функцию *loadVariables( )* можно также использовать со сценарием или серверным приложением, которые выводят переменные в кодировке URL. Когда сценарий посылает данные в фильм Flash в ответ на вызов функции *loadVariables( )*, он должен установить тип MIME данных как "application/x-www-form-urlencoded". Вот обычная команда установки MIME в сценарии Perl:

```
print "Content-type: application/x-www-form-urlencoded\n\n";
```

Хотя название *loadVariables( )* предполагает передачу переменных только в одном направлении, эта функция может также *передавать* переменные в сценарий на стороне сервера. Чтобы отправить в сценарий все переменные, определенные в текущей временной диаграмме, аргумент `method` в вызове функции *loadVariables( )* устанавливается равным `"GET"` или `"POST"`. Переменные пересылаются в формате кодировки URL. Если `method` установлен как `"GET"`, переменные добавляются к строке запроса в `url` и пересылаются удаленному сценарию. Если `method` установлен как `"POST"`, переменные посылаются после заголовка HTTP POST. До версии Flash 6 метод `"POST"` был недоступен в автономном Flash Player, но в автономном проигрывателе Flash Player 6 он действует. У большинства веб-серверов длина URL ограничена 1024 символами (а в некоторых случаях всего 255 символами), поэтому для передачи больших объемов данных следует применять `"POST"`, а не `"GET"`.

Важная информация об ограничениях безопасности, налагаемых на применение *loadVariables( )* в зависимости от домена, приведена в описании `System.security`.

Применение

Некоторые броузеры могут кэшировать результаты многократных обращений *loadVariable( )* по одному и тому же URL, в результате чего новые данные с сервера не загружаются. От этой неприятности можно избавиться, добавив в каждый вызов *loadVariables( )* фиктивную переменную, чтобы URL оказывался уникальным. Вот пример, в котором уникальный URL генерируется путем добавления времени в миллисекундах:

```
loadVariables("http://www.mysite.com/cgi-bin/myScript.pl?cacheKiller="
              + getTimer(), serverResponse);
```

Ошибки реализации

Метод POST не поддерживается в Internet Explorer 4.5 на Макинтошах. Это было исправлено в IE 5.

На Макинтошах метод *loadVariables( )* не работает с выполняемыми проекторами Flash 5 (то есть фильмами, соединенными с автономным проигрывателем Flash 5 в один исполняемый файл). Однако *loadVariables( )* работает с фильмами, открываемыми в автономном проигрывателе вручную. Чтобы решить эту проблему, отправляйте данные с помощью методов *XML.send( )* и *XML.sendAndLoad( )*.

См. также

loadVariablesNum( ), класс *LoadVars*, *MovieClip.loadVariables( )*, *System.security.allowDomain( )*; глава 17

loadVariablesNum(), глобальная функция

Flash 5

прикрепить группу внешних переменных к уровню документа

```
loadVariablesNum(URL, level)
loadVariablesNum(URL, level, method)
```

Аргументы

url Строка, задающая путь к источнику переменных: сценарию на стороне сервера, который возвращает переменные, или текстовому файлу, содержащему переменные.

level Неотрицательное целое или выражение с неотрицательным целым значением, задающее уровень документа, на котором будут определены загруженные переменные.

method Необязательная строка, указывающая метод отправки переменных внешнему сценарию. Если она задана, то переменные текущей временной диаграммы передаются сценарию, а `level` получает загруженные переменные. Если опущена, то переменные загружаются, но не отправляются. Допустимыми значениями для `method` являются `"GET"` и `"POST"`. Этот параметр должен быть литералом, а не переменной или другим выражением. Под Windows в автономных версиях Flash Player до версии 6 всегда применяется метод `"GET"` независимо от значения, задаваемого `method`. Во Flash 6 поддерживаются оба метода – `"GET"` и `"POST"`. См. приложение F.

Описание

Функция *loadVariablesNum( )* почти идентична *loadVariables( )*, но требует, чтобы целевой уровень `level` был задан как число, а не как строка. Это значит, что *loadVariablesNum( )* может прикреплять переменные только к уровням документов, а не к клипам. Целевой уровень можно задавать динамически, например:

```
var myLevel = 2;
loadVariablesNum("myVars.txt", myLevel);
```

Аналогичный результат можно получить с помощью конкатенации строк и обычной функции *loadVariables( )*:

```
loadVariables("myVars.txt", "_level" + myLevel);
```

См. также

loadVariables( )

LoadVars, класс

Flash 6

экспорт или импорт переменных из внешнего источника

Конструктор

```
new LoadVars()
```

Свойства

contentType

MIME-тип контента, используемый в операциях *send( )* и *sendAndLoad( )*.

loaded

Состояние операций *load( )* или *sendAndLoad( )*.

Методы

decode( )

Преобразовать в свойства строку переменных в кодировке URL.

getBytesLoaded( )

Узнать, какая часть загружаемых переменных (в байтах) уже получена.

getBytesTotal( )

Получить суммарный размер загружаемых переменных в байтах.

load( )

Импортировать переменные в кодировке URL из текстового файла или серверного приложения.

send( )

Передать переменные в кодировке URL приложению или сценарию на стороне сервера.

sendAndLoad( )

Передать переменные приложению на стороне сервера и получить переменные в обратном направлении.

toString( )

Возвращает строку переменных в кодировке URL, которая была бы отправлена *send( )* или *sendAndLoad( )*.

Обработчики событий

onData( )

Обработчик, выполняемый по окончании загрузки данных внешних переменных.

onLoad( )

Обработчик, выполняемый после преобразования внешних переменных в свойства объекта.

Описание

Класс *LoadVars* передает переменные внешнему источнику, которым может быть текстовый файл или сценарий на стороне сервера, или принимает их от него. В табл. С.9. приведены типичные сценарии применения *LoadVars* и методы, используемые в каждом случае.

Таблица С.9. Сценарии применения LoadVars

Требуемое действие	Метод
Получить внешние переменные из текстового файла или сценария на стороне сервера	*LoadVars.load( )*
Послать переменные ActionScript сценарию на стороне сервера и принять возможные результаты в веб-броузере	*LoadVars.send( )*
Послать переменные ActionScript сценарию на стороне сервера и принять возможные результаты во Flash	*LoadVars. sendAndLoad( )*

Методы *send( )*, *load( )* и *sendAndLoad( )* должны вызываться через экземпляр *LoadVars*, а не сам класс. Свойства экземпляра представляют собой переменные, которые должны быть отправлены или которые получены. Метод *send( )* преобразует свойства экземпляра *LoadVars* в строку переменных в кодировке URL и посылает их в виде запроса HTTP приложению, выполняющемуся на сервере. Метод *load( )* получает переменные в кодировке URL и преобразует их в свойства объекта *LoadVars*. Метод *sendAndLoad( )* выполняет обе эти задачи. Подробности и примеры см. в описании каждого метода.

Класс *LoadVars* обеспечивает решение тех же основных задач, что и глобальная функция *loadVariables( )*, впервые появившаяся во Flash 4, но имеет более логичный формат и несколько важных преимуществ:

- После получения переменных выполняется обработчик *onLoad( )*, что служит автоматическим оповещением и избавляет от необходимости опроса результатов, как в случае *loadVariables( )*.
- За процессом загрузки переменных можно наблюдать с помощью *getBytesLoaded( )* и *getBytesTotal( )*, что позволяет выявлять проблемы или показывать индикатор загрузки для дительных операций.
- Переменные хранятся как свойства объекта *LoadVars*, а не в физических клипах, что позволяет более эффективно разделить код и носитель. Это также позволяет посылать некоторое подмножество переменных временной диаграммы, а не все переменные, как в случае *loadVariables( )*.

Переменные, передаваемые в операциях *load( )* или *sendAndLoad( )*, должны быть отформатированы согласно следующим правилам кодировки URL :

- Имя каждой переменной должно отделяться от ее значения знаком равенства, без пробелов, как в `firstName=stephen`.
- Пары имен/значений должны разделяться амперсандами (&), как в `firstName=stephen&lastName=burke`.
- Пробелы должны быть заменены знаком «плюс» (+), а не `%20`.
- Любой символ, не являющийся пробелом, цифрой (1–9) или буквой Latin 1 без акцента (a–z, A–Z), должен быть заменен по следующим правилам:
  - Если `System.useCodepage` имеет значение `false` (по умолчанию), то символ заменяется одной или несколькими шестнадцатеричными escape-последовательностями вида `%xx`, представляющими код символа в UTF-8. Символы Unicode в диапазоне 0–127 кодируются в виде `%xx` (один байт), символы в диапазоне 128–2047 кодируются как `%xx%xx` (два байта), а символы от 2048 и выше кодируются как `%xx%xx%xx` (три байта).
  - Если `System.useCodepage` имеет значение `true` и язык системы не является азиатским двухбайтовым, то символ заменяется шестнадцатеричной escape-последовательностью `%xx`, где `xx` – шестнадцатеричная кодовая последовательность для этого символа. Кодовая позиция определяется согласно кодовой странице системы (на большинстве западных компьютеров это обычно Latin 1). Учтите, что на Макинтошах системная кодовая страница называется Mac Roman, но Flash использует вместо нее Latin 1.
  - Если `System.useCodepage` имеет значение `true` и язык системы является азиатским двухбайтовым, то символ заменяется шестнадцатеричной escape-последовательностью `%xx` или `%xx%xx` в соответствии с двухбайтовой кодировкой символов, поддерживаемой системой (например, Shift-JIS).
- Flash Player 5 действует так, как если бы `System.useCodepage` имела значение `true`.

В следующем коде приведено содержимое текстового файла, который надо импортировать во Flash. Импортируются переменные `name` и `address` со значениями `"stephen"` и `"65 nowhere st!"` соответственно (обратите внимание на кодировку восклицательного знака `%21`):

```
name=stephen&address=65+nowhere+st%21
```

Обычно приложения на сервере автоматически переводят переменные в кодировку URL для передачи их во Flash, но переменные в текстовых файлах нужно кодировать вручную. Например, во Flash Player 5 (или если `System.useCodePage` имеет значение `true` во Flash Player 6) символ й из верхней части таблицы ASCII имеет кодовую пози-

цию 233, или E9 в шестнадцатеричном виде, поэтому он записывается как `%E9`. Слово *resumé* превращается в `resum%E9`. С другой стороны, во Flash Player 6, когда `System.useCodePage` имеет значение `false` (по умолчанию), символ й надо вводить непосредственно, а файл переменных должен быть сохранен в формате Unicode UTF-8. (В Windows Notepad формат UTF-8 можно выбрать в диалоговом окне «Save as».) Если файл нельзя сохранить как UTF-8 или нельзя ввести символ непосредственно, необходимо использовать одно-, двух- или трехбайтовые escape-последовательности UTF-8 для этого символа. Например, последовательностью UTF-8 для й является `%C3%A9`, поэтому слово *resumé* записывается как `resum%C3%A9`. Получить escape-последовательность UTF-8 для символа во Flash Player 6 можно с помощью функции *escape( )*. Например:

```
trace(escape("e"));  // Выводит: %C3%A9
```

По общему правилу, если интернациональных символов много, то данные следует загружать из файлов XML в формате Unicode, а не с помощью *LoadVars*.

По соображениям безопасности *LoadVars* действует только с хостами того домена, из которого был загружен фильм. Подробности см. в `System.security`.

Класс *LoadVars* не всегда оказывается лучшим способом загрузки данных во Flash. Если переменные надо импортировать только один раз во время загрузки в броузер, можно воспользоваться атрибутом `FLASHVARS` тегов HTML `<OBJECT>` и `<EMBED>` (см. раздел «Загрузка внешних переменных» главы 2). Если объем данных велик, полезным может оказаться класс *XML*.

См. также

loadVariables( ), `System.security`, класс *XML*; «Загрузка внешних переменных» в главе 2; глава 17

LoadVars.contentType, свойство

Flash 6

тип MIME, используемый для операций send() и sendAndLoad() **чтение/запись**

loadVarsObject.contentType

Описание

Свойство `contentType` определяет MIME-тип содержимого, посылаемого серверу при вызове *LoadVars.send( )* или *LoadVars.sendAndLoad( )*. В большинстве практических случаев значение по умолчанию (`application/x-www-form-urlencoded`) изменять не требуется. Свойство доступно для записи, чтобы обеспечить гибкость типа MIME в особых обстоятельствах.

Свойство `LoadVars.prototype.contentType` позволяет изменить MIME-тип содержимого для всех экземпляров *LoadVars*, которые не задают его значение сами. Например:

```
LoadVars.prototype.contentType = "yourContentTypeGoesHere";
```

Пример

```
dataSender = new LoadVars();
trace(dataSender.contentType);  // Выводит: application/x-www-form-urlencoded
```

См. также

LoadVars.send( ), *LoadVars.sendAndLoad( )*

LoadVars.decode(), метод

Flash 6 (не документирован)

превращает строку переменных в кодировке URL в свойства

loadVarsObject.decode(*varString*)

Аргументы

varString

Строка переменных в кодировке URL. Сведения о кодировке URL см. в подразделе «Описание» статьи «LoadVars, класс».

Описание

Метод *decode( )* превращает содержащиеся в *varString* переменные в кодировке URL в свойства указанного объекта *loadVarsObject*. Он дает удобное средство для превращения переменных в кодировке URL в свойства объекта, требуемые для *LoadVars.onData( )*. Метод не документирован, но можно предположить, что его поддержка будет сохранена, поскольку он внутренне используется Flash для получения загруженных переменных.

Обратите внимание, что значения свойств, создаваемых *decode( )*, всегда представляю собой строки.

Пример

```
// Создать экземпляр LoadVars
varsObj = new LoadVars();

// Преобразовать строку в свойства объекта x и y
varsObj.decode("x=10&y=20");

// Проверить значение y
trace(varsObj.y);  // Выводит: 20
```

См. также *LoadVars.load( )*, *LoadVars.onData( )*

LoadVars.getBytesLoaded(), метод

Flash 6

узнать, какая часть (в байтах) загружаемых переменных уже получена

loadVarsObject.getBytesLoaded()

Возвращаемое значение

Количество байтов, переданных к данному времени, или undefined, если методы *load( )* или *sendAndLoad( )* не вызывались с *loadVarsObject*.

Описание

Метод *getBytesLoaded( )* сообщает, какая часть загружаемых переменных уже поступила в проигрыватель. Для нового объекта *LoadVars* метод *getBytesLoaded( )* возвращает undefined. Непосредственно после вызова *load( )* или *sendAndLoad( )* метод *getBytesLoaded( )* возвращает 0. По мере поступления данных *getBytesLoaded( )* возвращает количество байтов, переданных к данному моменту, пока не окажутся загруженными все переменные. С этого момента метод возвращает общее количество байтов, переданных в последний раз, пока не поступит следующее обращение к *load( )* или *sendAndLoad( )*. Чтобы превратить байты в килобайты, надо разделить полученное значение на 1024.

Как и *getBytesTotal( )*, *getBytesLoaded( )* чаще всего применяется, чтобы показать ход извлечения переменных, как иллюстрирует следующий пример.

Пример

Пример С.1 демонстрирует ход загрузки переменных из текстового файла *vars.txt*, в котором находятся две переменные в кодировке URL – `msg1` и `msg2`. В этом коде есть две интересные особенности:

- Чтобы избежать кэширования, применяется *sendAndLoad( )*, а не *load( )*. Подробности см. в описании *LoadVars. load( )*.
- Для отправки и приема данных используются два разных объекта. Подробности см. в описании *LoadVars.sendAndLoad( )*.

Пример С.1. Предварительный загрузчик LoadVars

```
// Создать текстовое поле для отображения хода загрузки
this.createTextField("loadmsg_txt", 1, 200, 200, 100, 20);

// Создать объект для отправки переменных
varSender = new LoadVars();

// Добавить фиктивное свойство, чтобы помешать броузеру
// кэшировать извлекаемые переменные
varSender.cacheKiller = new Date().getTime();

// Создать объект для получения переменных
varReceiver = new LoadVars();

// Назначить функцию обратного вызова, выполняемую по окончании загрузки переменных
varReceiver.onLoad = function () {
  // Показать загруженные переменные в окне Output
  trace("msg1 is " + this.msg1);
  trace("msg2 is " + this.msg2);

  // Загрузка завершена, прекратить вызов функции контроля хода загрузки
  clearInterval(varPreloader);

  // Вызвать checkVarStatus() для показа итогового количества загруженных байтов
  checkVarStatus();
}

// Запросить переменные с помощью sendAndLoad(), а не load(),
// чтобы броузер не воспользовался кэшем.
varSender.sendAndLoad("http://www.moock.org/temp/vars2.txt", varReceiver, "GET");

// Создать интервал для проверки состояния загрузки переменных каждые 100 мс
varPreloader = setInterval(checkVarStatus,100);

// Функция, сообщающая о состоянии загрузки переменных
function checkVarStatus () {
  // Получить количество загруженных байтов и преобразовать в килобайты
  var kbLoaded = Math.floor(varReceiver.getBytesLoaded()/1024);
  // Получить общее количество байтов и преобразовать в килобайты
  var kbTotal = Math.floor(varReceiver.getBytesTotal()/1024);
  // Если общее количество байтов нельзя узнать...
  if (kbTotal == undefined) {
    // ...вместо суммы показать вопросы
    kbTotal = "???";
  }
```

```
    // Отобразить текущее состояние загрузки в текстовом поле
    loadmsg_txt.text = kbLoaded + "/" + kbTotal;
  }
```

Ошибки реализации

В документации Macromedia по Flash MX утверждается, что *getBytesLoaded( )* возвращает `undefined`, если в настоящий момент операция загрузки не выполняется. На самом деле Flash Player 6 возвращает общее количество байтов, переданных в предыдущую загрузку, даже если она уже завершилась.

См. также

LoadVars.getBytesTotal( ), *LoadVars.load( )*, *LoadVars.sendAndLoad( )*, *setInterval( )*, класс *TextField*

LoadVars.getBytesTotal(), метод

Flash 6

получить предполагаемый суммарный размер загружаемых переменных

loadVarsObject.getBytesTotal()

Возвращаемое значение

Суммарное количество байтов, передаваемых в результате вызова *load( )* или *sendAndLoad( )*, либо `undefined`, пока количество байтов не определено.

Описание

Метод *getBytesTotal( )* сообщает о предположительном количестве байтов, участвующих в текущей передаче переменных. Для нового объекта *LoadVars* метод *getBytesTotal( )* возвращает `undefined`. Сразу после вызова *load( )* или *sendAndLoad( )* метод *getBytesTotal( )* продолжает возвращать `undefined`, пока не будет определен размер загружаемых переменных (обычно в течение секунды). Затем он возвращает суммарный объем в байтах загружаемых переменных, пока не последует следующий вызов *load( )* или *sendAndLoad( )*. Если суммарный объем в байтах невозможно определить (например, из-за отсутствия атрибута Content-Length в заголовке ответа HTTP), *getBytesTotal( )* продолжает возвращать `undefined`. Чтобы преобразовать байты в килобайты, необходимо разделить полученное значение на 1024.

Образец применения *getBytesTotal( )* для создания предзагрузчика переменных имеется в примере C.1 статьи «LoadVars.getBytesLoaded()».

Применение

Необходимо следить, чтобы деление на *getBytesTotal( )* выполнялось только после того, как метод вернет какое-нибудь положительное целое число.

См. также

LoadVars.getBytesLoaded( ), *LoadVars.load( )*, *LoadVars.sendAndLoad( )*

LoadVars.load(), метод

Flash 6

импорт переменных в кодировке URL из текстового файла или приложения на сервере

loadVarsObject.load(*url*)

Аргументы

url Строка, задающая путь к источнику переменных – сценарию на стороне сервера, который возвращает переменные, или текстовому файлу, содержащему переменные.

Возвращаемое значение

Булево значение, указывающее на успешность выполнения метода. Учтите, что возвращаемое значение менее полезно, чем может показаться. Метод *load( )* возвращает `false`, только когда вызывается без аргументов. Поскольку загрузка происходит асинхронно, возвращаемое *load( )* значение указывает только, успешно ли была начата загрузка. Значение `true` возвращается даже в том случае, когда операция загрузки в конечном итоге оказывается неудачной. В последнем случае вызывается *onLoad( )* со статусом `false`.

Описание

Метод *load( )* инициирует загрузку переменных в кодировке URL из *`url`*, который может быть локальным или удаленным текстовым файлом либо серверным приложением. За ходом загрузки можно следить с помощью методов *getBytesLoaded( )* и *getBytesTotal( )*. Загруженные переменные подвергаются синтаксическому анализу и преобразуются в свойства объекта `loadVarsObject`, после чего вызывается *`loadVarsObject`.onLoad( )*. Все свойства имеют тип строк. Для перехвата загруженных данных перед преобразованием их в свойства используется *LoadVars.onData( )*. Обратите внимание, что до выполнения *onLoad( )* переменные недоступны в виде свойств. Аналогично должны быть загружены все переменные, прежде чем они станут доступны ActionScript в обработчике *onData( )*.

Если *`url`* задает приложение или сценарий на сервере, тип MIME передаваемых данных должен быть указан как `"application/x-www-form-urlencoded"`. Вот обычная команда задания MIME в сценарии Perl:

```
print "Content-type: application/x-www-form-urlencoded\n\n";
```

Дополнительные сведения о кодировке URL во Flash см. в подразделе «Описание» статьи «LoadVars, класс».

По соображениям безопасности *LoadVars.load( )* работает только с хостами того домена, из которого был загружен фильм. Подробности см. в `System.security`.

Применение

На практике при вызове методов *load( )* и *sendAndLoad( )* загрузка переменных может не быть инициирована, если переменные на *`url`* кэшированы броузером, в котором выполняется фильм Flash. Чтобы заставить броузер всегда загружать переменные (и игнорировать его кэш), можно воспользоваться строкой запроса, содержащей фиктивную переменную с уникальным значением. Обычно в этом качестве выступает текущее время (в миллисекундах), поскольку предполагается достаточная его уникальность. Например:

```
theVars = new LoadVars();
theVars.load("http://www.site.com/vars.txt?cacheKiller=" + new Date().getTime());
```

Тот же результат можно получить посредством метода *sendAndLoad( )*, который породит запрос HTTP GET с фиктивной переменной (`cacheKiller`):

```
varSender = new LoadVars();
varReceiver = new LoadVars();
```

```
varSender.cacheKiller = new Date().getTime();
varSender.sendAndLoad("http://www.yourserver.com/vars.txt", varReceiver, "GET");
```

В любом случае необходимо проверять данные внутри обработчика *onLoad( )* перед тем, как их использовать, поскольку кэшированные данные могут быть поврежденными или неполными. См. пример в описании *LoadVars.onLoad( )*.

Пример

Следующий код использует объект *LoadVars* для получения переменной из текстового файла *vars.txt*, который выглядит так:

```
msg1=hello+world
```

Показаны два варианта *load( )*, в одном из которых есть уникальная переменная, предотвращающая кэширование:

```
// Создать текстовое поле для показа загружаемой переменной
this.createTextField("msg1_txt", 1, 200, 200, 100, 20);

// Создать экземпляр LoadVars
externalVars = new LoadVars();

// Назначить функцию обратного вызова onLoad(), которая сообщит о поступлении переменной
externalVars.onLoad = function () {
  // Когда поступит переменная, вывести ее в нашем текстовом поле
  msg1_txt.text = this.msg1;
}

// Теперь загрузим текстовый файл
externalVars.load("http://www.site.com/vars.txt");

// Или, чтобы обойти кэш броузера, добавим в строку запроса уникальную фиктивную переменую
externalVars.load("http://www.site.com/vars.txt?cacheKiller=" + new Date().getTime());
```

См. также

loadVariables( ), *LoadVars.getBytesLoaded( )*, *LoadVars.onData( )*, *LoadVars.sendAndLoad( )*

LoadVars.loaded, свойство

Flash 6

статус операции load() или sendAndLoad() **чтение/запись**

loadVarsObject.loaded

Описание

Свойство `loaded` позволяет быстро определить, закончена ли загрузка переменных. Свойство `loaded` нового объекта *LoadVars* имеет значение `undefined`. После первого обращения к *load( )* или *sendAndLoad( )* `loaded` получает значение `false`. Если загрузка не будет завершена из-за ошибок или плохого соединения, сохранится значение `false`. Если загрузка завершена, `loaded` устанавливается равным `true` в *loadVarsObject*. *onData( )* и остается истинным до следующего вызова *load( )* или *sendAndLoad( )*.

Как правило, лучше всего реагировать на завершение загрузки переменных с помощью *LoadVars.onLoad( )*. Однако и свойство `loaded` может оказаться удобным, когда некоторому участку кода требуется немедленный доступ к состоянию загрузки объекта *LoadVars*, как показывает следующий пример.

Пример

Допустим, что есть игра, в которой загружаются лучшие результаты и есть кнопка, нажатием которой можно отправить результаты по почте. Кнопка должна работать, только если загружены результаты, как в следующем коде:

```
// Предполагается, что highScoreVars - глобальная переменная, которая содержит объект LoadVars
emailButton.onRelease = function () {
  if (highScoreVars.loaded == true) {
    // Здесь код для email...
  } else {
    // Переменные не загружены. Предложить пользователю подождать...
  }
}
```

См. также *LoadVars.onLoad( )*

LoadVars.onData(), обработчик события

Flash 6 (не документирован)

обработчик, выполняемый по завершении загрузки внешних переменных, но перед их анализом

loadVarsObject.onData(*src*);

Аргументы

src Строка, содержащая необработанные переменные в кодировке URL.

Описание

Обработчик *onData( )* автоматически выполняется, когда завершается загрузка переменных, запрошенных методами *load( )* или *sendAndLoad( )*, но их синтаксический разбор еще не проведен.

Обработчику *onData( )* можно назначить пользовательскую функцию обратного вызова, чтобы перехватить необработанные исходные переменные до того, как ActionScript успеет преобразовать переменные в кодировке URL в свойства объекта *loadVarObject*, и перед вызовом обработчика *onLoad( )*. Такой непосредственный доступ к полученным с сервера данным может быть полезен при отладке. Однако в большинстве других ситуаций назначать пользовательскую функцию обратного вызова для *onData( )* не требуется.

По умолчанию (если не определять пользовательскую функцию обратного вызова) *onData( )* ведет себя следующим образом:

- Если полученные необработанные данные имеют значение `undefined`, вызывается *loadVarsObject*.*onLoad( )* с присваиванием параметру *success* значения `false`.
- В противном случае вызывается *loadVarsObject*.*decode( )* для преобразования переменных из *src* в свойства объекта *loadVarsObject*, затем свойству *loadVarsObject*.loaded присваивается значение `true` и вызывается *loadVarsObject*.*onLoad( )* с параметром *success*, равным `true`.

При реализации *onData( )* нужно обеспечить преобразование загруженных переменных в свойства с помощью *decode( )*, как это делается в следующем примере.

Пример

Здесь показано, как отобразить необработанные загруженные переменные и не дать ActionScript провести их анализ. Метод *onLoad( )* не вызывается, и свойство `loaded` не устанавливается:

```
loadedVars = new LoadVars();
loadedVars.onData = function (src) {
  trace("Variables received: " + src);
};
loadedVars.load("http://www.thesite.com/cgi-bin/getData.pl");
```

Далее мы выводим необработанные загруженные переменные, но затем вызываем *onLoad( )*, как это и должно обычно происходить. Обратите внимание, что мы стараемся воспроизвести все поведение *onData( )* по умолчанию, включая преобразование загруженных переменных в свойства с помощью *decode( )* и присваивание свойству `loaded` значения `true`:

```
loadedVars = new LoadVars();
loadedVars.onData = function (src) {
  trace("Variables received: " + src);
  if (src == undefined) {
    this.onLoad(false);
  } else {
    this.decode(src);
    this.loaded = true;
    this.onLoad(true);
  }
};

loadedVars.onLoad = function (success) {
  trace("Vars load successfully? " + success);
}

loadedVars.load("http://www.thesite.com/cgi-bin/getData.pl");
```

См. также *LoadVars.decode( )*, *LoadVars.onLoad( )*

LoadVars.onLoad(), обработчик события

Flash 6

обработчик, выполняемый после преобразования внешних переменных в свойства объекта

loadVarsObject.onLoad(*success*)

Аргументы

success Булево значение, указывающее, была ли загрузка успешной (`true`) или неудачной (`false`).

Описание

Обработчик *onLoad( )* автоматически выполняется, когда внешние переменные загружены и преобразованы в свойства *loadVarsObject*. Благодаря *onLoad( )* мы узнаем, что можно обрабатывать загруженные переменные; если сгенерировано событие *onLoad( )*, а следовательно, разрешен доступ к загруженному контенту. По умолчанию обработчик *onLoad( )* объекта *LoadVars* представляет собой пустую функцию. Для того чтобы воспользоваться *onLoad( )*, надо присвоить ему функцию обратного вызова. Например:

```
// Создать экземпляр LoadVars
varReceiver = new LoadVars();
```

```
// Присвоить onLoad() функцию обратного вызова
varReceiver.onLoad = handleLoad;

// Определить функцию обратного вызова
function handleLoad (success) {
  // Здесь выполняется обработка переменных...
}
```

Синтаксис, в котором применяется литерал функции, тоже стандартен:

```
varReceiver.onLoad = function (success) {
  // Здесь выполняется обработка переменных...
}
```

Внутри нашей функции обратного вызова мы обращаемся к свойствам *loadVarsObject* с помощью ключевого слова `this`. Например:

```
function handleLoad (success) {
  // Показать значение загруженной переменной score в окне Output
  trace(this.score);
}
```

Параметр *success* сообщает, были ли фактически загружены какие-либо переменные. Если данные, загруженные с сервера, ничего не содержат, *success* имеет значение `false`; в противном случае – `true`. Этот параметр позволяет организовать элементарную проверку факта загрузки каких-либо полезных данных:

```
function handleLoad (success) {
  if (success) {
    // Показать значение загруженной переменной score в окне Output
    trace(this.score);
  } else {
    trace("No variables were found");
  }
}
```

Однако значение `true` параметра *success* не гарантирует, что переменные правильно отформатированы и могут быть использованы; оно просто указывает, что содержимое ответа HTTP не пустое. Иногда данные переменных оказываются неправильно принятыми, даже если *success* имеет значение `true`. Например, сценарий на стороне сервера мог послать неверные данные, либо из кэша сервера были считаны устаревшие данные. Хорошим стилем считается явная проверка ожидаемых переменных, предшествующая их использованию. Так, в следующем коде проверка переменной `msg` предшествует обращению к ней:

```
function handleLoad (success) {
  if (success) {
    // Проверка msg
    if (this.msg != undefined) {
      // Можно продолжить...
    } else {
      // msg не была получена...прерваться или попытаться загрузить переменные еще раз...
    }
  } else {
    trace("Переменные не обнаружены");
  }
}
```

Обработчик *onLoad( )* избавляет от необходимости писать пользовательский код, который определяет получение данных после вызова *load( )* или *sendAndLoad( )* путем опроса. Законченный пример, демонстрирующий *onLoad( )* и технику предварительной загрузки переменных, см. в описании *LoadVars.getBytesLoaded( )* и *LoadVars.sendAndLoad( )*.

См. также Also

LoadVars.getBytesLoaded( ), *LoadVars.load( )*, *LoadVars.onData( )*, *LoadVars.sendAndLoad( )*

LoadVars.send(), метод

Flash 6

передача переменных в кодировке URL приложению или сценарию на сервере

```
loadVarsObject.send(url, target)
loadVarsObject.send(url, target, method)
```

Аргументы

url

Строка, задающая абсолютный или относительный адрес внешнего сценария или приложения, которые получают переменные.

target

Строка, задающая имя окна броузера или фрейма, в которые должны быть загружены результаты. Может быть пользовательским именем или одним из четырех предустановленных: `"_blank"`, `"_parent"`, `"_self"` или `"_top"`.

method

Необязательная строка, задающая метод HTTP, с помощью которого свойства *loadVarsObject* посылаются во внешний сценарий: `"GET"` или `"POST"`. Если он опущен, принимается значение по умолчанию `"POST"`. Автономная версия Flash Player в Windows всегда использует метод `"GET"` независимо от того, что указано в *method*. Поддержка `GET` и `POST` подробно рассмотрена в приложении F.

Возвращаемое значение

Булево значение, указывающее на успешность выполнения метода. Учтите, что возвращаемое значение менее полезно, чем может показаться. При передаче в качестве *url* любого значения (в том числе `undefined`!) *send( )* возвращает `true`. При вызове без аргументов *send( )* возвращает `false`.

Описание

Метод *send( )* преобразует все свойства объекта *loadVarsObject* в строку переменных в кодировке URL, которую потом посылает в запросе HTTP по указанному *url* методом, заданным параметром *method*. Если сценарий или приложение по указанному *url* генерируют результат, то он выводится во фрейм или окно броузера, заданные параметром *target*. Если фильм выполняется в автономном проигрывателе, запускается броузер, определенный по умолчанию в системе. Подробное описание параметров *target* и *method* приведено в статье, посвященной глобальной функции *getURL( )*. О кодировке URL рассказывается в «Описании» класса *LoadVars*. Строку в кодировке URL, отправляемую на *url*, можно локально просмотреть с помощью метода *LoadVars.toString( )*.

Обратите внимание, что функция *getURL( )* тоже может передавать переменные удаленному сценарию, но она не так удобна, как *LoadVars.send( )*, поскольку для ее работы необходим клип, в котором будут храниться переменные.

Ошибки реализации

В документации Macromedia утверждается, что если параметр *target* не задан, то сценарий *url* будет выполнен, но его результаты пропадут. Однако проверка Flash Player 6 показывает, что если *target* не задан, то метод *send( )* не выполняет сценарий на *url*.

Пример

В следующем коде объект *LoadVars* используется для того, чтобы послать свойства name и age в виде переменных в кодировке URL сценарию *userFind.pl*:

```
// Создать объект LoadVars
varSender = new LoadVars();

// Присвоить значения свойствам объекта varSender
// Это переменные, которые будут отправлены серверу
varSender.name = "Bruce";
varSender.age = "13";

// Отправить переменные на сервер и принять ответ в новое окно броузера
varSender.send("http://www.someserver.com/cgi-bin/userFind.pl", "_blank", "GET");
```

См. также

getURL( ), *LoadVars.onLoad( )*, *LoadVars.sendAndLoad( )*, *LoadVars.toString( )*

LoadVars.sendAndLoad(), метод

Flash 6

передача переменных приложению или сценарию на сервере и получение переменных в ответ

```
loadVarsObject.sendAndLoad(url, targetObject)
loadVarsObject.sendAndLoad(url, targetObject, method)
```

Аргументы

url

Строка, задающая абсолютный или относительный адрес внешнего сценария или приложения, которые получают переменные Flash.

targetObject

Ссылка на объект *LoadVars*, который получит ответ сервера.

method

Необязательная строка, задающая метод HTTP, с помощью которого свойства *loadVarsObject* посылаются во внешний сценарий: "GET" или "POST". Если он опущен, принимается значение по умолчанию "POST". В отличие от метода *send( )*, *sendAndLoad( )* может использовать "POST" во всех версиях автономного проигрывателя. Поддержка GET и POST описана в приложении F.

Возвращаемое значение

Булево значение, указывающее на успешность выполнения метода. Учтите, что поскольку операция *sendAndLoad( )* выполняется асинхронно, возвращаемое значение лишь указывает, успешным ли было начало операции – в конечном итоге она все равно

может оказаться неудачной, и тогда *onLoad( )* будет вызван с параметром *success*, равным false. Метод *sendAndLoad( )* возвращает false, если ему передано неправильное количество аргументов (их должно быть два или три), если параметр *targetObject* не является объектом или запрос запрещен из-за ограничений системы защиты.

Описание

Метод *sendAndLoad( )* преобразует все свойства объекта *loadVarsObject* (за исключением contentType и loaded) в строку переменных в кодировке URL, которую потом посылает в запросе HTTP по указанному *url* методом, заданным параметром *method*. Предполагается, что сценарий или приложение, расположенное по адресу *url*, возвратит строку переменных в кодировке URL, которые загружаются в проигрыватель и преобразуются в свойства объекта *targetObject*, как это описано для *LoadVars.load( )*. Когда свойства становятся доступными, выполняется обработчик события *onLoad( )* объекта *targetObject*. Все свойства имеют тип строки.

Хотя один объект *LoadVars* может использоваться как для отправки, так и получения переменных, лучше разделить эти две операции между двумя объектами *LoadVars*, как показано в следующем примере. Благодаря этому входящие и исходящие данные заключаются в свои собственные объекты и исключается ненужная передача свойств принимающего объекта, таких как обработчик *loadVarsObject.onLoad*.

По соображениям безопасности *LoadVars.sendAndLoad( )* работает только с хостами в том домене, из которого был загружен фильм. Дополнительные сведения см. в описании System.security.

Применение

На практике при вызове методов *sendAndLoad( )* и *load( )* загрузка переменных может не быть инициирована, если переменные на *url* кэшированы броузером, в котором выполняется фильм Flash. См. в статье *LoadVars.load( )* описание приема, который вынуждает броузер не использовать свой кэш.

Пример

В следующем коде объект *LoadVars* используется для отправки переменных сценарию *lookup.pl*. Объект varReceiver перехватывает ответ, присланный *lookup.pl*:

```
// Создать объекты LoadVars. Один для передачи...
varSender = new LoadVars();
// ...и другой для приема.
varReceiver = new LoadVars();

// Присвоить значения свойствам отправителя, которые будут переменными, посылаемыми серверу.
varSender.name = "Bruce";
varSender.age = "13";

// Присвоить функцию обратного вызова свойству onLoad объекта varReceiver.
// Этот обработчик будет обрабатывать полученные переменные.
varReceiver.onLoad = function () {
  // К загруженным переменным можно обращаться как к свойствам this, например, this.age.
  // Для перечисления свойств объекта varReceiver используется цикл  for-in.
  // Обратите внимание, что свойства loaded и contentType не входят в перечисление.
  for (p in this) {
    // Мы не хотим выводить свойство onLoad, поэтому проверяем,
    // является ли текущее свойство строкой. Если да...
    if (typeof this[p] == "string") {
      // ...показать его значение в окне Output. Все загруженные
      // переменные получаются в виде строк.
```

```
        trace("The variable " + p + " has the value " + this[p]);
      }
    }
  }

  // Перед отправкой переменных покажем их в окне Output. (Полезно для отладки)
  trace("Sending " + varSender.toString());

  // Теперь отправим переменные серверу и будем ждать ответ
  varSender.sendAndLoad("http://www.yourserver.com/cgi-bin/lookup.pl", varReceiver, "GET");
```

См. также

LoadVars.load( ), *LoadVars.onLoad( )*, *LoadVars.send( )*, *LoadVars.toString( )*

LoadVars.toString(), метод

Flash 6

получить ту же строку в кодировке URL, которую отправили бы send() или sendAndLoad()

loadVarsObject.toString()

Возвращаемое значение

Строка переменных в кодировке URL.

Описание

Метод *toString( )* возвращает перечислимые свойства объекта *loadVarsObject* в виде строки переменных в кодировке URL (перечислимые свойства – это те, которые видны в цикле *for-in*). Он возвращает те же данные, которые были бы посланы сценарию сервера при вызове метода *send( )* или *sendAndLoad( )*, что удобно для отладки. Свойства добавляются в возвращаемую строку в порядке, обратном времени их создания (последнее созданное свойство будет первым в строке, предпоследнее – вторым и т. д.). Значение каждого свойства преобразуется в строку в формате URL.

Пример

```
loadedVars = new LoadVars();
loadedVars.day = 22;
loadedVars.month = "Feb.";
loadedVars.year = 2002;

// Выводит: year=2002&month=Feb%2E&day=22
// Обратите внимание на преобразование '.' после "Feb" в шестнадцатеричное %2E.
// О кодировке URL см. "Описание" класса LoadVars
trace(loadedVars.toString());
```

См. также

LoadVars.onLoad( ), *LoadVars.send( )*, *LoadVars.sendAndLoad( )*

LocalConnection, класс

Flash 6

непосредственная передача данных между двумя фильмами, выполняющимися на одной машине

Конструктор

new LocalConnection()

Методы

close( ) Прекратить прием сообщений соединением.

connect( ) Открыть именованное соединение.

domain( ) Возвращает строку с поддоменом фильма.

send( ) Вызвать метод удаленного объекта *LocalConnection*.

Обработчики событий

allowDomain( )

Функция обратного вызова, разрешающая или запрещающая попытку соединения между разными доменами.

onStatus( )

Функция обратного вызова, сообщающая начальное состояние вызова *send( )*.

Описание

Класс *LocalConnection* позволяет фильмам, выполняющимся в различных проигрывателях Flash обмениваться данными между собой. Проигрыватели должны выполняться на одном и том же компьютере, но они могут быть встроены в любое приложение. Например, фильмы, выполняющиеся в подключаемом модуле Netscape Flash 6, Internet Explorer ActiveX Flash Player и Standalone Flash Player могут общаться друг с другом, если это происходит на одной машине. Обмен информацией между двумя проигрывателями, выполняющимися на разных машинах, следует организовывать при помощи *XML*, *LoadVars*, *XMLSocket* или Macromedia Flash Communication Server MX (Comm Server).

Во всех операциях с участием *LocalConnection* заняты хотя бы два объекта *LocalConnection*, один из которых выступает в качестве получателя (фильма, получающего сообщения), а другой – отправителя (фильма, посылающего сообщения). Оба фильма могут действовать одновременно в качестве получателя и отправителя, но для начала проиллюстрируем простую однонаправленную связь.

Первая строка кода получателя такая:

```
receiveConn = new LocalConnection();
```

Первая строка кода отправителя следующая:

```
sendConn = new LocalConnection();
```

Получатель и отправитель, создав у себя свои объекты *LocalConnection*, должны договориться о том, какое имя будет дано соединению. На приемном конце имя соединения указывается при открытии последнего. Однако соединение следует открывать не раньше, чем приемник определит методы, которые отправитель будет вызывать удаленно. Эти методы составляют основу связи между фильмами. Например, определим в объекте `receiveConn` метод, отображающий переданный ему аргумент в текстовом поле:

```
// Создать текстовое поле
this.createTextField("received_txt", 1, 50, 100, 400, 100);
received_txt.border = true;

// Определить метод
receiveConn.displayMsg = function (msg) {
  received_txt.text = msg;
}
```

Определив метод *receiveConn.displayMsg( )*, можно благополучно открыть соединение приемника. Ожидание входящих сообщений на приемном конце начинается пос-

ле вызова метода *connect( )*. В качестве аргумента методу *connect( )* передается имя соединения. В следующей строке это "channel1":

```
receiveConn.connect("channel1");
```

Теперь наш приемник готов к работе. Осталось послать запрос отправителя, чтобы вызвать метод *displayMsg( )* в принимающем фильме. Для вызова удаленных методов отправитель применяет метод *send( )*. Аргументы *send( )* задают имя соединения («channel1»), метод, который надо вызвать («displayMsg»), и аргумент(ы) метода («hello world»), например:

```
sendConn.send("channel1", "displayMsg", "hello world");
```

Вот полный код, необходимый для установления элементарного соединения с помощью *LocalConnection*:

```
// КОД В ФИЛЬМЕ-ПОЛУЧАТЕЛЕ
receiveConn = new LocalConnection();
receiveConn.methodName = function (param1, param2) {
  // предложения
}
receiveConn.connect("connectionName");

// КОД В ФИЛЬМЕ-ОТПРАВИТЕЛЕ
sendConn = new LocalConnection();
sendConn.send("connectionName", "methodName", "value1", "value2");
```

По соображениям безопасности отправитель и получатель, помещенные в разные домены, по умолчанию не могут связываться с помощью *LocalConnection*. Как разрешить связаться друг с другом фильмам, находящимся в разных доменах, подробно рассказано в описании *LocalConnection.allowDomain( )*. Все фильмы, выполняющиеся в локальной файловой системе, считаются находящимися в одном домене («localhost»), поэтому ограничения безопасности на них не распространяются.

Связь через *LocalConnection* с точки зрения получателя осуществляется по принципу «многие к одному». Любое количество отправителей могут посылать сообщения в соединение с некоторым именем, но только один получатель может принимать их на этом соединении. Если другой получатель попытается открыть соединение с именем, которое уже существует, это ему не удастся, и метод *connect( )* возвратит `false`.

Объект *LocalConnection* может выступать в роли как отправителя, так и получателя, благодаря чему два объекта могут осуществлять двунаправленную связь. Образец соответствующего кода см. в примере статьи *LocalConnection.onStatus( )*.

До версии Flash 6 локальная связь между фильмами могла осуществляться только с помощью *fscommand( )* и JavaScript и только для определенных комбинаций броузеров и платформ. Подробности см. в описании *fscommand( )*.

См. также *fscommand( )*

LocalConnection.allowDomain(), обработчик события — Flash 6

функция обратного вызова, разрешающая или запрещающая попытку соединения между разными доменами

receiveConn.allowDomain(*domain*)

Аргументы

domain Строка, представляющая поддомен отправителя.

Возвращаемое значение

`true`, если соединение разрешено, в противном случае `false`.

Описание

Событие *allowDomain( )* генерируется для принимающего объекта *LocalConnection* при получении сообщения. Функция обратного вызова *allowDomain( )* должна разрешить или запретить попытку соединения, возвращая значение `true` (разрешено) или `false` (отказано). Если отправитель и получатель загружены из одного и того же домена, включая локальную машину, Flash автоматически разрешает все соединения, и применять *allowDomain( )* не надо. Но если отправитель и получатель загружены из разных доменов, то соединение разрешается, только если *allowDomain( )* реализована и возвращает `true`. В реализации *allowDomain( )* обычно сравнивается значение параметра *domain* со списком доверенных хостов. Обратите внимание, что *LocalConnection.allowDomain( )* представляет собой обработчик обратного вызова, который возвращает значение, определяющее, допускается ли соединение. В этом отличие от метода *Security.settings.allowDomain( )*, которому надо один раз передать для настройки список разрешенных поддоменов.

Следующий код фильма-приемника создает объект *LocalConnection*, который открывает соединение с именем "channel1":

```
// Создать текстовое поле для вывода сообщений
this.createTextField("output_txt", 1, 50, 100, 400, 100);
output_txt.border = true;

// Создать объект соединения
receiveConn = new LocalConnection();
// Присвоить метод, вызываемый удаленно
receiveConn.displayMsg = function (msg) {
  output_txt.text += msg + "\n";
}

// Попытка открыть соединение с именем "channel1". Записать результат в переменную.
connectSuccess = receiveConn.connect("channel1");
// Показать, было ли соединение успешным
output_txt.text = "Receiver connection succeeded? " + connectSuccess + "\n";
```

Теперь предположим, что мы размещаем этот фильм на *http://www.graydient.com* и хотим разрешить ему получать сообщения только с *moock.org* (исключив даже *graydient.com*!). Присвоим *allowDomain( )* такую функцию обратного вызова:

```
// Обработка входящих соединений
receiveConn.allowDomain = function (domain) {
  // Если выполняется этот обработчик, значит, получено новое сообщение.
  // Показать, откуда прибыло сообщение.
  output_txt.text += "New message from: " + domain + "\n";

  // Теперь главное решение: если сообщение с moock.org...
  if (domain == "moock.org") {
    // ...разрешить
    return true;
  }  else {
    // ...иначе отклонить
    return false;
  }
}
```

Для того чтобы разрешить сообщения от *moock.org* и текущего поддомена принимающего фильма (в данном случае это *graydient.com*), можно обратиться к методу *domain( )*, который возвращает строку с поддоменом фильма:

```
// Обработка входящих сообщений
receiveConn.allowDomain = function (domain) {
  // Показать, откуда прибыло сообщение
  output_txt.text += "New message from: " + domain + "\n";

  // Если сообщение с moock.org или домена получателя...
  if (domain == "moock.org" || domain == this.domain()) {
    // ...разрешить
    return true;
  }  else {
    // ...иначе отказать
    return false;
  }
}
```

Либо в более кратком виде:

```
receiveConn.allowDomain = function (domain) {
  return domain == "moock.org" || domain == this.domain();
}
```

Наш получатель делает свое дело, но осталась одна маленькая неувязочка: фильм-отправитель должен правильно указывать имя соединения при отправке сообщений. Когда receiveConn открывал соединение, он задал его имя как "channel1", однако во избежание конфликтов имен соединений между разными доменами Flash автоматически добавляет в начало имени соединения имя поддомена фильма и двоеточие. Например, если наш фильм-получатель размещен на *graydient.com*, фактическим именем соединения будет не "channel1", а "graydient.com:channel1". Поэтому отправитель должен указать имя соединения так:

```
sendConn = new LocalConnection();
// Добавить домен получателя к имени соединения
sendConn.send("graydient.com:channel1", "displayMsg", "hello world");
```

Если мы знаем, что наш получатель будет принимать сообщения только при размещении на *graydient.com*, все хорошо. Однако если мы хотим получать сообщения независимо от размещения получателя, положение несколько осложняется. Допустим, например, что мы хотим, чтобы получатель работал как в локальной системе (с именем домена "localhost"), так и на *graydient.com*. Если мы вызовем метод *connect( )* из фильма, находящегося на *graydient.com*:

```
receiveConn.connect("channel1");
```

то итоговым именем соединения станет "graydient.com:channel1". Но при вызове метода *connect( )* из фильма, расположенного в локальной файловой системе, конечным именем соединения станет "localhost:channel1". Очевидно, отправитель не может задавать оба имени соединения одновременно. Решение заключается в том, чтобы дать Flash команду не помещать имя домена в начало имени соединения, для чего надо добавить ведущий символ подчеркивания (_), например:

```
receiveConn.connect("_channel1");
```

При отсутствии префикса в виде имени домена отправитель может указывать соединение непосредственно:

```
sendConn.send("_channel1", "displayMsg", "hello world");
```

Учтите, однако, что имена в разных доменах, начинающиеся с символа подчеркивания, могут вступать в конфликт между собой. Например, если получатель на *oreilly.com* открывает соединение с именем "_bookComm", а затем *moock.org* попытается открыть соединение с таким же именем, то соединение на *moock.org* не будет открыто. Поэтому следует задавать уникальные имена, если задан ведущий символ подчеркивания (например, "_org.moock. channel1") *oreilly.comoreilly.com*.

См. также

LocalConnection.connect( ), *LocalConnection.domain(_)*, *LocalConnection.send( )*, *Security. settings.allowDomain( )*

LocalConnection.close(), метод

Flash 6

прекратить прием сообщений соединением

receiveConn.close()

Возвращаемое значение

Булево значение `true`, если открытое соединение закрылось, и `false`, если открытое соединение *receiveConn* не было найдено.

Описание

Метод *close( )* применяется только принимающим объектом *LocalConnection*, чтобы закрыть соединение, после чего отправители не смогут вызывать его методы. Например, следующий код открывает соединение с именем "channel2", позволяя в результате удаленно вызывать *displayMsg( )*:

```
receiveConn = new LocalConnection();
receiveConn.displayMsg = function (msg) {
  output_txt.text += msg + "\n";
}
receiveConn.connect("channel2");
```

Позднее можно закрыть соединение "channel2", вызвав *close( )*:

```
receiveConn.close();
```

Закрытие соединения не позволит никаким отправителям вызывать *displayMsg( )*. Объект *LocalConnection* может одновременно поддерживать только одно открытое соединение, поэтому *close( )* не требует передачи имени соединения в качестве параметра. Обратите внимание, что *close( )* применяется только к получателям: объекту *LocalConnection*, который только отправляет сообщения, не требуется ни открывать, ни закрывать соединение.

См. также

LocalConnection.connect( )

LocalConnection.connect(), метод

Flash 6

открыть именованное соединение

receiveConn.connect(*connectionName*)

Аргументы

connectionName

Строка с именем соединения.

Возвращаемое значение

Булево значение `true`, если соединение успешно открыто, и `false`, если операция завершилась неудачей (например, если соединение с именем *connectionName* уже есть в каком-нибудь проигрывателе или у *receiveConn* уже есть открытое соединение).

Описание

Метод *connect( )* применяется принимающим объектом *LocalConnection* для открытия соединения, что позволит отправителям вызывать в дальнейшем его методы. Все методы должны быть определены до вызова *connect( )*. Например, следующий код создает объект *LocalConnection*, затем определяет его методы, затем открывает соединение с именем "channel3" (делая метод *displayMsg( )* доступным для удаленного выполнения):

```
receiveConn = new LocalConnection();
receiveConn.displayMsg = function (msg) {
  output_txt.text += msg + "\n";
}
receiveConn.connect("channel3");
```

Путем проверки значения, возвращаемого *connect( )*, можно корректно реагировать на неудачные попытки открытия соединения, например:

```
connectSuccess = receiveConn.connect("channel3");
if (connectSuccess) {
  // ...продолжить выполнение приложения
} else {
  // ...вывести экран с сообщением об ошибке
}
```

Если *connectionName* не начинается с символа подчеркивания (_), к указанному имени автоматически добавляется префикс в виде поддомена *receiveConn* и двоеточия (например, "channel3" превращается в "oreilly.com:channel3", если фильм размещен на *www.oreilly.com*). Если *connectionName* начинается с символа подчеркивания, префикс не добавляется.

Отправители при вызове удаленных методов *receiveConn* должны точно указывать окончательное имя соединения (автоматический префикс + *connectionName*) в методе *send( )*. Соответствие имен соединений рассмотрено в описании *LocalConnection.allowDomain( )* и *LocalConnection.send( )*.

Объект *LocalConnection* может одновременно поддерживать только одно открытое соединение и не может открывать соединение с именем, которое уже используется. Для закрытия открытого соединения предназначен метод *LocalConnection.close( )*.

См. также

LocalConnection.allowDomain( ), *LocalConnection.close( )*, *LocalConnection.send( )*

LocalConnection.domain(), метод

Flash 6

возвращает строку с именем поддомена фильма

localConn.domain()

Возвращаемое значение

Строка с именем поддомена фильма.

Описание

Метод *domain( )* возвращает так называемый *поддомен* текущего фильма в виде строки. Для имен доменов, состоящих из одной или двух компонентов, таких как *apollo* и *oreilly.com*, поддомен совпадает с полным именем домена. Для имен доменов, состоящих из трех и более компонентов, таких как *www.moock.org* или *events.news.thedailybugle.com*, поддомен определяется как все, что находится после первой точки, в данном случае "moock.org" и "news.thedailybugle.com". Для фильмов, выполняющихся в локальной файловой системе, *domain( )* возвращает "localhost". Поддомен используется в ряде проверок системы защиты Flash Player, включая передачу данных в сети классами *LoadVars* и *XML*.

Метод *domain( )* применяется объектами *LocalConnection*, осуществляющими обмен данными между разными доменами. Он позволяет указывать поддомен отправителя или получателя, имя которого заранее не известно. Например, следующий код разрешает принимать сообщения отправителя, находящегося на *oreilly.com* или в том же домене, где находится получатель (каким бы ни был этот домен):

```
receiveConn.allowDomain = function (domain) {
  return domain == "oreilly.com" || domain == this.domain();
}
```

См. также

LocalConnection.allowDomain( ), *LocalConnection.send( )*, `MovieClip._url`, *System.security.allowDomain( )*

LocalConnection.onStatus(), обработчик события

Flash 6

функция обратного вызова, сообщающая начальное состояние вызова send()

`sendConn.onStatus(`*`infoObject`*`)`

Аргументы

infoObject Объект, содержащий состояние последней операции *send( )*.

Описание

Событие *onStatus( )* генерируется, когда `sendConn` вызывает *LocalConnection.send( )*. Оно сообщает, смогла ли операция *send( )* найти действующее соединение с заданным именем *connectionName*. Функция обратного вызова *onStatus( )* получает один аргумент, *infoObject*, в свойстве `level` которого находится значение «error», если указанное соединение *connectionName* не было найдено, или «status», если оно было обнаружено. Этот обработчик используется при отладке и в коде, обрабатывающем ошибки. Например:

```
sendConn.onStatus = function (infoObject) {
  if (infoObject.level == "status") {
    trace("Connection found.");
  } else if (infoObject.level == "error") {
    trace("No connection found.");
    // ...обработка неудачной попытки соединения...
  }
}
```

Даже если *infoObject*`.level` содержит «status» (соединение обнаружено), получатель может отвергнуть соединение или не содержать заданного метода. О разрешении свя-

зи между различными доменами рассказано в описании *LocalConnection.allowDomain( )*. Для проверки успешности вызова метода получатель должен вызвать ответный метод отправителя, как показано в примере.

Пример

В следующем примере демонстрируется объект-отправитель, вызывающий метод *displayMsg( )* объекта-получателя. Объект-получатель отвечает тем, что вызывает метод *methodReturn( )* отправителя. Если соединение не обнаружено, отправитель выводит отладочное сообщение. Обратите внимание, что каждый объект *LocalConnection* действует одновременно и как отправитель, и как получатель, но для удобства мы будем называть «отправителем» первичного отправителя и «получателем» первичного получателя. В этом примере удаленный вызов осуществляется через соединение «channelA» (приемник его открывает, а отправитель к нему обращается), а ответ передается через соединение «channelB» (отправитель его открывает, а приемник к нему обращается). При таком механизме отправитель и получатель должны заранее договориться об именах каналов. Кроме того, предполагается, что фильмы отправителя и получателя находятся в одном поддомене. Организация связи между отправителями и получателями из разных доменов рассмотрена в описании *LocalConnection.send( )*.

Приведем код отправителя:

```
// Создать текстовое поле для вывода сообщений
this.createTextField("sendOutput_txt", 1, 50, 100, 400, 100);
sendOutput_txt.border = true;

// Создать объект LocalConnection отправителя
sendConn = new LocalConnection();
// Создать метод для перехвата сообщений, полученных от приемника
sendConn.methodReturn = function (result) {
  sendOutput_txt.text += result + "\n";
}

// Попытка открыть соединение "channelB" для получения ответа приемника
connectSuccess = sendConn.connect("channelB");

// Вывести сообщение об успешном открытии соединения
sendOutput_txt.text = "Sender connection succeeded: " + connectSuccess + "\n";

// Вызвать удаленный displayMsg() через "channelA" в расчете, что приемник его открыл
sendConn.send("channelA", "displayMsg", "hello world");

// Проверить доступность соединения
sendConn.onStatus = function (infoObject) {
  if (infoObject.level == "status") {
    sendOutput_txt.text += "Connection found.\n";
  } else if (infoObject.level == "error") {
    sendOutput_txt.text += "No connection found.\n";
  }
}
```

А вот код получателя:

```
// Создать текстовое поле для вывода сообщений
this.createTextField("receiveOutput_txt", 1, 50, 100, 400, 100);
receiveOutput_txt.border = true;

// Создать объект LocalConnection
receiveConn = new LocalConnection();
```

```
// Присвоить метод, который будет вызываться удаленно
receiveConn.displayMsg = function (msg) {
  receiveOutput_txt.text += msg + "\n";
  // Оповестить отправителя о том, что метод вызван. Ответ посылается через "channelB"
  this.send("channelB", "methodReturn", "displayMsg() called");
}

// Попытка открыть соединение "channelA" для приема сообщений
connectSuccess = receiveConn.connect("channelA");

// Вывести сообщение об успешном открытии соединения
receiveOutput_txt.text = "Receiver connection succeeded: " + connectSuccess + "\n";
```

См. также

LocalConnection.allowDomain( ), *LocalConnection.connect( )*, *LocalConnection.domain( )*, *LocalConnection.send( )*

LocalConnection.send(), метод

Flash 6

вызывает метод удаленного объекта LocalConnection

localConn.send(*connectionName*, *methodName*, *arg1*, *arg2*, ...*argn*)

Параметры

connectionName

Строка с именем соединения, через которое должен быть вызван *methodName*, заданным в методе *connect( )* получателя.

methodName

Строка с именем метода, который должен быть вызван у получателя. Не должна совпадать с зарезервированными словами «send», «connect», «close», «domain», «onStatus» или «allowDomain».

arg1, ...*argn*

Аргументы, которые может требовать *methodName*.

Возвращаемое значение

Булево значение `true`, если структура *send( )* синтаксически корректна, и `false` в противном случае.

Описание

Метод *send( )* посылает запрос на вызов метода *methodName* по соединению с именем *connectionName*. Метод *send( )* может передать не более 40 Кбайт данных. Передаваемые дополнительные аргументы могут иметь любой тип (булев, объектный, числовой и т. д.). Класс посылаемого объекта сохраняется только в том случае, если он зарегистрирован в *LocalConnection* на обоих концах с помощью *Object.registerClass( )*. Подробности см. в описании *Object.registerClass( )*.

Если фильмы отправителя и получателя загружены из одного и того же домена, *connectionName* для *send( )* должно совпадать с *connectionName*, заданным в методе *connect( )* получателя. Если фильмы отправителя и получателя загружены из разных доменов, то генерируемое Flash окончательное имя для отправителя должно совпадать с окончательным именем, генерируемым для получателя.

У отправителя окончательное имя соединения генерируется так:

- Если *connectionName*, переданное в *send( )*, не начинается с символа подчеркивания (_) и не содержит двоеточие (:), то к *connectionName* добавляется префикс из имени поддомена отправителя и двоеточие. Например, *connectionName* со значением «channel1» получает окончательное значение "*senderdomain*.com:channel1" (где *senderdomain*.com представляет собой поддомен фильма-отправителя).
- Если *connectionName*, переданное в *send( )*, содержит двоеточие, то принимается буквальное значение заданного имени. Например, *connectionName* со значением «moock.org:channel1» остается без изменений.
- Если *connectionName*, переданное в *send( )*, начинается с символа подчеркивания, то принимается буквальное значение заданного имени. Например, *connectionName* со значением «_channel1» остается без изменений.

У получателя окончательное имя соединения генерируется так:

- Если *connectionName*, переданное в *connect( )*, не начинается с символа подчеркивания (_) и не содержит двоеточия (:), то к *connectionName* добавляется префикс из имени поддомена получателя и двоеточие. Например, connectionName со значением «channel1» получает окончательное значение "*receiverdomain*.com: channel1" (где *receiverdomain*.com представляет собой поддомен фильма-получателя).
- Если *connectionName*, переданное в *connect( )*, начинается с символа подчеркивания, то принимается буквальное значение заданного имени. Например, *connectionName* со значением «_channel1» остается без изменений.

Получатель не может задавать *connectionName* в формате "moock.org:channel1" (получателю не разрешается задавать собственный поддомен). В табл. C.10. показаны значения заданного и окончательного имени соединения как для отправителя, так и для получателя и отмечено, будут ли получаемые комбинации совпадать при выполнении фильмов в одном домене и в разных доменах. Полное обсуждение вопросов совпадения имен и отмены ограничений системы защиты, действующих при связи фильмов из разных доменов, см. в статье *LocalConnection.allowDomain( )*.

Таблица C.10. Совпадение имен соединений

Указанный отправитель	Окончательный отправитель	Указанный получатель	Окончательный получатель	Совпадение для одного домена	Совпадение для разных доменов
name	senddomain: name	name	recdomain: name	Да	Нет (автоматические префиксы поддоменов не совпадут)
name	senddomain: name	_name	_name	Нет (получатель не использует поддомен)	Нет
_name	_name	name	recdomain: name	Нет (отправитель не использует поддомен)	Нет
_name	_name	_name	_name	Да	Да (но *allowDomain( )* должен разрешить соединение)

Таблица С.10 (продолжение)

Указанный отправитель	Окончательный отправитель	Указанный получатель	Окончательный получатель	Совпадение для одного домена	Совпадение для разных доменов
subdomain: name	subdomain: name	name	subdomain: name	Да, если указанный отправителем поддомен совпадает с фактическим поддоменом получателя	Да (но *allowDomain( )* должен разрешить соединение и указанный отправителем поддомен должен совпасть с фактическим поддоменом получателя)
subdomain: name	subdomain: name	_name	_name	Нет (получатель не использует поддомен)	Нет

См. также

LocalConnection.allowDomain( ), *LocalConnection.connect( )*, *LocalConnection.domain( )*

Math, объект

Flash 5; может использоваться при экспорте фильмов Flash 4

доступ к математическим функциям и константам

```
Math.propertyName
Math.methodName()
```

Свойства

E Константа *e*, основание натурального логарифма, приблизительно 2,71828.

LN10 Натуральный логарифм 10 (loge10), приблизительно 2,30259.

LN2 Натуральный логарифм 2 (loge2), приблизительно 0,69315.

LOG10E Логарифм *e* по основанию 10, приблизительно 0,43429.

LOG2E Логарифм *e* по основанию 2, приблизительно 1,44270.

PI Отношение длины окружности к ее диаметру, приблизительно 3,14159.

SQRT1_2 Величина, обратная квадратному корню из 2, приблизительно 0,70711.

SQRT2 Квадратный корень из 2, приблизительно 1,41421.

Методы

abs( ) Вычислить абсолютную величину числа.

acos( ) Вычислить арккосинус числа.

asin( ) Вычислить арксинус числа.

atan( ) Вычислить арктангенс числа.

atan2( ) Вычислить угол точки по отношению к оси x.

ceil( ) Округлить число вверх до ближайшего целого.

cos( ) Вычислить косинус угла.

exp( ) Возвести *e* в заданную степень.

floor( ) Возвратить ближайшее целое, меньшее или равное аргументу.

log( ) Вычислить натуральный логарифм числа.

max( ) Определить наибольшее из двух чисел.

min( ) Определить наименьшее из двух чисел.

pow( ) Возвести число в заданную степень.

random( ) Получить случайное число с плавающей точкой между 0 и 1.

round( ) Вычислить ближайшее целое.

sin( ) Вычислить синус угла.

sqrt( ) Вычислить квадратный корень из числа.

tan( ) Вычислить тангенс угла.

Описание

Объект *Math* предоставляет доступ к встроенным математическим функциям (через методы) и значениям констант (через свойства). Эти функции и константы применяются для того, чтобы с относительной легкостью выполнять сложные вычисления.

Обратите внимание, что свойства и методы объекта *Math* могут применяться в фильмах, экспортируемых в формате Flash 4, при этом Flash будет выполнять расчеты приближенно. Полученные приближенные значения могут быть приемлемыми, но могут не совпасть со значениями собственных функций Flash 5 и последующих версий. Точность значений, получаемых во Flash 4, удовлетворительна для таких «аккуратных» приложений, как вывод графики, но недостаточна для важных финансовых или инженерных расчетов.

Заметьте, что тригонометрические функции требуют измерения углов в радианах, а свойство Flash *MovieClip*.`_rotation` измеряется в градусах. Окружность содержит 2π радиан (1 радиан составляет примерно 57,3 градуса). Для преобразования из радиан в градусы применяется формула:

```
degrees = (radians / Math.PI) * 180;
```

А градусы в радианы преобразуются по формуле:

```
radians = (degrees / 180) * Math.PI;
```

См. также «Числовой тип данных» в главе 4

Math.abs(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет абсолютную величину числа

`Math.abs(`*x*`)`

Аргументы

x Любое число, целое или с плавающей точкой.

Возвращаемое значение

Абсолютная величина *x* (положительное число такой же величины, как *x*).

Описание

Метод *abs( )* вычисляет расстояние между *x* и 0 (называемое также *абсолютной величиной x*). Он оставляет ноль и положительные числа неизменными, но превращает отрицательные числа в положительные с той же величиной. Полезен для определения разности между двумя числами без учета того, которое из них больше. Например, он полезен для вычисления расстояния между двумя точками, потому что расстояния всегда положительны.

Пример

```
Math.abs(-5);  // Возвращает 5

// Вычисляет разность между двумя числами
function diff (num1, num2) {
  return Math.abs(num1-num2);
}

diff(-5, 5);  // Возвращает 10
```

Math.acos(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет арккосинус числа

`Math.acos(x)`

Аргументы

x Число от -1,0 до 1,0 (косинус угла).

Возвращаемое значение

Угол (в радианах), косинус которого равен *x*. Если *x* не находится в диапазоне от -1,0 до 1,0, возвращается NaN.

Описание

Функция арккосинус (иногда записываемая как $\cos^{-1}$) является обратной для функции косинуса. Она возвращает угол, косинус которого равен указанному значению, в радианах. Возвращаемое значение находится в диапазоне от 0 до π (то есть от 0 до 3,14159).

Пример

```
trace (Math.acos(1.0));  // Выводит: 0
trace (Math.acos(0.0));  // Выводит: 1.5707... (то есть pi/2)
```

См. также

Math.asin( ), *Math.atan()*, *Math.cos( )*

Math.asin(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет арксинус числа

`Math.asin(x)`

Аргументы

x Число от -1,0 до 1,0 (синус угла).

Возвращаемое значение

Угол (в радианах), синус которого равен *x*. Если *x* не находится в диапазоне от –1,0 до 1,0, возвращается `NaN`.

Описание

Функция арксинус (иногда записываемая как $\sin^{-1}$) является обратной для функции синуса. Она возвращает угол, синус которого равен указанному значению, в радианах. Возвращаемое значение находится в диапазоне от $-\pi/2$ до $\pi/2$ радиан.

Пример

```
trace (Math.asin(1.0));  // Выводит: 1.5707...  (то есть pi/2)
trace (Math.asin(0.0));  // Выводит: 0
```

См. также *Math.acos*, *Math.atan*, *Math.sin*

Math.atan(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет арктангенс числа

`Math.atan(x)`

Аргументы

x Число от `-Infinity` до `Infinity` включительно (тангенс некоторого угла).

Возвращаемое значение

Угол (в радианах), тангенс которого равен *x*.

Описание

Функция арктангенс (иногда записываемая как $\tan^{-1}$) является обратной для функции тангенса. Она возвращает угол, тангенс которого равен указанному значению, в радианах. Возвращаемое значение находится в диапазоне от $-\pi/2$ до $\pi/2$.

Пример

```
trace (Math.atan(1.0));       // Выводит: 0.78539...
trace (Math.atan(0.0));       // Выводит: 0
trace (Math.atan(-Infinity)); // Выводит: -1.5707... (то есть -pi/2)
```

См. также *Math.acos( )*, *Math.asin( )*, *Math.tan( )*

Math.atan2(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

определяет угол по точке

`Math.atan2(y, x)`

Аргументы

y Координата точки по оси y.

x Координата точки по оси x.

Возвращаемое значение

Угол (в радианах) между положительным направлением оси X и линией, соединяющей центр окружности и точку (x, y), измеряемый против часовой стрелки. Изменяется в диапазоне от π до $-\pi$. (Отрицательные значения указывают на углы ниже оси X.)

Описание

Метод *atan2( )*, как и *atan( )*, вычисляет арктангенс, но принимает в качестве аргументов координаты по осям x и y, а не их отношение. Поэтому вычисление с помощью *atan2( )* арктангенса:

```
Math.atan2(9, 3);  // Возвращает 1.24904577239825
```

эквивалентно вычислению арктангенса с помощью *atan( )* для отношения 9/3 (или 3):

```
Math.atan(3);      // То же самое
```

Применение

Обратите внимание, что координата *y* передается *atan2( )* в качестве первого аргумента, а координата *x* – как второй аргумент. Так и должно быть, поскольку тангенс равен отношению стороны, противоположной углу (y), к стороне, примыкающей к углу (x).

Пример

Метод *atan2( )* позволяет поворачивать клип в сторону движущейся цели. В коде следующего примера, который можно загрузить из сетевого хранилища кода, пользовательский метод *rotateTowardsMouse( )* добавляется ко всем клипам. Затем *rotateTowardsMouse( )* присваивается обработчику события *onEnterFrame( )* клипа `triangle_mc`, в результате чего в каждом новом кадре клип поворачивается в сторону курсора мыши. Такой же прием можно применить для ориентации неприятельского космического корабля в направлении космического корабля играющего:

```
// Преобразовать радианы в градусы. 2*pi радиан соответствует 360 градусам.
_global.radiansToDegrees = function (radians) {
  return (radians/Math.PI) * 180;
}

// Метод: MovieClip.rotateTowardsMouse()
//   Описание: Поворачивает положительную ось x в сторону указателя мыши.
MovieClip.prototype.rotateTowardsMouse = function () {
  // Создать объект точки, в котором хранятся x- и y-координаты этого клипа
  // относительно точки регистрации его родителя
  var point = {x:this._x, y:this._y};
  // Преобразовать локальные координаты родительского объекта клипа
  // в глобальные (сцены)
  this._parent.localToGlobal(point);
  // Измерить расстояние от точки регистрации этого клипа до мыши
  var deltaX = _root._xmouse - point.x;
  var deltaY = _root._ymouse - point.y;
  // Вычислить угол линии от точки регистрации этого клипа
  // до мыши. Обратите внимание, что сначала передается координата y.
  var rotationRadian = Math.atan2(deltaY, deltaX);
  // Преобразовать радианную меру угла в градусы.
  var rotationAngle = radiansToDegrees(rotationRadian); // См. функцию выше
  // Обновить угол поворота этого клипа, чтобы он был обращен к мыши.
  this._rotation = rotationAngle;
}
// Вызывать rotateTowardsMouse() в каждом кадре.
triangle_mc.onEnterFrame = rotateTowardsMouse;
```

Math.ceil(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

округляет число до следующего целого

`Math.ceil(x)`

Аргументы

x Число.

Возвращаемое значение

Следующее целое число, большее или равное *x*.

Описание

Метод *ceil( )* (то есть «потолок») преобразует число с плавающей точкой в первое целое число, большее или равное *x*. Целые числа не изменяются.

Пример

```
Math.ceil(1.00001);  // Возвращает 2
Math.ceil(5.5);      // Возвращает 6
Math.ceil(-5.5);     // Возвращает -5
Math.ceil(3);        // Возвращает 3
```

См. также

Math.floor( ), *Math.round( )*

Math.cos(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет косинус угла

`Math.cos(theta)`

Аргументы

theta

Угол в радианах (не в градусах), обычно в диапазоне от 0 до 2π, но может быть любым, поскольку косинус является периодической функцией.

Возвращаемое значение

Косинус *theta* (результат находится в диапазоне от –1,0 до 1,0).

Описание

Метод *cos( )* возвращает тригонометрический косинус угла. В прямоугольном треугольнике косинус угла получается в результате деления длины стороны, примыкающей к углу, на гипотенузу треугольника.

Применение

Обратите внимание, что *cos( )* предполагает задание угла в радианах, а не градусах.

Пример

```
trace (cos(0));            // Выводит: 1.0
trace (cos(Math.PI));      // Выводит: -1.0
```

Функцию *cos( )* можно применять вместе с функцией *sin( )* для вычисления точки на окружности, что используется в следующем примере для перемещения клипа по окружности. Если есть радиус окружности *r* и угол θ, измеренный против часовой стрелки от положительной горизонтальной оси, то точка имеет координаты (r×cosθ, r×sinθ):

```
// Глобальная функция: degreesToRadians()
// Преобразует градусы в радианы. В 360 градусах 2*pi радиан.
_global.degreesToRadians = function (degrees) {
  return (degrees/180) * Math.PI;
}

// Метод: MovieClip.moveInCircle()
//   Описание: Начинает перемещать клип по круговой траектории.
// Параметры: radius       Радиус кругового пути.
//            x            Горизонтальное положение центра круга.
//            y            Вертикальное положение центра круга.
//            degPerFrame  Угловое перемещение в каждом выводимом кадре.
MovieClip.prototype.moveInCircle = function (radius, x, y, degPerFrame) {
  var deg = 0;          // Угловое положение объекта в градусах, измеренное
                        // против часовой стрелки от горизонтали (оси X).

  // Присвоить функцию перемещения, которая будет выполняться один раз в каждом кадре.
  this.onEnterFrame = function () {
    // Увеличить угол поворота на 5 градусов
    deg += degPerFrame;

    // Поместить объект в нужное место. Flash инвертирует ось Y декартовых координат,
    // поэтому для получения новой точки мы уменьшаем координату y.
    var rotAngRad= degreesToRadians(deg);
    this._x = x + Math.cos(rotAngRad) * radius;
    this._y = y - Math.sin(rotAngRad) * radius;
  }
}

// Применение:
ball_mc.moveInCircle(100, 200, 200, 5);
```

См. также *Math.acos( )*, *Math.PI*, *Math.sin( )*, *Math.tan( )*

Math.E, свойство

Flash 5; может применяться при экспорте фильмов Flash 4

константа e (основание натуральных логарифмов)

`Math.E`

Описание

Свойство `E` хранит приближенное значение основания натуральных логарифмов (примерно 2,71828), обозначаемое в математике символом *e*. Это трансцендентное, так же как и π, число применяется в математических уравнениях, описывающих рост или изменение. Не путайте его с буквой *E*, используемой в экспоненциальных обозначениях, описываемых в разделе «Литералы с плавающей точкой» главы 4. Они не связаны между собой.

См. также *Math.log( )*, *Math.LN10( )*, *Math.LN2( )*

Math.exp(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

возводит константу *e* в заданную степень

`Math.exp(x)`

Аргументы

x Показатель степени, в которую следует возвести `Math.E`.

Возвращаемое значение

`Math.E` в степени *x*.

См. также `Math.E`, `Math.LN10`, `Math.LN2`, *Math.log( )*, `Math.LOG10E`, *Math.pow( )*

Math.floor(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

округляет число вниз до предыдущего целого

`Math.floor(x)`

Аргументы

x Число.

Возвращаемое значение

Ближайшее целое число, меньшее или равное *x*.

Описание

Метод *floor( )* преобразует число с плавающей точкой в первое целое число, меньшее или равное x. Целые числа остаются неизменными.

Пример

```
Math.floor(1.99999);  // Возвращает 1
Math.floor(5.5);      // Возвращает 5
Math.floor(-5.5);     // Возвращает -6
Math.floor(3);        // Возвращает 3

function minutesToHHMM (minutes) {
  var hours = Math.floor(minutes/60);
  minutes -= hours * 60;
  minutes = minutes < 10 ? "0" + minutes : minutes;
  return hours + ":" + minutes;
}
```

См. также *Math.ceil( )*, *Math.round( )*

Math.LN10, свойство

Flash 5; может применяться при экспорте фильмов Flash 4

натуральный логарифм 10 (loge10), приблизительно 2,30259

`Math.LN10`

Описание

Свойство `LN10` представляет натуральный логарифм 10 (логарифм 10 по основанию *e*), константу, приблизительно равную 2,30259.

См. также *Math.exp( )*, `Math.LN2`, `Math.LOG10E`

Math.LN2, свойство

Flash 5; может применяться при экспорте фильмов Flash 4

натуральный логарифм 2 (loge2), приблизительно 0,69315

`Math.LN2`

Описание

Свойство `LN2` представляет натуральный логарифм 2 (логарифм 2 по основанию *e*), константу, приблизительно равную 0,69315.

См. также *Math.exp( )*, `Math.LN10`, `Math.LOG10E`

Math.log(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет натуральный логарифм числа

`Math.log(x)`

Аргументы

x Целое положительное число.

Возвращаемое значение

Натуральный логарифм *x*.

Описание

Метод *log( )* вычисляет натуральный логарифм числа (то есть логарифм по основанию *e*). См. следующий пример вычисления логарифма по основанию 10.

Пример

```
trace (Math.log(Math.E));   // Выводит: 1

// Вычислить логарифм числа по основанию 10
function log10 (x) {
  return (Math.log(x) / Math.log(10));
}
```

Math.LOG10E, свойство

Flash 5; может применяться при экспорте фильмов Flash 4

логарифм e по основанию 10, приблизительно 0,43429

`Math.LOG10E`

Описание

Свойство `LOG10E` представляет обычный логарифм *e* (логарифм *e* по основанию 10), константу, приблизительно равную 0,43429.

Math.LOG2E, свойство

Flash 5; может применяться при экспорте фильмов Flash 4

логарифм e по основанию 2, приблизительно 1,44270

`Math.LOG2E`

Описание

Свойство `LOG2E` представляет логарифм *e* по основанию 2 (log2*e*), константу, приблизительно равную 1,44270.

Ошибки реализации

Во Flash 5.0.30.0 `LOG2E` ошибочно возвращает значение `Math.LN2` (0.69315). Ошибка исправлена во Flash 6.

Math.max(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

определяет большее из двух чисел

`Math.max(x, y)`

Аргументы

x Число.

y Число.

Возвращаемое значение

Большее из *x* и *y*.

Описание

Функция *max( )* сравнивает два числа и возвращает большее. При сравнении двух отрицательных чисел больше то число, у которого меньше абсолютная величина (см. пример ниже). Если аргументы – строки, то *max( )* вовзращает `NaN`.

Пример

```
Math.max(5, 1);                        // Возвращает 5
Math.max(-6, -5);                      // Возвращает -5
Math.max(Math.abs(-6), Math.abs(-5));  // Возвращает 6
```

В этом примере значение ограничивается заданным диапазоном :

```
function constrainToRange (checkVal, minVal, maxVal) {
  return Math.min(Math.max(checkVal, minVal), maxVal);
}
// Ограничить ползунок областью от 100 до 200
mySlider._x = constainToRange (mySlider._x, 100, 200);
```

В этом примере возвращается максимальное значение, содержащееся в массиве:

```
function maxInArray (checkArray) {
  maxVal = -Number.MAX_VALUE;// Присвоить maxVal очень маленькое число
  for (var i = 0; i < checkArray.length; i++) {
    maxVal = Math.max(checkArray[i], maxVal);
  }
  return maxVal;
}

trace(maxInArray([2,3,66,4,342,-90,0]));  // Выводит: 342
```

См. также

Math.abs( ), *Math.min( )*

Math.min(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

определяет меньшее из двух чисел

`Math.min(x, y)`

Аргументы

x Число.

y Число.

Возвращаемое значение

Меньшее из *x* и *y*.

Описание

Функция *min( )* сравнивает два числа и возвращает большее. При сравнении двух отрицательных чисел меньше то число, у которого больше абсолютная величина (см. пример ниже). Если аргументы – строки, то *min( )* возвращает NaN.

Пример

```
Math.min(5, 1);                        // Возвращает 1
Math.min(-6, -5);                      // Возвращает -6
Math.min(Math.abs(-6), Math.abs(-5));  // Возвращает 5
```

Упражнения для читателя: изменить пример из *Math.max( )*, чтобы возвращалось минимальное, а не максимальное значение в массиве.

См. также

Math.abs( ), *Math.max( )*

Math.PI, свойство

Flash 5; может применяться при экспорте фильмов Flash 4

отношение длины окружности к ее диаметру, приблизительно 3,14159

Math.PI

Описание

Свойство представляет константу π, отношение длины окружности к ее диаметру.

Пример

Math.PI чаще всего применяется для вычисления площади круга:

```
function circleArea (radius) {
  // PI умножить на радиус в квадрате можно также записать
  // как Math.PI * Math.pow(radius, 2)
  return Math.PI * (radius * radius);
}
```

См. также

Math.cos( ), *Math.sin( )*

Math.pow(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

возводит число в указанную степень

Math.pow(*base*, *exponent*)

Аргументы

base

Число, представляющее основание степенного выражения.

exponent

Число, представляющее степень (то есть показатель степени), в которую следует возвести *base*. Если основание отрицательно, показатель степени должен быть целым числом, иначе возвращается NaN.

Возвращаемое значение

base в степени *exponent*.

Описание

Метод *pow( )* можно применять для возведения любого числа в любую степень. Если аргумент *exponent* отрицателен, то *pow( )* возвращает $1 / (base^{abs(exponent)})$. Если же *exponent* – дробное число, то *pow( )* можно использовать для вычисления квадратного или кубического корня (в этом случае возвращается положительный корень, хотя, в математическом смысле, может иметься и отрицательный корень).

Пример

```
Math.pow(5, 2);    // Возвращает 25 (5 в квадрате)
Math.pow(5, 3);    // Возвращает 125 (5 в кубе)
Math.pow(5, -2);   // Возвращает 0.04 (1 делить на 25)
Math.pow(8, 1/3);  // Возвращает 2 (кубический корень из 8)
Math.pow(9, 1/2);  // Возвращает 3 (квадратный корень из 9)
Math.pow(10, 6);   // Возвращает 1000000 (можно записать и как 1e6)
```

Ошибки реализации

Во Flash Player 5.0.30.0 неправильно производился расчет *Math.pow( )* для отрицательных значений *base*. Например, *Math.pow(-2, 2)* давало результат NaN, хотя должно быть 4. Ошибка исправлена во Flash 6.

См. также

Math.exp( ), *Math.sqrt( )*, Math.SQRT2, Math.SQRT1_2

Math.random(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

генерировать случайное число от 0 до 1,0

```
Math.random()
```

Возвращаемое значение

Число с плавающей точкой, большее или равное 0,0 и меньшее 1,0.

Описание

Метод *random( )* предоставляет способ создания случайных чисел, которые можно использовать для случайного выбора действий в сценарии. Метод *random( )* генерирует случайное число от 0 до 0,99999... включительно, которое при необходимсоти можно масштабировать. Вот пример получения случайного числа от 0 до 5:

```
Math.floor(Math.random() * 5)
```

А так можно получить случайное целое число от 1 до 6:

```
Math.floor(Math.random() * 6) + 1
```

Следующая пользовательская функция возвращает случайное целое число в заданном диапазоне, а не число с плавающей точкой от 0 до 1:

```
// Возвращает число в диапазоне от minVal до maxVal включительно
function myRandom (minVal, maxVal) {
  return minVal + Math.floor(Math.random() * (maxVal + 1 - minVal));
}

// Вызвать функцию
dieRoll = myRandom(1, 6);  // Эмуляция шестигранной кости
trace(dieRoll);

// Моделировать две кости можно так:
twoDice = myRandom(2, 12);  // Минимальное значение 2, а не 1
```

```
// Чтобы возвратить выпавшие значения раздельно, используется массив
function rollTwoDice () {
  return [myRandom(1, 6), myRandom(1, 6)];
}
```

Из-за ошибки во Flash Player 5.0.30.0 такой подход чреват одной неточностью, проявляющейся чрезвычайно редко, но иногда имеющей значение. Во Flash Player 5.0.30.0 *random( )* генерирует значения в диапазоне от 0,0 до 1,0 включительно. Если умножить значение, возвращаемое *random( )*, на целое число *n*, получается значение в диапазоне от 0,0 до *n*. В нашем примере *Math.random( )* умножалось на 6, поэтому возвращалось значение от 0,0 до 6,0. Применение к полученному значению *floor( )* создает целые числа в диапазоне от 0 до *n* (0 ... 6 в нашем примере). Масштабирование и округление ведет к неправильному распределению случайных чисел: вероятность получить *n* значительно меньше, чем вероятность получить любое другое число из этого ряда.

Ошибка *Math.random( )* была исправлена во Flash Player 5.0.41.0. Если требуется поддержка старых версий проигрывателя, можно воспользоваться следующей версией функции *myRandom( )*, в которой проблема решается путем отбрасывания значения 1,0, если его возвратит *Math.random( )*:

```
// Возвращает целое число в диапазоне minVal .. maxVal включительно
function myRandom (minVal, maxVal) {
  do {
    r = Math.random();     // Продолжаем выбор, пока не получим число, отличное от 1.
  } while (r == 1);
  return minVal + Math.floor(r * (maxVal + 1 - minVal));
}

// Вызывать функцию
dieRoll = myRandom(1, 6);  // Безопасное моделирование шестигранной кости
                           // во Flash Player 5.0.30.0
```

Применение

Math.random( ) заменяет устаревшую функцию Flash 4 *random*.

Пример

Math.random( ) часто применяется для того, чтобы заставить головку воспроизведения скачком переместиться в случайный кадр на временной диаграмме. В следующем коде вызывается функция *myRandom( )* из предыдущего примера, после чего головка воспроизведения направляется в случайно выбранный кадр:

```
// Вызвать функцию; выбрать случайное число между 10 и 20
var destinationFrame = myRandom(10, 20);

// Направить головку воспроизведения в этот кадр
gotoAndStop(destinationFrame);
```

Math.round(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет ближайшее к данному числу целое

`Math.round(x)`

Аргументы

x Число.

Возвращаемое значение

Целое число, математически наиболее близкое к *x* (или сам *x*, если *x* – целое). Если дробная часть *x* равна в точности 0,5 (*x* находится на равном расстоянии от двух ближайших целых), *round( )* возвращает первое целое, большее *x*.

Описание

Метод *round( )* осуществляет обычное округление: преобразует число с плавающей точкой в ближайшее целое. Положительные числа с дробной частью, *меньшей* 0,5, и отрицательные числа с дробной частью, большей 0,5, округляются вниз. Положительные числа с дробной частью, *большей* или равной 0,5, и отрицательные числа с дробной частью, *меньшей* или равной 0,5, округляются вверх.

Пример

```
Math.round(1.4);      // Возвращает 1
Math.round(1.5);      // Возвращает 2
Math.round(1.6);      // Возвращает 2
Math.round(-5.4);     // Возвращает -5
Math.round(-5.5);     // Возвращает -5
Math.round(-5.6);     // Возвращает -6
```

См. также *int( )*, *Math.ceil( )*, *Math.floor( )*

Math.sin(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет синус угла

`Math.sin(`*theta*`)`

Аргументы

theta

Угол в радианах (не в градусах), обычно в диапазоне от 0 до 2π, но может быть любым, поскольку синус является периодической функцией.

Возвращаемое значение

Синус *theta* (результат находится в диапазоне от -1,0 до 1,0).

Описание

Метод *sin( )* возвращает тригонометрический синус угла. В прямоугольном треугольнике синус угла получается в результате деления длины стороны, противоположной углу, на гипотенузу треугольника.

Применение

Обратите внимание, что *sin( )* предполагает задание угла в радианах, а не градусах.

Пример

```
trace (Math.sin(0));            // Выводит: 0
trace (Math.sin(Math.PI/2));    // Выводит: 1.0
```

Функцию *sin( )* можно применять совместно с функцией *cos( )* для вычисления точки на окружности. См. пример в описании *Math.cos( )*.

См. также *Math.asin( )*, *Math.cos( )*, `Math.PI`, *Math.tan( )*

Math.sqrt(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет квадратный корень

`Math.sqrt(x)`

Аргументы

x Неотрицательное целое.

Возвращаемое значение

Корень квадратный из *x* или `NaN`, если *x* меньше 0.

Описание

Метод *sqrt( )* возвращает положительный корень из своего операнда (хотя имеется математически верный отрицательный корень). Он эквивалентен

```
Math.pow(x, 0.5)
```

Пример

```
Math.sqrt(4);       // Возвращает 2, хотя -2 тоже допустимый корень
Math.sqrt(36);      // Возвращает 6, хотя -6 тоже допустимый корень
Math.sqrt(-20);     // Возвращает NaN

// Вычислить расстояние между двумя точками как
// квадратный корень (deltaX квадрат плюс deltaY квадрат):
var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));
```

См. также

Math.pow( ), `Math.SQRT2`, `Math.SQRT1_2`

Math.SQRT1_2, свойство

Flash 5; может применяться при экспорте фильмов Flash 4

величина, обратная корню квадратному из 2, приблизительно 0,70711

`Math.SQRT1_2`

Описание

Свойство `SQRT1_2` является константой, аппроксимирующей значение `1/Math.SQRT2` (величина, обратная корню квадратному из 2), равной приблизительно 0,70711.

См. также

`Math.SQRT2`

Math.SQRT2, свойство

Flash 5; может применяться при экспорте фильмов Flash 4

корень квадратный из 2, приблизительно 1,41421

`Math.SQRT2`

Описание

Свойство `SQRT2` представляет собой константу, аппроксимирующую корень квадратный из 2, иррациональное число, равное приблизительно 1,41421.

См. также

`Math.SQRT1_2`

Math.tan(), метод

Flash 5; может применяться при экспорте фильмов Flash 4

вычисляет тангенс угла

`Math.tan(theta)`

Аргументы

theta Угол в радианах (не в градусах), в диапазоне от $-\pi/2$ до $\pi/2$, хотя может быть любым, поскольку тангенс является периодической функцией.

Возвращаемое значение

Тангенс *theta* (результат находится в диапазоне от `-Infinity` до `Infinity`).

Описание

Метод *tan( )* возвращает тригонометрический тангенс угла. В прямоугольном треугольнике тангенс угла получается в результате деления длины стороны, противоположной углу, на длину стороны, примыкающей к углу. Это то же самое, что отношение *Math.sin(theta)/Math.cos(theta)*, поэтому когда *cos(theta)* приближается к нулю, *tan(theta)* приближается к `Infinity`. Поэтому *tan(theta)* не может быть вычислен для значений аргумента $-\pi/2$, $\pi/2$, $-3\pi/2$, $3\pi/2$ и т. д.

Пример

```
trace (Math.tan(0));            // Выводит: 0
trace (Math.tan(Math.PI/4));    // Выводит: 1
```

См. также

Math.atan( ), *Math.cos( )*, *Math.sin( )*

maxscroll, свойство

Flash 4; заменено TextField.maxscroll во Flash 6

последняя допустимая верхняя строка текстового поля

theTextField.maxscroll

Возвращаемое значение

Положительное целое, представляющее номер последней допустимой верхней строки текстового поля.

Описание

Свойство `maxscroll` сообщает о самом большом допустимом значении `scroll` для текстового поля. Оно представляет номер последней строки в текстовом поле, которую можно использовать в качестве верхней строчки в области просмотра. Во Flash 4 и Flash 5 свойство `maxscroll` используется вместе со свойством `scroll` для управления полем с прокруткой текста. Во Flash 6 `maxscroll` оформлено в виде свойства класса *TextField*, *TextField*.maxscroll.

См. также

scroll; *TextField*.bottomScroll, *TextField*.hscroll, *TextField*.maxscroll, *TextField*.scroll

Mouse, объект

Flash 5; во Flash 6 добавлены события приемников

скрывает или показывает указатель мыши

Mouse.*methodName*()

Методы

addListener( )

Регистрирует объект для получения событий *onMouseDown( )*, *onMouseMove( )* и *onMouseUp( )*

hide( )

Скрывает курсор мыши.

removeListener( )

Прекращает отправку событий указанному приемнику.

show( )

Показывает курсор мыши.

Описание

Объект *Mouse* служит для получения информации о событиях мыши и управления видимостью курсора. События объекта *Mouse* позволяют создавать управляемые мышью интерфейсы, реагирующие на щелчки кнопок и перемещение указателя. Однако в приложениях, основанных на меню и кнопках, более простой доступ к действиям мыши предоставляют связанные с мышью события объектов *Button* (*onPress( )*, *onRelease( )*, *onRollOver( )* и т. д.). Обработчики событий мыши, такие как *onMouseDown( )*, удобны для перехвата щелчков мыши, выполняемых вне кнопок. Например, с их помощью можно перехватывать щелчки в произвольной точке экрана, чтобы переходить к следующему кадру слайд-шоу.

Метод *Mouse.hide( )* применяется для того, чтобы скрыть системный курсор мыши, обычно, чтобы заменить его пользовательским курсором, как будет показано в примере для *Mouse.hide( )*. Желательно бывает также спрятать курсор мыши в полноэкранном режиме, в фильмах, управляемых с клавиатуры и сенсорных киосках (touchscreen kiosks).

Обратите внимание, что объект *Mouse* не сообщает о местонахождении курсора мыши. Для определения местоположения курсора мыши относительно главной сцены следует применять `_root._xmouse` и `_root._ymouse`. Для определения местоположения курсора мыши относительно клипа предназначены свойства клипа `_xmouse` и `_ymouse`.

Применение

Во Flash 6 в объект *Mouse* были добавлены события приемников *onMouseDown( )*, *onMouseMove( )* и *onMouseUp( )*. Обычно они оказываются предпочтительными по сравнению с аналогичными событиями клипов *mouseDown*, *mouseMove* и *mouseUp* в стиле Flash 5, применяемыми внутри блока *onClipEvent* для обнаружения действий с мышью. Однако во Flash 6 функция *updateAfterEvent( )*, обновляющая экран в промежутке между кадрами, может использоваться только с событиями *MovieClip* для мыши и клавиатуры, либо с помощью *setInterval( )*. Поэтому когда требуется обновлять экран в промежутке межу кадрами, следует применять обработчики *MovieClip.onMouseDown( )*, *MovieClip.onMouseMove( )* и *MovieClip.onMouseUp( )*. Подробности см. в соответствующих статьях данного справочника. Возможно, в будущие версии проигрывателя будет включена поддержка *updateAfterEvent( )* в событиях приемника для *Mouse*.

См. также

Button.onPress( ), *Button.onRelease( )*, *MovieClip.onMouseDown( )*, *MovieClip. onMouseMove( )*, *MovieClip.onMouseUp( )*, `MovieClip._xmouse`, `MovieClip._ymouse`, *updateAfterEvent( )*

Mouse.addListener(), метод

Flash 6

регистрирует объект для получения событий onMouseDown(), onMouseMove() и onMouseUp()

`Mouse.addListener(listener)`

Аргументы

listener Объект, в котором определены методы *onMouseDown( )*, *onMouseMove( )* и/или *onMouseUp( )*.

Описание

Метод *addListener( )* помещает `listener` в список объектов, оповещаемых о событиях *onMouseDown( )*, *onMouseMove( )* и *onMouseUp( )* объекта *Mouse*. Параметр `listener` – это объект любого класса, в котором определены методы с именами *onMouseDown*, *onMouseMove* и *onMouseUp*. Эти методы вызываются, когда происходит соответствующее событие для *Mouse*. Экземпляры встроенных классов, например *TextField* или *Button*, умеют реагировать на события *Mouse*. Прекратить получение событий объектом приемника позволяет метод *Mouse.removeListener( )*.

На каждое событие *Mouse* могут независимо реагировать несколько объектов. Если в *Mouse* зарегистрировано несколько объектов, то их методы вызываются в том порядке, в котором добавлялись приемники.

Общие сведения о работе приемников и функций обратного вызова приведены в главе 10. Пример кода приемника *Mouse* см. в описании *Mouse.onMouseDown( )*.

См. также

Mouse.onMouseDown( ), *Mouse.onMouseMove( )*, *Mouse.onMouseUp( )*, *Mouse. removeListener( )*; глава 10

Mouse.hide(), метод

Flash 5

скрывает курсор мыши

```
Mouse.hide()
```

Возвращаемое значение

Возвращаемое значение указывает, был ли курсор мыши виден (1) или скрыт (0) перед вызовом *hide( )*.

Описание

Метод *hide( )* вызывает исчезновение нормального курсора мыши (обычно стрелки, иногда ладони) при нахождении над любой частью проигрывателя. Нормальный системный курсор появляется вновь, когда мышь выходит за пределы активной области Flash Player.

Применение

Обратите внимание, что даже после вызова *Mouse.hide( )* обычный системный текстовый курсор I появляется при прохождении мыши над доступным для редактирования текстовым полем.

Пример

Метод *hide( )* позволяет скрыть системный указатель мыши, отображаемый по умолчанию, – обычно для того, чтобы заменить его пользовательским курсором. Во Flash последний представляет собой клип, перемещающийся вслед за мышью. С помощью обработчика события *MovieClip.onMouseMove( )* можно вызвать перемещение курсора вслед за мышью, обновляя свойства клипа `_x` и `_y` при перемещении мыши. Этот прием демонстрируется в следующем коде:

```
Mouse.hide();

// Заставляет клип customPointer_mc перемещаться вслед за мышью.
// Обратите внимание, что для работы этого примера customPointer_mc должен находиться
// на главной временной диаграмме. Чтобы поместить customPointer_mc на другую
// временную диаграмму, пришлось бы с помощью globalToLocal() преобразовывать координаты
// главной временной диаграммы в локальное координатное пространство customPointer_mc.
customPointer_mc.onMouseMove = function () {
  this._x = _root._xmouse;
  this._y = _root._ymouse;
  // Сразу обновить экран, не ожидая следующего кадра.
  updateAfterEvent();
}
```

В этом коде не задействован приемник *Mouse.onMouseMove( )*, поскольку во Flash 6 *updateAfterEvent( )* работает только с событиями мыши и клавиатуры *MovieClip* или с функцией *setInterval( )*.

Иногда желательно скрыть пользовательский курсор, когда мышь не действует, например, если курсор вышел за пределы активной области Flash Player, потому что тогда на экране оказываются два курсора (пользовательский и системный). В следующем примере показано, как изменить предыдущий код, чтобы скрыть курсор после 5 секунд бездействия:

```
Mouse.hide();

// Заставляет клип customPointer_mc перемещаться вслед за мышью.
customPointer_mc.onMouseMove = function () {
  this._x = _root._xmouse;
  this._y = _root._ymouse;
  this.lastMove = getTimer();
  // Сразу обновить экран, не ожидая следующего кадра.
  updateAfterEvent();
}

customPointer_mc.onEnterFrame = function () {
  if (getTimer() - this.lastMove > 5000) {
    this._visible = false;
  } else {
    this._visible = true;
  }
}
```

Изменение курсора на значок «ладонь», когда мышь находится над кнопкой, осуществляется при помощи события кнопки *rollOver*, вызывающего переход клипа пользовательского указателя в кадр, содержащий изображение ладони, и событие *rollOut*, чтобы пользовательский указатель принял вид по умолчанию:

```
go_btn.onRollOut = function () {
  _root.customPointer.gotoAndStop("hand");
}

go_btn.onRollOut = function () {
  _root.customPointer.gotoAndStop("default");
}
```

Не забывайте поместить пользовательский курсор в верхний слой фильма, чтобы он отображался поверх любого другого содержимого. Другая возможность заключается

в динамическом генерировании клипа пользовательского курсора с помощью *duplicateMovieClip( )* или *attachMovie( )* с присвоением ему очень высокого уровня.

См. также *Mouse.show( )*, `MovieClip._xmouse`, `MovieClip._ymouse`

Mouse.onMouseDown(), событие приемника

Flash 6

происходит при нажатии главной кнопки мыши

`listener.onMouseDown()`

Описание

Подобно событию *Button.onPress( )*, событие *onMouseDown( )* обнаруживает нажатие кнопки мыши. В отличие от события кнопки *press*, *onMouseDown( )* не привязано к активной области объекта кнопки Flash (активная область кнопки определяется графически при создании кнопки в среде разработки Flash, тогда как активная область клипа может быть также задана с помощью свойства `hitArea`). Событие *onMouseDown( )* происходит при каждом нажатии главной кнопки мыши во время нахождения указателя мыши над любым участком сцены.

Чтобы ответить на событие *onMouseDown( )*, надо присвоить функцию обратного вызова свойству `onMouseDown` какого-нибудь объекта, скажем, *listener*, а затем добавить *listener* в качестве приемника с помощью *Mouse.addListener( )*, например:

```
// Создать общий объект (хотя допускается объект любого типа)
mouseListener = new Object();

// Сначала присвоим функцию обратного вызова свойству onMouseDown объекта
mouseListener.onMouseDown = function () {
  trace("The mouse was pressed!");
}

// Теперь зарегистрируем приемник в объекте Mouse
Mouse.addListener(mouseListener);
```

На событие *onMouseDown( )* может отвечать любое количество объектов-приемников, принадлежащих любому классу.

Событие *onMouseDown( )* выполняется немедленно при нажатии клавиши, независимо от кода на временной диаграмме фильма. Однако изменения вида фильма, осуществленные приемником, не отображаются до момента очередного обновления кадра, определяемого частотой кадров фильма. К сожалению, во Flash 6 *updateAfterEvent( )* нельзя применять для принудительного обновления экрана из приемника *onMouseDown( )*, поскольку *updateAfterEvent( )* действует только при вызове из события мыши или клавиатуры клипа, или из функции обратного вызова *setInterval( )*. Например:

```
someMovieClip_mc.onMouseDown = function () {
  // Увеличить клип до 200% от начального размера.
  this._xscale = this._yscale = 200;
  // Обновить экран.  Во Flash 6 это не действует в приемнике мыши.
  updateAfterEvent();
}
```

См. также

Button.onPress( ), *Mouse.addListener( )*, *Mouse.onMouseMove( )*, *Mouse.onMouseUp( )*, *MovieClip.onMouseDown( )*

Mouse.onMouseMove(), событие приемника

Flash 6

происходит при перемещении курсора мыши

listener.onMouseMove()

Описание

Событие *onMouseMove( )* позволяет обнаружить изменение положения курсора мыши. Если мышь находится в движении, события *onMouseMove( )* генерируются многократно с максимальной частотой, которую позволяет процессор. Каждое событие *onMouseMove( )* означает, что указатель мыши изменил свое положение. Узнать это положение относительно любого клипа можно с помощью свойств клипа _xmouse и _ymouse. Положение указателя мыши относительно главной сцены определяется посредством _root._xmouse и _root._ymouse.

Чтобы ответить на событие *onMouseMove( )*, надо присвоить функцию обратного вызова свойству onMouseMove какого-нибудь объекта, скажем, *listener*, а затем добавить *listener* в качестве приемника с помощью *Mouse.addListener( )*, например:

```
// Создать общий объект (хотя допускается объект любого типа)
mouseListener = new Object();

// Присвоить функцию обратного вызова свойству onMouseMove объекта
mouseListener.onMouseMove = function () {
  trace("The mouse was moved!");
}

// Зарегистрировать приемник в объекте Mouse
Mouse.addListener(mouseListener);
```

На событие *onMouseMove( )* может отвечать любое количество объектов-приемников, принадлежащих любому классу.

Событием *onMouseMove( )* удобно пользоваться в коде, призванном оживлять приложения, находящиеся в ждущем режиме (отображать шлейф мыши или создавать пользовательские указатели). Однако в приложениях Flash 6 следует предпочесть обработчик *MovieClip. onMouseMove( )*, а не *Mouse.onMouseMove( )*, поскольку он допускает обновление экрана в промежутке между кадрами.

Событие *onMouseMove( )* выполняется немедленно при нажатии клавиши, независимо от кода на временной диаграмме фильма. Однако изменения вида фильма, осуществленные приемником, не видны до момента очередного обновления кадра, определяемого частотой кадров фильма. К сожалению, во Flash 6 *updateAfterEvent( )* нельзя применять для принудительного обновления экрана из приемника *onMouseMove( )*, поскольку *updateAfterEvent( )* действует только при вызове из события мыши или клавиатуры клипа, или из функции обратного вызова *setInterval( )*. Поэтому во Flash 6 часто имеет смысл создать клип, задача которого состоит только в реализации *onMouseMove( )* с обновлением экрана. Например:

```
// Создать клип
this.createEmptyMovieClip("mouseListener_mc", 1);
// Присвоить функцию обратного вызова обработчику onMouseMove
mouseListener_mc.onMouseMove = function () {
  // Реакция на перемещение мыши (детали опущены)...затем обновление экрана
  updateAfterEvent();
}
```

Код с пользовательским указателем мыши см. в описании *Mouse.hide( )*. Код различной степени сложности для создания шлейфа мыши можно найти в сетевом хранилище кода.

См. также

Mouse.addListener( ), *Mouse.onMouseDown( )*, *Mouse.onMouseUp( )*, *MovieClip.onMouseMove( )*

Mouse.onMouseUp(), событие приемника

Flash 6

происходит, когда отпускается главная кнопка мыши

```
listener.onMouseUp()
```

Описание

Событие *onMouseUp( )* дополняет *onMouseDown( )*. Оно происходит, если сначала нажать главную кнопку мыши (при этом указатель мыши находится над каким-то участком сцены), а затем отпустить ее.

Чтобы ответить на событие *onMouseUp( )*, надо присвоить функцию обратного вызова свойству `onMouseUp` какого-нибудь объекта, скажем, *listener*, а затем добавить *listener* в качестве приемника с помощью *Mouse.addListener( )*, например:

```
// Создать общий объект (хотя допускается объект любого типа)
mouseListener = new Object();

// Присвоить функцию обратного вызова свойству onMouseUp объекта
mouseListener.onMouseUp = function () {
  trace("The mouse was released!");
}

// Зарегистрировать приемник в объекте Mouse
Mouse.addListener(mouseListener);
```

На событие *onMouseUp( )* может отвечать любое количество объектов-приемников, принадлежащих любому классу.

Событие *onMouseUp( )* выполняется немедленно при нажатии клавиши, независимо от кода на временной диаграмме фильма. Однако изменения вида фильма, осуществленные приемником, не видны до момента очередного обновления кадра, определяемого частотой кадров фильма. К сожалению, во Flash 6 *updateAfterEvent( )* нельзя применять для принудительного обновления экрана из приемника *onMouseUp( )*, поскольку *updateAfterEvent( )* действует только при вызове из события мыши или клавиатуры клипа, или из функции обратного вызова *setInterval( )*.

См. также

Button.onRelease( ), *Mouse.addListener( )*, *Mouse.onMouseDown( )*, *Mouse.onMouseMove( )*, *MovieClip.onMouseUp( )*

Mouse.show(), метод

Flash 5

делает курсор мыши видимым

```
Mouse.show()
```

Возвращаемое значение

Возвращаемое значение указывает, что курсор мыши был виден (1) или скрыт (0) перед вызовом *show( )*.

Описание

Метод *show( )* вызывает возвращение системного курсора мыши по умолчанию, спрятанного при вызове *hide( )*, и позволяет выводить обычный курсор в некоторые моменты фильма, например, когда пользователь должен заполнить форму.

См. также

Mouse.hide( )

Mouse.removeListener(), метод

Flash 6

отменяет отправку уведомлений о событии указанному приемнику

```
Mouse.removeListener(listener)
```

Аргументы

listener

Объект, ранее зарегистрированный в качестве приемника для *Mouse*.

Возвращаемое значение

Булево значение `true`, если `listener` найден и удален, и `false`, если `listener` не был зарегистрирован в качестве приемника событий для *Mouse*. Возвращаемое значение полезно при отладке кода, удаляющего приемники.

Описание

Метод *removeListener( )* прекращает получение объектом `listener` уведомлений о событиях *Mouse*. Применять его следует только после того, как `listener` будет зарегистрирован в качестве приемника событий с помощью *addListener( )*. Объект приемника не уничтожается и может многократно удаляться и добавляться. Приемники событий следует удалять, только если выполняемая ими задача больше не нужна.

См. также

Mouse.addListener( ), *Mouse.onMouseDown( )*, *Mouse.onMouseMove( )*, *Mouse.onMouseUp( )*; глава 10

MovieClip, класс

Flash 3; усилен во Flash 4, Flash 5 и Flash 6

тип данных для главных фильмов и клипов, похожий на класс

Конструктор

```
new MovieClip();
```

Синтаксис конструктора применяется только для создания подклассов *MovieClip*. Конструктор не создает клипов на сцене! Экземпляр клипа создается методами *attachMovie( )*, *createEmptyMovieClip( )* или *duplicateMovieClip( )*. Символы *MovieClip*, на которых базируются экземпляры, должны создаваться вручную в среде разработки Flash.

Свойства

Свойства клипов описывают и контролируют различные физические характеристики клипов и главных фильмов. Доступ к свойствам производится посредством опера-

тора «точка» или [], как и для любых других объектов. О применении свойств клипов подробно рассказано в разделе «„Объектность“ клипов» главы 13.

Следует отметить три момента, связанных со свойствами:

- При изменении физического свойства объекта *MovieClip* он начинает управляться программно, разрывая связь со своей внутренней временной диаграммой. В результате прекращается действие трансформаций, происходящих в этот момент.
- Некоторые свойства MovieClip являются числами с плавающей точкой, но внутренне отображаются в другие форматы. Свойство _alpha, к примеру, внутренне отображается в целое число, принимающее значение от 0 до 255. Это приводит к потере точности между установкой и чтением некоторых свойств. Можно, например, установить значение свойства равным 90 и сразу после этого прочесть его как 89,84375. Если небольшие отклонения в значениях свойств играют роль в сценарии, необходимо вручную округлять эти значения после извлечения с помощью *Math.round( )*, *Math.floor( )* или *Math.ceil( )* либо записывать исходное значение в отдельную от свойства переменную.
- Встроенные свойства *MovieClip* не перечисляются в цикле *for-in*.

В табл. С.11 перечислены свойства класса *MovieClip*. Свойства клипа доступны для любого экземпляра клипа и, в большинстве случаев, на главной временной диаграмме фильма (_root). Те из них, которыми нельзя пользоваться на главной временной диаграмме фильма, помечены в таблице символом «*».

До Flash 6 все встроенные свойства клипов начинались с символа подчеркивания. Для большей согласованности с другими языками стандарта ECMA-262 свойства, добавленные во Flash 6 и не аналогичные старым свойствам Flash 5 *Button*, таким как _focusRect, не следуют соглашению о символе подчеркивания. Свойства, введенные во Flash 6, помечены в табл. С.11 символом «†».

В колонке *Доступ* табл. С.11 «чтение/запись» указывает на возможность как чтения, так и записи значения свойства, а «только чтение» указывает, что значение свойства можно извлекать, но не задавать (доступно только для чтения). Некоторым доступным только для чтения свойствам можно косвенным образом присваивать значения в среде разработки или некоторой связанной с ними функции (например, *gotoAndStop( )* устанавливает свойство _currentframe), но непосредственно через ActionScript можно задавать значения только свойствам, доступным для чтения и записи. В колонке *Тип* описан тип данных значений каждого свойства. В колонке *Описание свойства* дан краткий обзор назначения свойства, но в последующих полных описаниях имеются важные детали. Ко всем свойствам текущей временной диаграммы можно обращаться без явной ссылки на нее, например писать _alpha вместо *someClip.*_alpha.

Таблица С.11. Сводка свойств клипов

Имя свойства	Доступ	Тип	Описание свойства
_alpha	чтение/запись	числовой	Процент непрозрачности: 0 – прозрачный; 100 – непрозрачный
_currentframe	только чтение	числовой	Номер кадра, в котором находится головка воспроизведения
_droptarget	только чтение	строковый	Путь к клипу, над которым находится или куда сброшен перетаскиваемый клип, в слэш-нотации

Таблица C.11 (продолжение)

Имя свойства	Доступ	Тип	Описание свойства
enabled	чтение/запись	булев	Разрешает или запрещает взаимодействие с кнопкой
focusEnabled	чтение/запись	булев	Разрешает клипу получать фокус ввода с клавиатуры программным образом
_focusrect	чтение/запись	булев	Отключает автоматический показ желтой рамки вокруг клипов в фокусе
_framesloaded	только чтение	числовой	Количество загруженных кадров
_height	чтение/запись	числовой	Высота текущего содержимого в пикселах
hitArea*†	чтение/запись	строковый	Назначает активную область для клипа, используемого как кнопка
_name*	чтение/запись	строковый	Идентификатор клипа в виде строки (не ссылки)
_parent*	только чтение	ссылка на MovieClip	Ссылка на экземпляр или фильм, содержащий текущий экземпляр
_rotation	чтение/запись	числовой	Угол поворота в градусах
tabChildren*†	чтение/запись	булев	Определяет, включаются ли дочерние объекты в автоматический обход при табуляции
tabEnabled*†	чтение/запись	булев	Включает клип в обход по табуляции или исключает из него
tabIndex*†	чтение/запись	булев	Задает индекс клипа в пользовательском порядке табуляции
_target	только чтение	строковый	Абсолютный путь к цели в слэш-нотации
_totalframes	только чтение	числовой	Количество кадров во временной диаграмме
trackAsMenu*†	чтение/запись		Модифицирует требования обработчика *onRelease( )*, включая поведение в стиле меню
_url	только чтение	строковый	Адрес исходного файла *.swf* на диске или в сети
useHandCursor*†	чтение/запись	булев	Определяет, нужно ли представлять указатель мыши над клипом в виде руки (применимо только к клипам с обработчиками кнопок)
_visible	чтение/запись	булев	Видимость: true, если показывается; false, если скрыт
_width	чтение/запись	числовой	Ширина текущего содержимого в пикселах
_x	чтение/запись	числовой	Позиция по горизонтали в пикселах
_xmouse	только чтение	числовой	Позиция курсора мыши по горизонтали в пикселах
_xscale	чтение/запись	числовой	Масштаб по горизонтали в процентах
_y	чтение/запись	числовой	Позиция по вертикали в пикселах

Имя свойства	Доступ	Тип	Описание свойства
_ymouse	только чтение	числовой	Позиция курсора мыши по вертикали в пикселах
_yscale	чтение/запись	числовой	Масштаб по вертикали в процентах

* Применимо к экземплярам клипов; неприменимо к главной временной диаграмме.

† Появилось во Flash 6; не поддерживалось в прежних версиях.

Методы

В табл. С.12 и С.13 перечислены методы класса *MovieClip*.

Методы клипов могут вызываться с любым экземпляром клипа и в большинстве случаев с главной временной диаграммой (_root). В табл. С.12 те методы, которые не могут быть использованы с главной временной диаграммой фильма, помечены звездочкой (*). Методы, появившиеся только во Flash 6, помечены в таблицах символом «†». Методы клипов, помеченные в этой таблице символом «‡», выполняют такие же действия, как аналогичные глобальные функции, но имеют более удобный синтаксис с точками *MovieClip.method()* (перед точкой указывается экземпляр клипа). Те методы, которым не соответствуют глобальные функции, могут применяться к текущему клипу без явной ссылки на него, например attachMovie() вместо myClip.attachMovie().

Таблица С.12. Сводка методов клипов

Имя метода	Описание метода
attachMovie( )	Создает новый экземпляр на основе символа, экспортированного из библиотеки текущего документа
createEmptyMovieC lip( )†	Создает новый пустой экземпляр *MovieClip*
createTextField( )†	Создает новый объект *TextField*
*duplicateMovieClip( )**‡	Создает копию экземпляра клипа
getBounds( )	Возвращает объект, свойствами которого являются координаты рамки, ограничивающей занимаемую клипом видимую область
getBytesLoaded( )	Возвращает количество загруженных байтов экземпляра или фильма. Не применяется к внутренним клипам
getBytesTotal( )	Возвращает физический размер в байтах экземпляра или главного фильма
getDepth( )†	Возвращает позицию клипа в стеке видимого контента
getURL( )‡	Загружает в броузер документ (обычно веб-страницу)
globalToLocal( )	Преобразует свойства объекта *coordinates* из координат сцены в координаты экземпляра
gotoAndPlay( )‡	Перемещает головку воспроизведения экземпляра или фильма в заданный кадр, затем воспроизводит экземпляр или фильм
gotoAndStop( )‡	Перемещает головку воспроизведения экземпляра или фильма в заданный кадр и останавливает ее

Таблица C.12 (продолжение)

Имя метода	Описание метода
hitTest( )	Возвращает булево значение, показывающее, пересекается ли клип с заданной точкой или другим клипом
loadMovie( )‡	Загружает в проигрыватель внешний файл *swf*
loadVariables( )‡	Получает внешние данные, состоящие из имен переменных и их значений, и преобразует эти данные в эквивалентные переменные ActionScript
localToGlobal( )	Преобразует свойства объекта *coordinates* из координат объекта в координаты сцены
nextFrame( )‡	Перемещает головку воспроизведения фильма или экземпляра на один кадр вперед и останавливает ее там
play( )‡	Приводит в движение головку воспроизведения экземпляра или фильма (то есть воспроизводит клип)
prevFrame( )‡	Перемещает головку воспроизведения фильма или экземпляра на один кадр назад и останавливает ее там
*removeMovieClip( )**‡	Удаляет дублированный или прикрепленный экземпляр
setMask( )†	Назначает клип маской для другого клипа
startDrag( )‡	Вызывает физическое перемещение экземпляра или фильма вслед за курсором мыши
stop( )‡	Останавливает воспроизведение экземпляра или фильма
stopDrag( )‡	Прекращает текущую операцию перетаскивания
*swapDepths( )**	Изменяет положение экземпляра в стеке контента
unloadMovie( )‡	Удаляет экземпляр или главный фильм с уровня документа или из содержащего его клипа
valueOf( )	Возвращает объектную ссылку на экземпляр

* Применим к экземплярам клипов; неприменим к главной временной диаграмме (`_root`).

† Появился во Flash 6. Не поддерживается более ранними версиями.

‡ Метод MovieClip, аналогичный одноименной глобальной функции.

Методы для черчения и заливки фигур в экземпляре клипа, так называемые *Drawing API*, также представляют собой новинку Flash 6 и приведены в табл. C.13. См. также метод *MovieClip.createEmptyMovieClip( )*, удобный для создания чистого холста, на котором можно чертить.

Таблица C.13. Методы графического Drawing API для клипов

Имя метода	Описание метода
beginFill( )†	Начинает черчение закрашенной фигуры
beginGradientFill( )†	Начинает черчение фигуры с градиентной заливкой
clear( )†	Удаляет из клипа все начерченное
curveTo( )†	Чертит кривую

Имя метода	Описание метода
endFill( )†	Завершает черчение закрашенной фигуры
lineStyle( )†	Устанавливает толщину штриха, его цвет и прозрачность
lineTo( )†	Чертит прямую линию
moveTo( )†	Перемещает перо в новое положение, не вычерчивая линии

† Появился во Flash 6. Не поддерживается более ранними версиями.

События

Экземпляры клипов поддерживают *обработчики событий*, автоматически реагирующие на определенный набор событий (таких, как взаимодействие с мышью или клавиатурой либо воспроизведение фильма). Во Flash 6 клипам доступны также события класса *Button* (например, *onPress( )*, *onRelease( )* и *onRollOver( )*), хотя главной временной диаграмме (`_root`) они не доступны. Обработчик событий клипа можно создать следующими способами:

- Прикрепить блок *onClipEvent( )* непосредственно к экземпляру клипа на этапе разработки.
- Прикрепить блок в стиле кнопки *on( )* непосредственно к экземпляру клипа на этапе разработки.
- Присвоить функцию обратного вызова соответствующему свойству-событию с помощью ActionScript.

Вот примеры применения каждого из этих приемов:

```
// Блок onClipEvent()
onClipEvent (enterFrame) {
  // предложения
}

// Блок on()
on (release) {
  // предложения
}

// Две функции обратного вызова
theClip_mc.onEnterFrame = function () {
  // предложения
}

theClip_mc.onPress = function () {
  // предложения
}
```

Функции обратного вызова не поддерживаются Flash 5, но во Flash 6 обычно лучше применять их, а не блоки *onClipEvent( )* и *on( )*. О важных различиях между типами обработчиков событий рассказано в главе 10. В частности, отметим следующее:

- При выгрузке содержимого клипа свойства-обработчики событий (например, *onEnterFrame( )*) теряются, а обработчики *onClipEvent( )* (например, *onClipEvent(enterFrame)*) сохраняются.
- При дублировании клипа с помощью *duplicateMovieClip( )* свойства-обработчики событий не копируются в дубликат, в отличие от обработчиков *onClipEvent( )*.

Поддерживаемые обработчики событий клипов перечислены в табл. C.14. Обработчики, помеченные звездочкой (*), не поддерживаются главными фильмами (`_root`). Все обработчики событий из табл. C.14 появились только во Flash 6, но если во Flash 5 для них есть эквиваленты, то они тоже показаны. За исключением *onKillFocus( )* и *onSetFocus( )*, события, приведенные без их эквивалентов во Flash 5, заимствованы от событий класса *Button*.

Таблица C.14. Сводка обработчиков событий клипов

Обработчик события клипа	Событие происходит, когда...	Эквивалент во Flash 5
onData( )	Заканчивается поток загружаемых переменных или порции загружаемого фильма	*onClipEvent(data)*
*onDragOut( )**	Кнопка мыши нажата над клипом, затем курсор уводится с клипа без освобождения кнопки	Нет
*onDragOver( )**	Кнопка мыши нажата над клипом, затем курсор уводится с клипа и возвращается обратно	Нет
onEnterFrame( )	Во Flash Player прошел один такт частоты смены кадров	*onClipEvent(enterFrame)*
onKeyDown( )	Была нажата клавиша, когда клип находился на сцене и обладал фокусом ввода	*onClipEvent(keyDown)*
onKeyUp( )	Нажатая клавиша отпущена, когда клип находился на сцене и обладал фокусом ввода	*onClipEvent(keyUp)*
*onKillFocus( )**	Клип потерял фокус ввода	Нет
*onLoad( )**	Клип впервые появляется на сцене или закончена загрузка в клип файла *.swf*	*onClipEvent(load)*
onMouseDown ( )	Главная кнопка мыши нажата во время нахождения клипа на сцене	*onClipEvent(mouseDown)*
onMouseMove( )	Курсор мыши переместился (хоть самую малость) во время нахождения клипа на сцене	*onClipEvent(mouseMove)*
onMouseUp( )	Главная кнопка мыши нажата и отпущена во время нахождения клипа на сцене	*onClipEvent(mouseUp)*
*onPress( )**	Кнопка мыши нажата над клипом	Нет
onRelease( )	Кнопка мыши нажата и затем отпущена над клипом	Нет
*onReleaseOutsi de( )**	Кнопка мыши нажата над клипом и затем отпущена после увода курсора с клипа	Нет
*onRollOut( )**	Курсор уводится с клипа при не нажатой кнопке	Нет
*onRollOver( )**	Курсор перемещается на клип при не нажатой кнопке	Нет
*onSetFocus( )**	Клип получает фокус ввода	Нет
*onUnload( )**	Клип удаляется со сцены	*onClipEvent(unload)*

* Применимо только к экземплярам клипов; неприменимо к главной временной диаграмме (`_root`).

Описание

MovieClip в ActionScript – это не просто класс, а особый тип данных, предоставляющий возможность получать информацию о фильмах и клипах и управлять ими. В большинстве случаев с клипами и фильмами работают как с объектами: можно создавать свойства клипов и обращаться к ним, можно создавать и вызывать методы клипов. Класс *MovieClip* подробно рассмотрен в главе 13.

Для создания новых экземпляров *MovieClip* конструктор не применяется. Символы клипов создаются в среде разработки, и их экземпляры вручную помещаются на сцену. Однако есть методы, позволяющие копировать существующие клипы (*duplicateMovieClip( )*) или программно помещать новые клипы на сцену (*attachMovie( )* и *createEmptyMovieClip( )*).

Не все объекты *MovieClip* равны между собой: некоторые методы и свойства *MovieClip* применимы только к экземплярам клипов, а не к главным фильмам (главный фильм – это временная диаграмма `_root` документа *.swf*). При рассмотрении свойств и методов *MovieClip* будут отмечены случаи, когда их действие ограничено только одним типом объектов *MovieClip*. Обратите внимание, что «класс» объектов обозначается словом *MovieClip*, тип данных ActionScript обозначается словом *movieclip* (строчными буквами); *клип* или *экземпляр* обозначают конкретный клип, а слово *фильм* употребляется для ссылки на главный фильм файла *.swf*. В кратком описании каждого детализируемого свойства или метода *mc* обозначает имя клипа или главного фильма. Для многих свойств и методов *mc* может отсутствовать: если оно опущено, используется текущая временная диаграмма (см. табл. С.12).

В данной статье, посвященной *MovieClip*, говоря о координатах, необходимо ссылаться на местоположение экземпляров клипов. Положение клипа измеряется относительно одной характерной точки, так называемой *точки регистрации*, отмеченной перекрестием в библиотечном символе клипа.

Если клип расположен на главной сцене, то его позиция описывается относительно левого верхнего угла сцены, соответствующего точке (0,0). Если клип располагается внутри другого клипа, то его позиция описывается относительно точки регистрации родительского клипа, которой опять-таки соответствует точка (0,0). В обоих случаях точка (0,0) представляет собой *начальную точку* (*origin*) пространства координат, в котором отображается положение клипа. Как будет показано, преобразование между этими двумя пространствами координат обеспечивается методами *localToGlobal( )* и *globalToLocal( )*.

В системе координат Flash ось Y декартовых координат перевернута, то есть значения y возрастают вниз, а не вверх. Например, y-координата 100 указывает на точку, находящуюся на 100 пикселов *ниже* оси X.

Свойства и методы, связанные с координатами, часто будут применяться для перемещения клипа, определения положения клипа или определения места, в котором он пересекается с другим объектом или точкой. Последний прием называется *обнаружением коллизий* (*collision detection*), поскольку с его помощью часто определяют, не надо ли изменить направление движения клипа, как если бы он отскочил от другого объекта (см. метод *MovieClip.hitTest( )*).

Обратите внимание, что в ActionScript нет собственного типа данных для представления *точки* (то есть координат x и y). В описании *MovieClip.globalToLocal( )* объясняется, как создать объект точки из общего объекта.

Во Flash 5 не было прямого API для создания в ActionScript подклассов класса *MovieClip*. Во Flash 6 появился метод *Object.registerClass( )*, позволяющий зарегистрировать пользовательский класс в качестве подкласса *MovieClip*. Подробное обсуждение можно найти в главе 14.

Во Flash 6 экземпляры клипов получили возможность вести себя в точности как кнопки. Чтобы реализовать для клипа поведение, характерное для кнопки, надо определить один или несколько следующих обработчиков событий кнопок: *onDragOut( )*, *onDragOver( )*, *onPress( )*, *onRelease( )*, *onReleaseOutside( )*, *onRollOut( )*, *onRollOver( )* (см. табл. С.14). После того как для клипа определен обработчик события кнопки, начинают действовать следующие свойства, относящиеся к поведению кнопок: `enabled`, `hitArea`, `trackAsMenu` и `useHandCursor`. Присвоить этим свойствам значения можно в любой момент, но действовать они станут только тогда, когда будет определен хотя бы один обработчик события кнопки. Чтобы обеспечить для клипа состояния Up, Over и Down в стиле кнопок, создайте кадры с особыми метками `"_up"`, `"_over"` и `"_down"`. Определить для клипа активную область, как для кнопки, позволяет свойство `hitArea`. Если не определить свойство `hitArea`, то в качестве активной области будет выступать собственное содержимое клипа. Учтите, что хотя в документации Macromedia клипы с обработчиками кнопок называются «button movie clips», символа такого типа или класса нет, а термин «button movie clip» просто означает в документации Macromedia клип, для которого определен хотя бы один обработчик кнопки. См. «Применение клипов в качестве кнопок» в главе 13.

См. также Глава 13, глава 14

MovieClip._alpha, свойство

Flash 4

степень непрозрачности клипа или фильма **чтение/запись**

mc.`_alpha`

Описание

Свойство `_alpha`, выражаемое числом с плавающей точкой, задает степень непрозрачности (или, наоборот, прозрачности) *mc* в процентах: 0 соответствует полной прозрачности, а 100 – полной непрозрачности. Задание `_alpha` для *mc* воздействует на видимую прозрачность всех клипов, вложенных в *mc*, но не изменяет их свойства `_alpha`. Это значит, например, что если взять клип `square`, содержащий другой клип, `circle`, и присвоить `square._alpha` значение 50, то `circle` будет прозрачен на 50%, но значение `_alpha` для него останется равным 100. Однако эффект задания значения `_alpha` для обоих клипов – `circle` и его родительского (`square`) – оказывается кумулятивным. Если значение `_alpha` для каждого из них равно 50, то `circle` показывается на экране с непрозрачностью 25% (75% прозрачности). По этой причине не задавайте значение `_alpha` для вложенных клипов, если только не требуется, чтобы их прозрачность отличалась от прозрачности клипов одного с ними уровня вложенности. Короче, задавайте значение `_alpha` только для родительских клипов.

Применение

Обратите внимание, что установка свойства `_alpha` для клипа оказывает влияние на свойство `aa` объекта, возвращаемого *Color.getTransform( )*.

Пример

```
ball_mc._alpha = 60;   // Сделать ball_mc частично прозрачным
ball_mc._alpha = 0;    // Сделать ball_mc невидимым
```

Следующий обработчик события клипа увеличивает прозрачность `theClip_mc` по мере перемещения мыши вниз по экрану:

```
theClip_mcEvent(enterFrame) {
  this._alpha = 100 - (_root._ymouse / 400) * 100;
}
```

См. также *Color* класс; *MovieClip*.`_visible`

MovieClip.attachMovie(), метод

Flash 5; во Flash 6 добавлены initObject и возвращаемое значение

создает новый экземпляр клипа из экспортированного символа библиотеки

mc.attachMovie(*symbolIdentifier*, *newName*, *depth*)
mc.attachMovie(*symbolIdentifier*, *newName*, *depth*, *initObject*)

Аргументы

symbolIdentifier

Строка, задающая идентификатор связи экспортированного символа клипа, как он задан в библиотеке в Options → Linkage.

newName

Строка, задающая имя экземпляра вновь создаваемого клипа. Имя должно соответствовать правилам построения идентификаторов, описанным в разделе «Идентификаторы» главы 15.

depth

Целое число от –16 384 до 1 048 575 включительно, задающее уровень, на который должен быть помещен новый клип в *mc*. Клипы с большей глубиной закрывают клипы с меньшей глубиной, поэтому клип с глубиной 6 находится перед клипом с глубиной 5, который находится перед клипом с глубиной 4, и т. д. Глубины, большие –1, зарезервированы для динамически создаваемого содержимого, поэтому, как правило, в *attachMovie( )* задается значение глубины от 0 до 1 048 575. Однако для динамического содержимого, которое должно оказаться в *mc* ниже любого содержимого среды разработки (например, для фона этапа исполнения), следует задавать глубину –16 384. Глубины от –16 383 до –1 включительно зарезервированы для содержимого, создаваемого на этапе разработки, и не должны использоваться в *attachMovie( )*. Важные подробности можно найти в разделе «Добавление в стек клипов, созданных с помощью *attachMovie( )*» главы 13.

initObject

Объект, свойства которого будут скопированы в новый экземпляр клипа. Не поддерживается во Flash 5.

Возвращаемое значение

Во Flash 6 *attachMovie( )* возвращает объектную ссылку на прикрепленный клип или `undefined`, если создать клип не удалось (во Flash 5 всегда возвращалось значение `undefined`).

Описание

Метод *attachMovie( )* создает новый экземпляр с именем *newName* на основе экспортированного символа клипа, указанного в *symbolIdentifier*. Если *mc* является главным фильмом, то новый экземпляр помещается в верхний левый угол сцены. Если же

mc – это экземпляр клипа, то новый экземпляр помещается в точку регистрации *mc*. В любом случае новый экземпляр располагается в стеке видимого содержимого *mc* на уровне, заданном *depth*. Так, следующий код прикрепляет экспортированный из библиотеки клип с идентификатором связи "ball_symbol". Новый экземпляр получает имя tennisBall_mc и помещается внутри court_mc на уровень 1, над любым содержимым, созданным во время разработки court_mc:

```
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1);
```

После этого мы получаем возможность управления новым экземпляром:

```
// Удвоить ширину шарика
court_mc.tennisBall_mc._xscale = 200;
```

Во Flash 6 *attachMovie( )* стал возвращать ссылку на новый экземпляр, поэтому можно написать более короткий код:

```
tBall = court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1);
tBall_mc._xscale = 200;
```

и даже еще более короткий (но менее понятный):

```
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1)._xscale = 200;
```

Для удаления прикрепленного экземпляра применяется метод *MovieClip.removeMovieClip( )* (важные подробности можно найти в описании глобальной функции *removeMovieClip( )*).

Во Flash 5 все экспортированные клипы загружаются раньше, чем первый кадр фильма, вызывая задержку загрузки фильма. Во Flash 6 экспортированные кадры можно загружать в другие кадры, для чего необходимо:

1. В верхнем правом углу окна Library выбрать Linkage во всплывающем меню Options.
2. В окне диалога Linkage Properties сбросить флажок Export in first frame, затем выбрать OK.
3. На главной временной диаграмме в кадре, куда должен быть загружен экспортированный клип, создать ключевой кадр и поместить на него экземпляр клипа.

Клип недоступен методу *attachMovie( )* до того кадра, в который он экспортирован.

Если задан необязательный параметр *initObject*, то его *прямые свойства* (те, которые прикреплены непосредственно к *initObject*, а не получены по цепочке прототипов) копируются в новый клип и сразу становятся доступными. Например:

```
// Создать новый объект для передачи свойств новому экземпляру tennisBall_mc
var tempObj = new Object();
// Добавить свойство speed для передачи tennisBall_mc
tempObj.speed = 200;
// Передать tempObj в attachMovie()
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1, tempObj);
// Проверить значение speed у tennisBall_mc
trace(court_mc.tennisBall_mc.speed);  // Выводит: 200
```

Однако свойства, унаследованные по цепочке прототипов *initObject*, не копируются в клип. Узнать, унаследованным или прямым является свойство, можно с помощью *Object.hasOwnProperty( )*. Посредством *initObject* можно присваивать значения

встроенным свойствам клипа, таким как _x и _rotation,[1] а также обработчикам событий, таким как *onEnterFrame( )* (то есть копировать их во вновь созданный экземпляр клипа). Например:

```
var tempObj = new Object();
// Задать угол поворота
tempObj._rotation = 200;
// Присвоить значение функции обратного вызова onEnterFrame()
tempObj.onEnterFrame = function () {
  this._x++
};
// После прикрепления tennisBall_mc будет повернут на 200 градусов и станет
// перемещаться на один пиксел вправо в каждом новом кадре обработчиком onEnterFrame()
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1, tempObj);
```

Во Flash 6 параметр *initObject* служит для присваивания значений свойствам newName, которые будут использованы его конструктором класса (что равносильно передаче аргументов конструктору). Конструктор класса символа клипа определяется методом *Object.registerClass( )*. Для удобства можно задать *initObject* в виде анонимного объекта, например:

```
court_mc.attachMovie("ball_symbol", "tennisBall_mc", 1, {speed: 200, _x: 165} );
```

Параметр *initObject* не поддерживается в версиях, предшествующих Flash 6.

См. также

duplicateMovieClip( ), *MovieClip.createEmptyMovieClip( )*, *MovieClip.duplicateMovieClip( )*; «Создание экземпляров с помощью attachMovie()» и «Добавление в стек клипов, созданных с помощью duplicateMovieClip() и createEmptyMovieClip()» в главе 13

MovieClip.beginFill(), метод

Flash 6

начинает вычерчивать фигуру со сплошной заливкой

mc.beginFill(*RGB*, *alpha*)

Аргументы

RGB

Целое число, задающее цвет фигуры, обычно в виде шестнадцатеричной тройки RGB – 0x*RRGGBB*, где *RR*, *GG* и *BB* – двузначные шестнадцатеричные цифры, представляющие красную, зеленую и синюю составляющие. Например, шестнадцатеричное число 0xFF0000 задает красный цвет. Если *RGB* опущен или имеет значение undefined, заливка не применяется. Подробно о тройках RGB рассказано в описании класса *Color*.

alpha

Целое число от 0 до 100, задающее степень непрозрачности (или, наоборот, прозрачности) фигуры, в процентах: 0 соответствует полной прозрачности, а 100 –

[1] Стоит отметить, что в отношении унаследованных от класса свойств дело обстоит несколько неоднозначно. Свойство *initObject* позволяет задать клипу свойства _x, _y, _rotation, _xscale, _yscale и т. п., хотя эти свойства – наследственные (убедитесь в этом при помощи метода *hasOwnProperty( )*). Но нельзя задать свойства _width и _height: если в *initObject* задать для клипа высоту или ширину, то он просто не отобразится. Это вполне можно считать ошибкой реализации. – *Примеч. науч. ред.*

полной непрозрачности. По умолчанию принимается равным 100. Значения, большие 100, интерпретируются как 100, а меньшие 0 интерпретируются как 0.

Описание

Входящий в состав Drawing API метод *beginFill( )* начинает вычерчивание внутри `mc` новой фигуры со сплошной заливкой `RGB` и прозрачностью `alpha`. Эта фигура чертится поверх любого содержимого `mc`, созданного Drawing API, но под всем остальным содержимым `mc` (например, клипами, фигурами, текстом и кнопками, размещенными на стадии разработки, либо содержимым, прикрепленным на этапе исполнения с помощью *attachMovie( )*, *duplicateMovieClip( )* или *createTextField( )*). Для черчения фигуры с градиентной заливкой применяется *beginGradientFill( )*.

Метод *beginFill( )* составляет лишь одну часть процедуры вычерчивания фигуры. Drawing API можно представить себе как старый перьевой графопостроитель (вроде тех, на которых исполнялись архитектурные чертежи), оснащенный воображаемым пером, которое можно поднимать, опускать и перемещать, чтобы вычерчивать прямые и кривые линии. Чтобы начертить закрашенную фигуру, *beginFill( )* следует применять в сочетании с другими методами черчения *MovieClip*, например:

1. Установить перо в нужное положение с помощью *moveTo( )*.
2. Начать черчение фигуры, вызвав метод *beginFill( )*.
3. Очертить фигуру с помощью ряда вызовов *lineTo( )* и/или *curveTo( )*, последний из которых должен закончиться в той точке, которая была задана в начальном вызове *moveTo( )*.
4. Завершить вычерчивание фигуры, вызвав *endFill( )*.

Чертить с помощью Drawing API можно в существующем или пустом клипе, созданном в среде разработки или программным образом. Чтобы начать чертить на чистом холсте, вызовите метод *MovieClip.createEmptyMovieClip( )*. Можно чертить на главной временной диаграмме, если указать `_root` в качестве целевого клипа, но лучше делать это в клипе, прикрепленном к главной временной диаграмме.

Следующий код создает клип `drawing_mc` и вычерчивает в нем красный треугольник в 100 пикселах вправо вниз от точки регистрации клипа:

```
// Создать клип
this.createEmptyMovieClip("drawing_mc", 1);
// Установить перо
var startX = 100;
var startY = 100;
drawing_mc.moveTo(startX , startY);
// Начать черчение фигуры
drawing_mc.beginFill(0xFF0000, 100);
// Начертить стороны треугольника
drawing_mc.lineTo(startX+100, startY );     // Вправо на 100 пикселов
drawing_mc.lineTo(startX+50, startY+100);   // Влево на 50 пикселов, вниз 100 пикселов
drawing_mc.lineTo(startX, startY);          // Возвратиться в начальную точку,
                                            // замкнув фигуру
// Завершить черчение
drawing_mc.endFill();
```

Контур фигуры автоматически вычерчивается тем типом линии, который был установлен при последнем вызове *lineStyle( )*. Если *lineStyle( )* ни разу не вызывался, контур фигуры не обводится. Например, следующий код вычерчивает тот же красный треугольник, но с черной линией толщиной в один пиксел по периметру. На этот раз мы зададим позиции с помощью числовых литералов вместо переменных `startX` и `startY`:

```
this.createEmptyMovieClip("drawing_mc", 1);
// Обвести фигуру черной линией толщиной в 1 пиксел
drawing_mc.lineStyle(1, 0x000000);
drawing_mc.moveTo(100, 100);
drawing_mc.beginFill(0xFF0000, 100);
drawing_mc.lineTo(200, 100);
drawing_mc.lineTo(150, 200);
drawing_mc.lineTo(100, 100);
drawing_mc.endFill();
```

Flash пытается исправить неправильно заданные фигуры (например, фигуры без метода *endFill( )*), но надежные результаты можно получить только при соблюдении следующих правил черчения:

- Начальная и конечная точки периметра фигуры должны совпадать.
- Черчение фигуры должно завершаться вызовом *endFill( )*.
- Вызову *beginFill( )* должен непосредственно предшествовать вызов *moveTo( )*, определяющий начальную точку закрашиваемой фигуры.

Например, следующий код проводит линию в точку (100, 100), а затем чертит наш красный треугольник, начав с этой точки. Перед черчением треугольника *moveTo( )* устанавливает в качестве отправной точку (100, 100), несмотря на то что перемещать перо не требуется. Если опустить вызов *moveTo( )* или выполнить его после вызова *beginFill( )*, Flash неправильно отобразит линию и треугольник.

```
this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc.lineStyle(1);
drawing_mc.lineTo(100, 100);  // Начертить линию
// Теперь начертим треугольник
drawing_mc.moveTo(100, 100);  // Задание начальной точки треугольника обязательно!
drawing_mc.beginFill(0xFF0000);
drawing_mc.lineTo(200, 100);
drawing_mc.lineTo(150, 200);
drawing_mc.lineTo(100, 100);
drawing_mc.endFill();
```

Клипы, в которых есть содержимое, начерченное с помощью ActionScript, ведут себя точно так же, как клипы, содержимое которых создано в среде разработки Flash MX. Например, свойства `_width` и `_height` отражают содержимое, начерченное на этапе исполнения, как и вызовы *getBounds( )* и *hitTest( )*. Более того, при условии, что они не являются полностью прозрачными, фигуры, созданные с помощью *beginFill( )* или *beginGradientFill( )*, могут быть использованы в качестве активных областей для событий *onRollover( )* и *onPress( )* либо в качестве маски (см. *MovieClip.setMask( )*).

Чтобы начертить фигуру с отверстием, нарисуйте саму фигуру и отверстие в ней в рамках одного блока *beginFill( )/endFill( )*. Например, следующий код вычерчивает «квадратный бублик»:

```
this.lineStyle(3);
this.beginFill(0xFF0000, 100);
  // Первый квадрат
```

```
    this.moveTo(100, 100);
    this.lineTo(200, 100);
    this.lineTo(200, 200);
    this.lineTo(100, 200);
    this.lineTo(100, 100);

    // Второй квадрат(дырка в бублике!)
    this.moveTo(150, 150);
    this.lineTo(170, 150);
    this.lineTo(170, 170);
    this.lineTo(150, 170);
    this.lineTo(150, 150);
  this.endFill;
```

Обратите внимание, что растровая заливка не поддерживается в Drawing API непосредственно, но аналогичный эффект можно получить, загрузив графический образ в клип, а затем начертив фигуру во втором клипе и применив его в качестве маски для клипа с образом. Клип с образом должен быть полностью загружен, прежде чем будет применена маска. Например:

```
// Создать клип маски
this.createEmptyMovieClip("mask_mc", 2 );

// Начертить маску
mask_mc.beginFill(0xFF0000);
mask_mc.moveTo(100,100);
mask_mc.lineTo(200,100);
mask_mc.lineTo(200,200);
mask_mc.lineTo(100,200);
mask_mc.lineTo(100,100);
mask_mc.endFill();

// Созать клип с графическим образом, который будет маскирован
this.createEmptyMovieClip("image_mc", 1);
// Загрузить маскируемый образ
image_mc.loadMovie("image.jpg");

// Применить маску, когда образ загружен
this.onEnterFrame = function () {
  // Ждать конца загрузки образа
  if (image_mc._width > 0 && image_mc._totalframes == image_mc._framesloaded) {
    // Образ загружен, поэтому применить маску
    this.setMask(mask_mc);
    // Прекратить проверку загрузки образа
    delete this.onEnterFrame;
  }
}
```

Пример

В Drawing API нет методов для вычерчивания элементарных фигур, таких как треугольники, окружности и многоугольники. Их должны создавать сами разработчики ActionScript, как показано в примере C.2, где методы *drawRect( )* и *drawCircle( )* добавляются во все клипы.

Пример C.2. Пользовательские методы drawRect( ) и drawCircle( )

```
//     Метод: MovieClip.drawRect()
//  Описание: Вычерчивает прямоугольник с возможностью заливки.
```

```
// Параметры: x          Горизонтальная координата левого верхнего угла.
//            y          Вертикальная координата левого верхнего угла.
//            w          Ширина прямоугольника в пикселах.
//            h          Высота прямоугольника в пикселах.
//            RGB        Цвет прямоугольника.
//            alpha      Прозрачность прямоугольника.
MovieClip.prototype.drawRect = function (x, y, w, h, RGB, alpha) {
  this.moveTo(x, y);
  this.beginFill(RGB, alpha);
  this.lineTo(x+w, y);
  this.lineTo(x+w, y+h);
  this.lineTo(x, y+h);
  this.lineTo(x, y);
  this.endFill;
};

// Применение drawRect():
// Задать толщину линии в 3 пиксела.
this.lineStyle(3);
// Начертить перекрывающиеся прямоугольники с эффектом прозрачности.
this.drawRect(300, 200, 50, 20, 0x0000FF, 100);
this.drawRect(320, 210, 100, 120, 0x00FF00, 60);

//     Метод: MovieClip.drawCircle()
//  Описание: Вычерчивает окружность (без заливки) как последовательность n кривых.
// Параметры: x         Горизонтальная координата центра окружности.
//            y         Вертикальная координата центра окружности.
//            segments  Количество кривых, используемых при вычерчивании (по умолчанию 8).
MovieClip.prototype.drawCircle = function (x, y, r, segments) {
  // Переменные, содержащие конечную и контрольную точки каждой кривой.
  var endX, endY, controlX, controlY;
  // Использовать 8 сегментов, если не задано иное.
  if (segments == undefined) {
    segments = 8;
  }
  // Вычислить количество градусов в каждом сегменте
  // (360/кол. сегментов) и преобразовать его в радианы.
  var segmentAngle = (360/segments)/180 * Math.PI;
  // Вычислить расстояние до контрольной точки каждой кривой.
  var controlDist = r / Math.cos(segmentAngle/2);
  // Начать черчение окружности с оси X.
  this.moveTo(x+r, y);
  // Начертить сегменты окружности.
  for (var i = 1; i <= segments; i++) {
    // Рассчитать конечную точку этой кривой.
    endX = x + r * Math.cos(i*segmentAngle);
    endY = y - r * Math.sin(i*segmentAngle);

    // Рассчитать контрольную точку этой кривой.
    controlX = x + controlDist * Math.cos(i*segmentAngle-segmentAngle/2);
    controlY = y - controlDist * Math.sin(i*segmentAngle-segmentAngle/2);

    // Начертить эту кривую.
    this.curveTo(controlX, controlY, endX, endY);
  }
};

// Применение drawCircle():
```

```
// Окружность
this.lineStyle(1);
this.drawCircle(200, 200, 100);
// Окружность с заливкой
this.lineStyle(3, 0xFF0000);
this.beginFill(0x0000FF);
this.drawCircle(300, 250, 50);
this.endFill();
```

Рик Эвинг (Ric Ewing) выложил другие методы черчения на страницу *http://www.formequalsfunction.com/downloads/drawmethods.html*.

См. также

MovieClip.beginGradientFill( ), *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*; глава 13

MovieClip.beginGradientFill(), метод

Flash 6

начинает черчение фигуры с градиентной заливкой

`mc.beginFill(fillType, colors, alphas, ratios, matrix)`

Аргументы

fillType

Строка, задающая тип градиента, «linear» или «radial». Если задать другое значение, градиент не выводится.

colors

Массив целых чисел, задающий цвета градиента. Обычно задается в виде шестнадцатеричной тройки RGB – `0x`*RRGGBB*, где *RR*, *GG* и *BB* – это двузначные шестнадцатеричные цифры, представляющие красную, зеленую и синюю составляющие. Например, шестнадцатеричное число `0xFF0000` задает красный цвет. Подробно о тройках RGB рассказано в описании класса *Color*.

alphas

Массив целых чисел в диапазоне от 0 до 100, задающих степень непрозрачности (или, наоборот, прозрачности) фигуры в процентах: 0 соответствует полной прозрачности, а 100 – полной непрозрачности. По умолчанию принимается равным 100. Значения, большие 100, интерпретируются как 100, а меньшие 0 интерпретируются как 0. Если количество элементов в массиве *alphas* не совпадает с количеством элементов в массиве *colors*, градиент не выводится.

ratios

Массив целых чисел, задающий пространственное распределение цветов в градиенте по шкале от 0 до 255. Каждый элемент массива задает относительное положение, в котором соответствующий цвет из массива *colors* будет показан как чистый несмешанный цвет. Если количество элементов в массиве *ratios* не совпадает с количеством элементов в массиве *colors*, градиент не выводится.

matrix

Объект, задающий положение, размер, поворот и наклон градиента. Объект может быть одного из двух видов: простая квадратная рамка, в которой положение, размер, поворот и наклон градиента задаются с помощью интуитивно понятных

свойств, либо матрица преобразования размером 3×3, отображающая единичный градиент с центром в точке (0,0) в окончательные положение, размер, поворот и наклон. Подробности см. в «Описании».

Описание

Как и *beginFill( )*, метод *beginGradientFill( )* начинает черчение фигуры, но закрашивает ее линейным или радиальным градиентом (рис. C.1), а не сплошным цветом.

Рис. C.1. *Радиальный и линейный градиенты*

Определением типа градиента, положения, размера и угла поворота управляют параметры `fillType`, `colors`, `alphas`, `ratios` и `matrix`. Закрашенная фигура чертится поверх любого содержимого `mc`, созданного Drawing API, но под всем остальным содержимым `mc` (например, клипами, фигурами, текстом и кнопками, размещенными на стадии разработки, либо содержимым, прикрепленным на этапе исполнения с помощью *attachMovie( )*, *duplicateMovieClip( )* или *createTextField( )*).

Так же как и для *beginFill( )*, вычерчивание фигуры с градиентной заливкой начинается с вызова *moveTo( )*, задающего начальную точку контура фигуры. Затем мы вызываем *beginGradientFill( )*, чтобы начать черчение и определить атрибуты градиента. После этого чертим контур с помощью вызовов *lineTo( )* и/или *curveTo( )*. Градиент и фигура теоретически отделены друг от друга, как если бы они размещались на двух разных уровнях, причем градиент находился бы под фигурой. Фигура выступает в роли отверстия, через которое виден градиент (то есть градиент «усекается» фигурой). Например, если нарисовать квадрат размером 100×100 в точке (25,0) и двухцветный линейный градиент размером 75×300 в точке (0,0), то смешанная часть градиента зайдет в квадрат на 50 пикселов, а оставшаяся часть квадрата будет закрашена цветом правого фронта градиента, как показано на рис. C.2.

Для правильной градиентной заливки квадрата надо изменить размер градиента, сделав его равным 100×100, и поместить в точку (25,0), то есть в левый верхний угол квадрата. Посмотрим, как это выглядит в коде, сначала для смещенного градиента, а потом для правильно выровненного:

```
this.createEmptyMovieClip("drawing_mc", 1);
```

Теперь поместим перо в точку (25,0):

```
drawing_mc.moveTo(25, 0);
```

Теперь мы должны определить градиент. Чтобы легче было читать код, запишем все аргументы *beginGradientFill( )* в переменную. Параметр `fillType` должен иметь значение «linear» или «radial», определяя прямолинейную или круговую форму полосок цвета градиента. Мы зададим линейный градиент (в виде прямых линий):

```
var fillType = "linear";
```

Рис. C.2. *Линейный градиент и квадрат*

Затем зададим массив значений цветов (параметр *colors*). Это цветные полосы, которые будут смешиваться в нашем градиенте. У нас два цвета, белый и черный, но вы можете задать любое их количество:

```
var colors = [0xFFFFFF, 0x000000];
```

Затем зададим прозрачность каждого цвета в виде массива целых чисел от 0 до 100 (параметр *alphas*). Каждый элемент *alphas* соответствует элементу *colors* (количество элементов в обоих массивах должно быть равным). Значение 0 соответствует полной прозрачности, а 100 – полной непрозрачности. Наши цвета не прозрачны, поэтому установим оба значения равными 100:

```
var alphas = [100, 100];
```

Теперь определим пропорцию каждого цвета в нашем градиенте в виде массива целых чисел от 0 до 255 (параметр *ratios*). Массив *ratios* также должен состоять из одинакового с массивом *colors* количества элементов. Значение 0 соответствует чистому цвету, находящемуся в крайней левой (для линейного градиента) или в центральной (для радиального) точке градиента. Значение 255 соответствует чистому цвету, находящемуся в крайней правой (для линейного градиента) или в самой внешней (для радиального) точке градиента. Мы хотим, чтобы в нашем градиенте белый цвет равномерно переходил в правый по всей ширине, поэтому зададим позицию чистого белого равной 0, а позицию чистого черного равной 255:

```
var ratios = [0, 255];
```

Наконец, зададим положение, ширину, высоту и угол поворота нашего градиента в виде объекта (параметр *matrix*). Присвоим свойству `matrixType` объекта *matrix* значение «box», а затем зададим значения следующим свойствам: `x` (положение по горизонтали), `y` (положение по вертикали), `w` (ширина), `h` (высота) и `r` (угол поворота). Свойства `x` и `y` задают левый верхний угол градиента относительно точки регистрации клипа, а свойство `r` задает угол поворота градиента в радианах, а не в градусах. В нашем коде вызывается пользовательская функция *toRadians( )*, преобразующая градусы в радианы:

```
var matrix = { matrixType:"box", x:0, y:0, w:75, h:300, r:toRadians(0) };
function toRadians (deg) {
```

```
    return deg * Math.PI/180;
  }
```

Итак, аргументы градиента определены и можно вызвать *beginGradientFill( )*:

```
drawing_mc.beginGradientFill(fillType, colors, alphas, ratios, matrix);
```

Затем мы начертим квадрат так же, как при однородной заливке, обеспечив возврат в начальную точку (25,0) при последнем обращении к *lineTo( )*:

```
drawing_mc.lineTo(125, 0);
drawing_mc.lineTo(125, 100);
drawing_mc.lineTo(25, 100);
drawing_mc.lineTo(25, 0);
```

Наконец, формально закроем фигуру с помощью *endFill( )*, что завершит вычерчивание квадрата с градиентной заливкой:

```
drawing_mc.endFill();
```

Как правильно выровнять градиент с квадратом? Это нетрудно, достаточно совместить начальную точку градиента с началом фигуры (25,0) и установить размер градиента равным 100×100, изменив объект *matrix*:

```
var matrix = { matrixType:"box", x:25, y:0, w:100, h:100, r:toRadians(0) };
```

Присвоив свойству `r` объекта *matrix* значение, равное количеству радиан в 45° и полученное с помощью *toRadians(45)*, мы можем создать диагональный градиент:

```
var matrix = { matrixType:"box", x:25, y:0, w:100, h:100, r:toRadians(45) };
```

Выровненный квадрат можно залить круговым градиентом, если присвоить *fillType* соответствующее значение:

```
var fillType = "radial";
```

Чтобы изменить порядок цветов, изменим порядок элементов в массиве *colors*:

```
var colors = [0x000000, 0xFFFFFF];
```

К фигуре автоматически добавляется линия контура, стиль которой определен в последнем вызове *lineStyle( )*. Если *lineStyle( )* не вызывался, то фигура не обводится. Например, следующий код, помещенный перед командой *moveTo( )*, добавляет в наш квадрат красный контур:

```
drawing_mc.lineStyle(1, 0xFF0000);
```

В большинстве случаев объект рамки, выступавший у нас в качестве *matrix* в примере с квадратом, предоставляет большие возможности управления градиентом. Однако положение, угол поворота, масштаб и наклон можно также задать с помощью матрицы преобразования 3×3, применяемого к единичному градиенту с центром в точке (0,0). Матрицы преобразований удобны для программного генерирования заливок, что требуется, например, в 3D-генераторе. Чтобы использовать матрицу преобразования 3×3, сделаем значением параметра *matrix* метода *beginGradientFill( )* объект со свойствами `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h` и `i`, соответствующими следующим позициям в матрице 3×3:

```
---------------
|  a   b   c  |
|  d   e   f  |
|  g   h   i  |
---------------
```

Для того чтобы масштабировать, повернуть и переместить градиент, применяются следующие матрицы преобразования:

```
МАСШТАБИРОВАНИЕ              СДВИГ              ПОВОРОТ (по часовой стрелке)
---------------          ---------------        --------------------------------
|  sx  0   0  |          |  1   0   0  |        | cos(theta)  sin(theta)    0  |
|  0   sy  0  |          |  0   1   0  |        | -sin(theta) cos(theta)    0  |
|  0   0   1  |          |  tx  ty  1  |        | 0           0             1  |
---------------          ---------------        --------------------------------
sx = коэффициент по оси x     tx = величина сдвига вдоль оси x
sy = коэффициент по оси y     ty = величина сдвига вдоль оси y
```

В простейшем случае, когда не происходит ни масштабирования, ни поворота, ни сдвига градиента, матрицы имеют вид:

```
МАСШТАБИРОВАНИЕ              СДВИГ              ПОВОРОТ (по часовой стрелке)
---------------          ---------------        --------------------------------
|  1   0   0  |          |  1   0   0  |        | 1           0             0  |
|  0   1   0  |          |  0   1   0  |        | 0           1             0  |
|  0   0   1  |          |  0   0   1  |        | 0           0             1  |
---------------          ---------------        --------------------------------
sx = коэффициент по оси x     tx = величина сдвига вдоль оси x
sy = коэффициент по оси y     ty = величина сдвига вдоль оси y
```

и объект матрицы задается так:

```
var matrix = { a:1, b:0, c:0, d:0, e:1, f:0, g:0, h:0, i:1 };
```

Для того чтобы масштабировать единичный градиент до размеров 100×100, присвоим свойствам `a` и `e` значение 100, оставив `i` равным 1, а остальные – 0:

```
var matrix = { a:100, b:0, c:0, d:0, e:100, f:0, g:0, h:0, i:1 };
```

Это преобразование похоже на то, которое мы задали ранее для нашего градиента квадрата, но не совпадает с ним. Масштабирующие коэффициенты матрицы (`a` и `e`) приводят к тому, что ширина и высота единичного градиента с центром в начале координат (0,0) увеличиваются до 100. Однако при масштабировании не изменяется положение центра градиента – он остается в точке (0,0), если свойства сдвига (`g` и `h`) имеют значение 0.

Для того чтобы поместить начало градиента в точку (25,0), присвоим параметру сдвига по горизонтали (`g`) значение, передвигающее его на 25 пикселов вправо:

```
var matrix = { a:100, b:0, c:0, d:0, e:100, f:0, g:25, h:0 , i:1 };
```

В результате центр градиента будет смещен в точку (25,0).

Однако раньше наша матрица перемещала в эту точку левый верхний угол градиента, а не его центр. Чтобы совместить центр градиента с центром нашего квадрата (как это делалось раньше), надо выполнить сдвиг из точки (25,0) в центр квадрата, прибавив по 50 к горизонтальной и вертикальной координатам (50 – половина ширины и высоты квадрата).

Поэтому для центрирования нашего градиента присвоим `g` значение 75 (то есть 25 + 50) и `h` – значение 50 (то есть 0 + 50):

```
var matrix = { a:100 , b:0 , c:0 , d:0 , e:100 , f:0 , g:75 ,h:50 , i:1 };
```

Для сравнения приведем полный код, выполняющий заливку квадрата и демонстрирующий оба способа задания аргумента *`matrix`* в ***beginGradientFill( )***:

```
// Преобразование градусов в радианы.
function toRadians (deg) {
  return deg * Math.PI/180;
}

// Определить параметры beginGradientFill().
var fillType = "linear";
var colors = [0xFFFFFF, 0x000000];
var alphas = [100, 100];
var ratios = [0, 255];
// Использовать matrix в форме "box"
var matrix = { matrixType:"box", x:25, y:0, w:100, h:100, r:toRadians(0) };
// или в форме матрицы преобразования:
var matrix = { a:100 , b:0 , c:0 , d:0 , e:100 , f:0 , g:75 ,h:50 , i:1 };

// Создать клип.
this.createEmptyMovieClip("drawing_mc", 1);

// Начертить квадрат с градиентной заливкой.
drawing_mc.moveTo(25, 0);
drawing_mc.beginGradientFill(fillType, colors, alphas, ratios, matrix);
drawing_mc.lineTo(125, 0);
drawing_mc.lineTo(125, 100);
drawing_mc.lineTo(25, 100);
drawing_mc.lineTo(25, 0);
drawing_mc.endFill();
```

Объединяя матрицы, мы можем создать одну матрицу, описывающую масштабирование, вращение и сдвиг одновременно. Умножение матриц выходит за рамки этой книги. Однако на диске с Flash MX, поставлемом Macromedia, в файле *Goodies\Macromedia\Other Samples\transformmatrix.as* есть класс *TransformMatrix*, определяющий удобные методы для задания и объединения матриц преобразований.

Применение

Flash пытается исправить неправильно заданные фигуры (например, фигуры без метода *endFill( )*), но надежные результаты можно получить только при соблюдении правил черчения, отмеченных в описании *MovieClip.beginFill( )*.

Пример

Следующий код вычерчивает треугольник с радиальным градиентом из красного, зеленого и синего цветов. Центр треугольника располагается в точке регистрации клипа, а центр градиента совмещен с центром треугольника:

```
// Создать пустой клип для черчения и установить перо
this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc._x = drawing_mc._y = 200;
drawing_mc.lineStyle(1, 0xFF0000);
drawing_mc.moveTo(0, -50);

// Задать параметры градиента
var fillType = "radial";
var colors = [0xFF0000, 0x00FF00, 0x0000FF];
var alphas = [100, 100, 100];
var ratios = [0, 127, 255];
var matrix = { matrixType:"box", x:-50, y:-35, w:100, h:100, r:toRadians(0) };
function toRadians (deg) {
  return deg * Math.PI/180;
}
```

```
// Начать черчение фигуры
drawing_mc.beginGradientFill(fillType, colors, alphas, ratios, matrix);
// Начертить стороны треугольника
drawing_mc.lineTo(50, 50);
drawing_mc.lineTo(-50, 50);
drawing_mc.lineTo(0, -50);
// Закончить черчение
drawing_mc.endFill();
```

См. также

MovieClip.beginFill( ), *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*; глава 13

MovieClip.clear(), метод

Flash 6

удаляет из клипа все содержимое, созданное с помощью Drawing API

mc.clear()

Описание

Метод *clear( )* удаляет все линии (контуры) и фигуры (заливку), созданные в *mc* с помощью методов *MovieClip* Drawing API, не трогая содержимое, созданное в среде разработки Flash. Он также сбрасывает текущий стиль линии (как при вызове *lineStyle(undefined)*) и возвращает перо в положение (0,0). Чтобы включить отображение линий в последующих чертежах, вызовите *lineStyle( )* перед какими-либо другими методами черчения.

С помощью Drawing API нельзя удалить содержимое, созданный в среде разработки. Используйте свойство _visible, чтобы сделать клип невидимым.

Пример

Следующий код вычерчивает в каждом кадре прямую линию, стиль которой выбирается случайным образом. Метод *clear( )* стирает линию, вычерченную в предыдущем кадре:

```
// Пользовательская функция, которая возвращает случайное целое число
Math.randomInt = function (minVal, maxVal) {
  r = Math.random();
  return minVal + Math.floor(r * (maxVal + 1 - minVal));
}

this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc.onEnterFrame = function () {
  this.clear();
  this.lineStyle(Math.randomInt(1, 40), Math.randomInt(1, 0xFFFFFF));
  this.moveTo(Math.randomInt(1, 550),Math.randomInt(1, 400));
  this.lineTo(Math.randomInt(1, 550),Math.randomInt(1, 400));
}
```

Упражнение 1: Попробуйте убрать вызов *clear( )* и посмотреть, как накапливаются вызовы функций черчения, пока не будет выполнен метод *clear( )*.

Упражнение 2: Метод *clear( )* удаляет содержимое Drawing API только из заданного клипа. Напишите пользовательскую функцию, удаляющую все содержимое Drawing API из всех клипов, начиная с заданного и рекурсивно стирая динамическое содержимое из всех вложенных клипов. Поиск вложенных клипов организуйте при помо-

щи цикла *for-in*. Чтобы удалить все динамическое содержимое, можно применить эту функцию к главной временной диаграмме.

См. также

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*, `MovieClip._visible`; глава 13

MovieClip.createEmptyMovieClip(), метод

Flash 6

создает новый пустой экземпляр MovieClip

`mc.createEmptyMovieClip(instanceName, depth)`

Аргументы

instanceName

Строка с именем экземпляра для создаваемого клипа. Имя должно удовлетворять правилам создания идентификаторов, описанным в разделе «Идентификаторы» главы 15. Для того чтобы активизировать подсказчик кода в среде разработки Flash, включите в имя экземпляра суффикс `"_mc"`, как в `ball_mc`.

depth

Целое число от –16 384 до 1 048 575 включительно, задающее уровень, на который должен быть помещен новый клип в `mc`. Клипы с большей глубиной закрывают клипы с меньшей глубиной, поэтому клип с глубиной 6 находится перед клипом с глубиной 5, расположенным перед клипом с глубиной 4, и т. д. Глубины, превосходящие по модулю –1, зарезервированы для динамически создаваемого содержимого, поэтому, как правило, в *createEmptyMovieClip( )* указывается значение глубины от 0 до 1 048 575. Однако для создания динамического содержимого, которое должно оказаться в `mc` ниже любого содержимого среды разработки (например, для фона этапа исполнения), следует задавать глубину –16 384. Глубины от –16 383 до –1 включительно зарезервированы для содержимого, создаваемого на этапе разработки, и не должны использоваться в *createEmptyMovieClip( )*. Важную дополнительную информацию можно найти в разделе «Добавление в стек клипов, созданных с помощью attachMovie()» в главе 13.

Возвращаемое значение

Ссылка на новый экземпляр *MovieClip*.

Описание

Метод *createEmptyMovieClip( )* создает новый, пустой экземпляр клипа и прикрепляет его к `mc` на заданной глубине `depth`. Если `mc` представляет собою главный фильм, то новый экземпляр помещается в верхний левый угол сцены. Если же `mc` – это экземпляр клипа, то новый экземпляр помещается в точку регистрации `mc`. Новый экземпляр располагается в стеке содержимого `mc` на уровне, заданном `depth`, удаляя прежнее его содержимое, если оно там есть.

Метод *createEmptyMovieClip( )* применяется для создания содержимого на этапе исполнения без обращения к каким-либо символам библиотеки. Например, чтобы динамически создавать космический корабль, можно в пустом клипе нарисовать корабль с помощью методов *MovieClip* Drawing API. Аналогично, чтобы создать на этапе исполнения форму или панель интерфейса, можно поместить в пустой клип текстовые поля, например:

```
this.createEmptyMovieClip("form_mc", 1);
form_mc.createTextField("firstName_txt", 1, 0, 0, 200, 20);
form_mc.firstName_txt.border = true;
form_mc.firstName_txt.type = "input";
form_mc.createTextField("lastName_txt", 2, 0, 30, 200, 20);
form_mc.lastName_txt.border = true;
form_mc.lastName_txt.type = "input";
form_mc.createTextField("email_txt", 3, 0, 60, 200, 20);
form_mc.email_txt.border = true;
form_mc.email_txt.input = "input";
```

В пустом клипе можно также просто хранить события, такие как *onEnterFrame( )*, *onKeyDown( )*, *onMouseDown( )* и т. д.:

```
this.createEmptyMovieClip("process_mc", 1);
process_mc.onEnterFrame = function () {
  // задачи, выполняемые в каждом кадре
}
```

Метод *createEmptyMovieClip( )* возвращает ссылку на созданный им клип. Для удобства доступа можно сохранить ее в переменной. Например, следующий код создает пустой клип, вложенный глубоко в интерфейс приложения, но сохраняет ссылку на клип в переменной `fileMenu`:

```
// Создать клип.
var fileMenu = appPanels.editPanel.menuBar.createEmptyMovieClip
       ("fileMenu_mc", 1);
// Обрабатываем клип через переменную fileMenu.
fileMenu._x = 20;
fileMenu._y = 50;
```

Клип, созданный с помощью *createEmptyMovieClip( )*, может быть использован в качестве маски для других клипов (см. *MovieClip.setMask( )*) и может получать события кнопок, такие как *onPress( )* и *onRelease( )*. Однако контуры и прозрачные заливки не воспринимаются как маски или активные области для событий кнопок.

Удалить клип можно с помощью *MovieClip.removeMovieClip( )*. Однако *removeMovieClip( )* не удаляет клип, помещенный на отрицательную глубину. Чтобы удалить клип с отрицательной глубины, сначала перебросьте его на положительную глубину, а потом удалите:

```
theClip_mc.swapDepths(1234);   // Выбрать положительную глубину для обмена клипов
theClip_mc.removeMovieClip();  // Удалить клип
```

Важную дополнительную информацию можно найти в описании глобальной функции *removeMovieClip( )*.

Применение

Глубины, превосходящие 1 048 575, частично функционируют, но не поддерживаются. Например, можно поместить клип на глубину 1 048 576, но нельзя удалить его оттуда. Попытка прикрепить клип на глубине ниже –16 384 окажется безуспешной.

См. также

duplicateMovieClip( ), *MovieClip.attachMovie( )*, *MovieClip.createTextField( )*, *MovieClip.duplicateMovieClip( )*, *MovieClip.getDepth( )*, *MovieClip.removeMovieClip( )*, *MovieClip. swapDepths( )*

MovieClip.createTextField(), метод

Flash 6

создает новый объект TextField

`mc.createTextField(instanceName, depth, x, y, width, height)`

Аргументы

instanceName

Строка с именем экземпляра для создаваемого объекта *TextField*. Имя должно удовлетворять правилам создания идентификаторов, описанным в разделе «Идентификаторы» главы 15. Для того чтобы активизировать подсказчик кода в среде разработки Flash, включите в имя экземпляра суффикс "`_txt`", как в `firstName_txt`.

depth

Целое число от –16 384 до 1 048 575 включительно, задающее уровень, на который должно быть помещено новое текстовое поле в *mc*. Клипы с большей глубиной закрывают клипы с меньшей глубиной, поэтому клип с глубиной 6 расположен перед клипом с глубиной 5, а он, в свою очередь, – перед клипом с глубиной 4, и т. д. Глубины, превосходящие по модулю –1, зарезервированы для динамически создаваемого содержимого, поэтому, как правило, в *createTextField( )* указывается значение глубины от 0 до 1 048 575. Однако для создания динамического содержимого, которое должно оказаться в `mc` ниже любого содержимого среды разработки (например, для фона этапа исполнения), следует задавать глубину –16 384. Глубины от –16 383 до –1 включительно зарезервированы для содержимого, создаваемого на этапе разработки, и не должны использоваться в *createTextField( )*. См. также раздел «Порядок размещения фильмов и экземпляров в стеке» главы 13.

x Число с плавающей точкой, задающее (в пикселах) горизонтальную позицию левого края создаваемого текстового поля относительно пространства координат *mc*. См. *TextField.*`_x`.

y Число с плавающей точкой, задающее (в пикселах) вертикальную позицию верхнего края создаваемого текстового поля относительно пространства координат *mc*. См. *TextField.*`_y`.

width

Число с плавающей точкой, задающее (в пикселах) ширину рамки создаваемого текстового поля. См. *TextField.*`_width`.

height

Число с плавающей точкой, задающее (в пикселах) высоту рамки создаваемого текстового поля. См. *TextField.*`_height`.

Описание

Метод *createTextField( )* создает на экране объект *TextField* нового текстового поля с именем *instanceName*, управляемый посредством ActionScript. Это эквивалент этапа исполнения для создания текстового поля инструментом Text в среде разработки Flash. Если *mc* представляет собою главный фильм, то новое текстовое поле помещается в точку (*x*, *y*) относительно левого верхнего угла сцены. Если же *mc* – это экземпляр клипа, то новое текстовое поле помещается в точку (*x*, *y*) относительно точки регистрации *mc*. Новое текстовое поле располагается в стеке содержимого *mc* на уровне, заданном *depth*, удаляя прежнее его содержимое. По умолчанию свойство `type` текстового поля получает значение «dynamic».

Созданным текстовым полем можно управлять посредством методов и свойств класса *TextField*. Инструкции по удалению текстового поля приведены в описании метода *TextField.removeTextField( )*.

Пример

Следующий код создает текстовое поле на главной временной диаграмме фильма (_root), на глубине 2, в точке (0, 0) (левый верхний угол фильма), с рамкой шириной 200 и высотой 20. После создания поля в него выводится текст «Hello world!»:

```
_root.createTextField("welcome_txt", 1, 0, 0, 200, 20);
_root.welcome_txt.text = "Hello world!";
```

Следующий код создает такое же текстовое поле, но, кроме того, заключает его в рамку и помещает в центре сцены фильма:

```
// Создать текстовое поле.
_root.createTextField("welcome_txt", 1, 0, 0, 200, 20);
// Добавить рамку и текст.
_root.welcome_txt.border = true;
_root.welcome_txt.text = "Welcome to my site!";
// Привести рамку в соответствие с размером текста.
_root.welcome_txt.autoSize = true;

// Задать режим выравнивания в фильме.
Stage.align = "LT";
Stage.scaleMode = "noScale";

// Перемещать текстовое поле в центр при каждом изменении размеров фильма.
_root.welcome_txt.onResize = function () {
  this._x = (Stage.width/2) - (this._width/2);
  this._y = (Stage.height/2) - (this._height/2);
}
// Обеспечить прием текстовым полем события Stage.onResize().
Stage.addListener(_root.welcome_txt);

// Переместить текстовое поле в центр вручную.
// Это требуется для автономного проигрывателя,
// который не генерирует автоматически onResize() при загрузке фильма.
_root.welcome_txt.onResize();
```

См. также

MovieClip.createEmptyMovieClip( ), *TextField.getDepth( )*, *TextField.removeTextField( )*; класс *Stage*, класс *TextField*

MovieClip._currentframe, свойство

Flash 4

номер кадра, в котором находится головка воспроизведения клипа или фильма — **только для чтения**

mc._currentframe

Описание

Целочисленное свойство _currentframe представляет номер кадра, в котором в данное время находится головка воспроизведения *mc*. Обратите внимание, что первый кадр имеет номер 1, а не 0, поэтому _currentframe находится в диапазоне от 1 до *mc*._totalframes.

Пример

```
// Переместить головку воспроизведения на 2 кадра назад от текущего положения
this.gotoAndStop(_currentframe - 2);
```

См. также

MovieClip.gotoAndPlay( ), *MovieClip.gotoAndStop( )*, *MovieClip*._totalframes

MovieClip.curveTo(), метод

Flash 6

вычерчивает кривую

mc.curveTo(*controlX*, *controlY*, *anchorX*, *anchorY*)

Аргументы

controlX

Число с плавающей точкой – горизонтальная координата контрольной точки квадратичной кривой Безье. Измеряется относительно точки регистрации *mc*.

controlY

Число с плавающей точкой – вертикальная координата контрольной точки квадратичной кривой Безье. Измеряется относительно точки регистрации *mc*.

anchorX

Число с плавающей точкой – горизонтальная координата конечной точки вычерчиваемой кривой. Измеряется относительно точки регистрации *mc*.

anchorY

Число с плавающей точкой – вертикальная координата конечной точки вычерчиваемой кривой. Измеряется относительно точки регистрации *mc*.

Описание

Метод *curveTo( )* вычерчивает квадратичную кривую Безье из текущей позиции пера в точку (*anchorX*, *anchorY*), основываясь на внешней контрольной точке (*controlX*, *controlY*). Через эту контрольную точку проходят касательные к кривой в ее начальной и конечной точках. Можно сказать, что прямая, которая соединила бы начальную позицию пера с конечной точкой (*anchorX*, *anchorY*), оттягивается контрольной точкой (*controlX*, *controlY*), в результате чего образуется кривая. Альтернативный способ, в соответствии с которым кривая проводится через три конкретные точки, не требующий задания пользователем контрольной точки, приводится ниже в примере.

Параметры штриха кривой (толщина, цвет, непрозрачность) определяются последним обращением к *lineStyle( )*.

Если обращений к *lineStyle( )* не было, стиль штриха имеет значение undefined. В этом случае *curveTo( )* перемещет перо в точку (*anchorX*, *anchorY*), как это сделал бы метод *moveTo( )*, но не вычерчивает кривую на экране.

Начерченная кривая появляется над всем содержимым, уже созданным средствами Drawing API в *mc*, но ниже всего остального содержимого в *mc* (например, клипов, фигур, текста и кнопок, размещенных на этапе разработки, или содержимого, прикрепленного на этапе выполнения с помощью *attachMovie( )*, *duplicateMovieClip( )* или *createTextField( )*).

Рис. С.3. *Пример квадратичной кривой Безье*

Следующий код создает клип с именем `drawing_mc` и вычерчивает черную кривую толщиной 4 пиксела из начального положения пера по умолчанию (0,0) в конечную точку (100,0), при этом контрольная точка имеет координаты (500, –100). Полученная кривая показана на рис. С.3.

```
// Создать клип для черчения
this.createEmptyMovieClip("drawing_mc", 1);

// Начертить черную линию толщиной 4 пиксела
drawing_mc.lineStyle(4);

// Начертить кривую
drawing_mc.curveTo(50, -100, 100, 0);

// Расположить клип в центре сцены
drawing_mc._x = 200;
drawing_mc._y = 225;
```

По завершении вычерчивания кривой перо остается в ее конечной точке. Для того чтобы чертить сложные кривые или замкнутые фигуры, например окружности и многоугольники, можно вызывать *curveTo( )* и/или *lineTo( )* многократно. Подробности см. в примере 2 для *MovieClip.beginFill( )*.

Если какой-либо из аргументов *curveTo( )* отсутствует, то операция завершается без сообщений об ошибках, и положение пера не меняется. О том, что представляет собой перо, рассказано в описании *MovieClip.moveTo( )*.

Учебные материалы по математическим основам и программированию кривых Безье см. по адресу:

http://www.ipm.sci-nnov.ru/~demidov/VRML/Splines/Intro/Bezier.htm

Пример

Иногда удобнее задать три точки на кривой вместо двух конечных точек и контрольной точки. В следующем коде во все клипы добавляется пользовательский метод *curveThrough3Pts( )*. Этот метод принимает три точки в качестве аргументов и вычерчивает квадратичную кривую, проходящую через них. При этом считается, что вторая точка расположена на середине кривой (по времени – t=0,5):

```
// Взято из метода drawCurve3Pts() Роберта Пеннера (Robert Penner)
MovieClip.prototype.curveThrough3Pts =
```

```
    function (startX, startY, throughX, throughY, endX, endY) {
  var controlX = (2 * throughX) - .5 * (startX + endX);
  var controlY = (2 * throughY) - .5 * (startY + endY);
  this.moveTo(startX, startY);
  this.curveTo(controlX, controlY, endX, endY);
};

// Применение
this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc.lineStyle(1, 0xFF0000);
drawing_mc.curveThrough3Pts(100, 100, 150, 50, 200, 100);
```

См. также

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*

MovieClip._droptarget, свойство

Flash 4

путь к клипу или фильму, на который опущен перетаскиваемый клип **только для чтения**

mc.`_droptarget`

Описание

Во время перетаскивания *mc* `_droptarget` хранит строку, указывающую путь к клипу, над которым находится *mc* (если такой есть). Если *mc* не находится над клипом, `_droptarget` возвращает значение `undefined`. Если *mc* ранее перетаскивался и был опущен на клип, `_droptarget` хранит строку, указывающую путь к клипу, на который был опущен *mc*. Путь указывается в слэш-нотации. Клип считается находящимся «над» другим клипом, если точка регистрации перетаскиваемого клипа перекрывается с какой-либо частью целевого клипа. Если *mc* находится над двумя или более перекрывающимися клипами, то в качестве `_droptarget` берется самый верхний из них. Определить пересечение для любого количества перекрывающихся клипов можно с помощью *MovieClip.hitTest( )*.

Пример

Свойство `_droptarget` удобно при создании интерфейсов с буксировкой. В следующем примере показано, как создать простой интерфейс корзины покупок с перетаскиванием с помощью `_droptarget`. Когда клип `item` опускается на клип `basket`, клипу `item` разрешается остаться в `basket`, в противном случае `item` возвращается в исходное положение):

```
// Задать начальное положение
item_mc.origX = _x;
item_mc.origY = _y;

// Определить функцию, вызываемую при щелчке по товару
item_mc.drag = function () {
  this.startDrag();
}

// Определить функцию, вызываемую при опускании товара в корзину
item_mc.drop = function () {
  stopDrag();
  if (this._droptarget != "/basket") {
```

```
        this._x = this.origX;
        this._y = this.origY;
      }
    }

    // Присвоить функцию обратного вызова соответствующим обработчикам событий кнопок item_mc
    item_mc.onPress = item_mc.drag;
    item_mc.onRelease = item_mc.onReleaseOutside = item_mc.drop;
```

Обратите внимание, что в `_droptarget` хранится строка в слэш-нотации, а не ссылка на клип. Для преобразования строки `_droptarget` в ссылку на клип применяется способ, показанный в примере для глобальной функции *eval( )*.

См. также *MovieClip.startDrag( )*, *MovieClip.stopDrag( )*

MovieClip.duplicateMovieClip(), метод

Flash 5; во Flash 6 добавлены initObject и возвращаемое значение

создает копию клипа

mc.duplicateMovieClip(*newName*, *depth*)
mc.duplicateMovieClip(*newName*, *depth*, *initObject*)

Аргументы

newName

Строка, которая станет именем экземпляра создаваемой копии клипа. Она должна соответствовать правилам создания идентификаторов, приведенным в разделе «Идентификаторы» главы 15.

depth

Целое число от –16 384 до 1 048 575 включительно, задающее уровень, на который должен быть помещен новый клип в родительском объекте *mc*. Клипы с большей глубиной закрывают клипы с меньшей глубиной, поэтому клип с глубиной 6 расположен перед клипом с глубиной 5, а он, в свою очередь, перед клипом с глубиной 4, и т. д. Глубины, превосходящие по модулю –1, зарезервированы для динамически создаваемого содержимого, поэтому, как правило, в *duplicateMovieClip( )* указывается значение глубины от 0 до 1 048 575. Однако для создания динамического содержимого, которое должно оказаться в *mc* ниже любого содержимого среды разработки (например, для фона этапа исполнения), следует задавать глубину –16 384. Глубины от –16 383 до –1 включительно зарезервированы для содержимого, создаваемого на этапе разработки, и не должны использоваться в *duplicateMovieClip( )*. См. раздел «Порядок размещения фильмов и экземпляров в стеке» главы 13.

initObject

Необязательный объект, свойства которого будут скопированы в новый экземпляр клипа. Не поддерживается во Flash 5.

Возвращаемое значение

Во Flash 6 *duplicateMovieClip( )* возвращает объектную ссылку на прикрепленный клип (во Flash 5 всегда возвращалось значение `undefined`).

Описание

Метод *MovieClip.duplicateMovieClip( )* представляет собой альтернативу глобальной функции *duplicateMovieClip( )*. При вызове в качестве метода *MovieClip duplicateMovie-*

Clip( ) не принимает параметр *target*, а дублирует *mc* и возвращает ссылку на новую копию. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем у соответствующей глобальной функции, поскольку не требуется параметр *target*.

Общие рекомендации по применению приведены в описании глобальной функции *duplicateMovieClip( )*. Параметр *initObject* подробно описан в статье, посвященной *MovieClip.attachMovie( )*, и в главе 14.

Применение

В виде глобальной функции *duplicateMovieClip( )* не поддерживает параметр *initObject* и не возвращает ссылку на созданную копию клипа (эти ограничения относятся и формату метода клипа во Flash 5).

Пример

Создать дубликат ball1_mc под именем ball2_mc. Дубликат помещается на глубину 1 в родительском объекте ball1_mc:

```
ball1_mc.duplicateMovieClip("ball2_mc", 1);
```

См. также *duplicateMovieClip( )*, *MovieClip.attachMovie( )*

MovieClip.enabled, свойство

Flash 6

булево значение; разрешает или запрещает взаимодействие с кнопкой **чтение/запись**

mc.enabled

Описание

Булево свойство *MovieClip*.enabled, подобно *Button*.enabled, определяет, включен ли для *mc* режим кнопки (true) или выключен (false). По умолчанию *MovieClip*.enabled имеет значение true. Однако оно действует только в том случае, когда для *mc* определено одно или более событий кнопки (*onPress( )*, *onRelease( )* и т. д.). Если enabled имеет значение false, *mc* не получает уведомлений о событиях кнопки, курсор «ладонь» не показывается и головка воспроизведения *mc* в ответ на события мыши не перескакивает в кадры клипа с метками _up, _over и _down. Однако воспроизведение *mc* происходит нормально, и на него можно установить фокус ввода с помощью клавиши Tab. Для того чтобы кнопка не получала фокус ввода при табуляции, надо установить свойство mc.tabEnabled равным false.

Сведения о реализации поведения кнопки для клипа см. в описании класса *MovieClip*, метода *MovieClip*.hitArea и в разделе «Применение клипов в качестве кнопок» главы 13.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

См. также

Button.enabled, *MovieClip*.hitArea, *MovieClip*.tabEnabled, *MovieClip*.useHandCursor; «Применение клипов в качестве кнопок» в главе 13

MovieClip.endFill(), метод

Flash 6

завершает заливку, начатую beginFill() или beginGradientFill()

mc.endFill()

Описание

Метод *endFill( )* завершает заливку фигуры, начатую предшествующим обращением к *beginFill( )* или *beginGradientFill( )*, вызывая появление фигуры на экране. Если текущее положение пера не совпадает с его положением при вызове *beginFill( )* или *beginGradientFill( )*, Flash замыкает фигуру прямой линией и применяет заливку. Однако создание незамкнутых фигур может привести к непредсказуемым результатам и прибегать к нему не рекомендуется.

Подробно о вычерчивании фигур см. *MovieClip.beginFill( )* и *MovieClip.beginGradientFill( )*.

Пример

```
// Начертить четырехугольник на главной временной диаграмме
_root.moveTo(225, 200);
_root.beginFill(0x00FF00, 50);
_root.lineTo(100,100);
_root.lineTo(0,100);
_root.lineTo(0,300);
_root.lineTo(225,200);
_root.endFill();
```

См. также

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*

MovieClip.focusEnabled, свойство

Flash 6

разрешает клипу получать фокус программно

mc.focusEnabled

Описание

Если булево свойство focusEnabled имеет значение false или undefined (по умолчанию) и в *mc* не определены обработчики событий кнопок (*onPress( )*, *onRelease( )* и т. д.), то *mc* нельзя поместить в фокус с помощью *Selection.setFocus( )*. Если focusEnabled имеет значение true, то *mc* можно поместить в фокус с помощью *Selection.setFocus( )* в любой момент, даже если не определены обработчики событий кнопок. Если в *mc* определен хотя бы один обработчик события кнопки, его все же можно поместить в фокус с помощью *Selection.setFocus( )*, даже если focusEnabled имеет значение false. Чтобы отключить возможность помещения в фокус для клипа с обработчиками событий кнопки, удалите обработчики и установите focusEnabled равным false. Например, следующий код удаляет обработчик *onPress( )* из theClip_mc:

```
delete theClip_mc.onPress;
```

Свойство focusEnabled используется при создании клипов, выступающих в роли управляемых с клавиатуры элементов интерфейса, таких как модальные диалоговые окна с кнопками Yes и No, активируемыми клавишами «Y» и «N». Диалоговое окно должно реагировать на «Y» и «N», только если оно находится в фокусе. Ниже приво-

дится упрощенная функция, создающая клип диалогового окна и помещающая его в фокус:

```
function displayAlert () {
  // appInterface - клип, содержащий главный интерфейс приложения.
  // getFreeDepth(), не показана; функция, возвращающая не занятую глубину,
  // на которую помещается клип dialog. dialog - клип окна диалога.
  var dialog = appInterface.attachMovie("dialog",
                                        "alertDialog",
                                        appInterface.getFreeDepth());
  // Разрешить помещение окна диалога в фокус.
  dialog.focusEnabled = true;
  // Поместить окно диалога в фокус.
  Selection.setFocus(dialog);
}
```

Чтобы выделять клип dialog при получении им фокуса, можно создать на временной диаграмме клипа dialog кадры со специальными метками _over и _up. Кадры _up и _over будут автоматически отображаться Flash соответственно при потере и получении клипом фокуса. Может также потребоваться задать свойство клипа _focusrect равным false. Специальные вопросы, относящиеся к фокусу для клипов, управляемых с клавиатуры, рассмотрены в описании *MovieClip.onKeyDown( )*.

Обратите внимание, что focusEnabled и tabEnabled — не синонимы. Даже если focusEnabled имеет значение false, клип можно поместить в фокус с помощью клавиши Tab, если tabEnabled равно true. Однако присваивание focusEnabled значения true не включает установку фокуса на клип клавишей Tab. Установка фокуса клавишей Tab независимо управляется свойствами tabEnabled и tabIndex.

См. также

MovieClip._focusrect, *MovieClip*.tabEnabled, *MovieClip*.tabIndex, *TextField*.selectable, ***Selection. setFocus( )***

MovieClip._focusrect, свойство

Flash 6

отключает автоматический вывод желтой рамки вокруг клипа, находящегося в фокусе — **чтение/запись**

mc._focusrect

Описание

Когда клип находится в фокусе ввода с клавиатуры, Flash заключает его в желтую прямоугольную рамку, которая может выглядеть непривлекательно. Отключить показ рамки для отдельного клипа можно, задав для его свойства _focusrect значение false:

```
listbox_mc._focusrect = false;
```

Если _focusrect имеет значение true и *mc* находится в фокусе ввода с клавиатуры, Flash выводит желтый прямоугольник и содержимое кадра *mc* _over (если существует кадр с меткой _over). Если _focusrect имеет значение false, то Flash выводит только кадр _over. Если _focusrect имеет значение null (по умолчанию), используется значение глобального свойства _focusrect. Чтобы отключить показ желтого прямоугольника фокуса для всех клипов и кнопок фильма, присвойте глобальному свойству _focusrect значение false. Обратите внимание, что значение глобального свойства _focusrect не отражается в свойствах _focusrect отдельных экземпляров клипов.

Однако присваивание значения свойству _focusrect отдельного клипа преобладает над глобальной установкой.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

См. также *Button*._focusrect, _focusrect

MovieClip._framesloaded, свойство

Flash 4

количество кадров клипа или фильма, загруженных в проигрыватель **только для чтения**

mc._framesloaded

Описание

Целочисленное свойство _framesloaded указывает количество кадров *mc*, загруженных во Flash Player. Обычно применяется при создании предзагрузчиков, приостанавливающих воспроизведение до загрузки достаточного количества кадров. Для клипа свойство _framesloaded всегда равно _totalframes (потому что клипы целиком загружаются перед воспроизведением), кроме случая, когда экземпляр находится в процессе загрузки внешнего файла *.swf* в результате вызова *loadMovie( )*. Поэтому свойство _framesloaded полезно только для главных фильмов или внешних файлов *.swf*, загружаемых в экземпляры или уровни. Для контроля процесса загрузки в байтах, а не в кадрах, следует применять методы *getBytesLoaded( )* и *getBytesTotal( )*.

Код предзагрузчика обычно помещается непосредственно на главную временную диаграмму предварительно загружаемого фильма. Одно из простых решений состоит в том, чтобы в цикле показывать кадры 1 и 2, пока не будет загружен фильм, после чего перейти в начальный кадр фильма. Например:

```
//КОД В КАДРЕ 1
if (_framesloaded > 0 && _framesloaded == _totalframes) {
  gotoAndPlay("beginMovie");
}

// КОД В КАДРЕ 2
gotoAndPlay(1);
```

Во Flash 6 и выше можно, используя обработчик события клипа *onEnterFrame( )*, построить предзагрузчик, отличающийся большей переносимостью. В том кадре фильма, где начинается предварительная загрузка, вызывается функция *stop( )*. Затем на временную диаграмму этого фильма помещается клип со следующим кодом обработчика *onEnterFrame( )*:

```
stop();
this.createEmptyMovieClip("loader", 1);
loader.onEnterFrame = function () {
  this.loaded = this._parent._framesloaded;
  if (this.loaded > 0 && this.loaded == this._parent._totalframes) {
    this._parent.gotoAndPlay("beginMovie");
    this.removeMovieClip();
  }
}
```

В этом примере клип следит за ходом процесса загрузки своего родителя и запускает воспроизведение родителя по окончании загрузки. Благодаря использованию клипа в качестве предзагрузчика удается обойтись без цикла на временной диаграмме фильма с предварительной загрузкой. Предзагрузчик клипа можно даже превратить в компонент, что облегчило бы жизнь менее опытным разработчикам.

Обратите внимание, что в наших примерах предзагрузчиков помимо проверки `_framesloaded == _totalframes` проверялось также условие `_framesloaded > 0`. Это необходимо, поскольку при выгрузке фильма из клипа у этого клипа `_totalframes` становится равным 0. Поэтому если `_framesloaded` равно 0 (что может иметь место при очень низкоскоростном соединении), то сравнение `_framesloaded == _totalframes` может возвратить `true`, даже если ни один кадр еще не загружен. Наша проверка не дает фильму проскочить вперед, пока не будет загружено соответствующее содержимое. В такой предосторожности нет необходимости при загрузке предзагрузчиков, помещенных на главную временную диаграмму фильма для загрузки файлов *.swf*, поскольку их свойство `_totalframes` никогда не равно нулю.

Функции *loadMovie( )* и *unloadMovie( )* выполняются асинхронно, поэтому значение `_framesloaded` будет различным в зависимости от того, когда происходит вызов, и от значения *mc*:

- Если *mc* имеет значение `undefined`, то `_framesloaded` возвращает `undefined`.
- После обнаружения в исходном коде команд *loadMovie( )* или *unloadMovie( )*, но до того как они будут выполнены или будет загружен какой-либо внешний файл, `_framesloaded` возвращает количество кадров в прежнем *mc* (существовавшем до команд *loadMovie( )* или *unloadMovie( )*).
- После извлечения части нового файла *.swf* `_framesloaded` возвращает количество кадров, загруженных к данному времени, от 0 до `_totalframes`, если только файл на *url* найден.
- Если файл на *url* не найден и *mc* представляет собой клип, `_framesloaded` возвращает –1, а *mc*.`_totalframes` хранит 0. Однако если *mc* – это документ `_level`, то `_framesloaded` и `_totalframes` возвращают `undefined`.
- После выполнения команды *unloadMovie( )* `_framesloaded` возвращает 0, если *mc* – это клип, и возвращает `undefined`, если *mc* представляет собой уровень документа, например `_level0`.

Пример

В предзагрузчиках часто имеется горизонтальный индикатор загрузки и текстовое поле, в котором выводится процент загрузки фильма. Индикаторы загрузки реализуются в виде клипов, размер которых задается свойством `_width` или свойством `_xscale`. Однако учтите, что клип масштабируется относительно своей точки регистрации (пропорционально в левую и правую стороны от точки регистрации клипа). Поэтому для изменения размера клипа только в одну сторону надо поместить все содержимое клипа по одну сторону от точки регистрации в символе клипа. В следующем примере показано, как добавить индикатор загрузки и тестовое поле состояния в прежний код обработчика клипа. Индикатор загрузки рисуется с помощью *MovieClip* Drawing API, но код загрузки должен работать и с клипами, нарисованными вручную.

```
// Переносимый предзагрузчик с индикатором состояния
// Остановить загружающий фильм
stop();
// Создать клип-контейнер для предварительной загрузки
this.createEmptyMovieClip("loader", 1);
```

```
// Создать клипы индикатора загрузки
this.loader.createEmptyMovieClip("totalBar", 2);
this.loader.createEmptyMovieClip("loadedBar", 3);
// Создать текстовое поле для вывода
this.loader.createTextField("loadProgress_txt", 4, 80, 10, 200, 20);
// В клипах индикатора загрузки начертить красную линию поверх более жирной черной линии
this.loader.loadedBar.lineStyle(5, 0xFF0000);
this.loader.totalBar.lineStyle(8, 0x000000);
this.loader.loadedBar.lineTo(250, 0);
this.loader.totalBar.lineTo(250, 0);
// Назначить обработчик onEnterFrame(), который будет выполнять
// предварительную загрузку
this.loader.onEnterFrame = function () {
  // Определить процент загруженных байтов
  this.percentDone = Math.floor((this._parent.getBytesLoaded()
                        / this._parent.getBytesTotal()) * 100);
  // Вывести процент загруженных байтов в текстовом поле
  this.loadProgress_txt.text =  this.percentDone + "% complete";
  // Задать размер клипа loadedBar
  this.loadedBar._xscale = this.percentDone;
  // Узнать, сколько кадров загружено
  this.loaded = this._parent._framesloaded;
  // Если загрузка завершилась...
  if (this.loaded > 0 && this.loaded == this._parent._totalframes) {
    // ...запустить фильм и...
    this._parent.gotoAndPlay("beginMovie");
    // ...удалить загрузчик.
    this.removeMovieClip();
  }
}
```

Для моделирования загрузки фильма с целью тестирования предзагрузчика применяется команда View → Bandwidth Profiler в режиме Test Movie.

См. также

MovieClip.getBytesLoaded(), *MovieClip.onData()*, *MovieClip*._totalframes; «The Bandwidth Profiler» в технической заметке на *http://www.moock.org/asdg/technotes*

MovieClip.getBounds(), метод

Flash 5

определияет ограничивающий прямоугольник клипа или фильма

mc.getBounds(*targetCoordinateSpace*)

Аргументы

targetCoordinateSpace

Строка, указывающая путь к фильму или клипу, в координатном пространстве которого измеряются размеры *mc*. Поскольку ссылка на клип в строковом контексте преобразуется в путь, *targetCoordinateSpace* может быть и ссылкой на объект клипа, например *mc.getBounds*(_root) вместо *mc.getBounds*("_root"). Если не задана, то по умолчанию принимается *mc*.

Возвращаемое значение

Объект, свойства которого – xMin, xMax, yMin, yMax – описывают прямоугольник, ограничивающий пространство, занимаемое *mc*. Эти четыре свойства объекта соответству-

ют координатам самого левого, самого правого, самого верхнего и самого нижнего пикселов *mc*.

Описание

Метод *getBounds( )* возвращает анонимный объект, свойства которого определяют прямоугольную область, занимаемую *mc* (то есть *ограничивающий прямоугольник mc – bounding box*). Для получения значений, хранящихся в возвращаемом объекте, надо обращаться к его свойствам, как показано в следующем примере.

Размеры ограничивающего прямоугольника клипа могут измеряться относительно любого другого клипа или фильма. С помощью аргумента *targetCoordinateSpace* можно задать вопрос: «Если бы *mc* располагался на холсте *targetCoordinateSpace*, то какую область он бы занимал?» Ответ зависит от того, является ли *targetCoordinateSpace* главным фильмом или экземпляром клипа: начальная точка координатного пространства главного фильма находится в левом верхнем углу сцены, а начальная точка координатного пространства экземпляра находится в его точке регистрации, отмеченной в библиотечном символе клипа (показана перекрестием).

Метод *getBounds( )* позволяет обнаруживать коллизию между фильмом или клипом и какой-либо другой точкой (хотя *MovieClip.hitTest( )* лучше служит этой цели). Он также позволяет определить прямоугольную область для размещения клипа, добавляемого в фильм с помощью *MovieClip.attachMovie( )*.

Пример

```
// Получить границы текущего клипа
var clipBounds = this.getBounds();
// Записать границы в соответствующие переменные
var leftX   = clipBounds.xMin;
var rightX  = clipBounds.xMax;
var topY    = clipBounds.yMin;
var bottomY = clipBounds.yMax;
```

См. также

MovieClip.globalToLocal( ), *MovieClip.hitTest( )*, *MovieClip.localToGlobal*

MovieClip.getBytesLoaded(), метод

Flash 5

получить количество байтов, загруженных в проигрыватель

mc.getBytesLoaded()

Возвращаемое значение

Целое число, представляющее количество байтов *mc*, загрузка которых в проигрыватель завершена. (Разделить на 1024 для преобразования в килобайты.)

Описание

Метод *getBytesLoaded( )* сообщает о количестве байтов фильма или образа JPEG, загруженных в проигрыватель. Однако обычно данные поступают блоками по несколько сот или тысяч байт, и *getBytesLoaded( )* измеряет байты только с учетом этих блоков. Поэтому если первый кадр имеет размер 200 байт, а второй – 3000 байт, то *getBytesLoaded( )* возвратит 200 или 3200, но не число между ними.

Обратите внимание, что внутренние клипы всегда полностью загружаются перед воспроизведением, поэтому значение, возвращаемое *getBytesLoaded( )* для внутреннего

клипа, всегда совпадает с *getBytesTotal( )* (если только клип не загружает в данный момент внешний файл *.swf* в ответ на *loadMovie( )*). Поэтому *getBytesLoaded( )* обычно эффективен только при использовании с внешним файлом *.swf* или *.jpg*, который загружается в экземпляр или на уровень.

Функции *loadMovie( )* и *unloadMovie( )* выполняются асинхронно, поэтому значение *getBytesLoaded( )* зависит от того, когда происходит вызов, и от значения `mc`:

- Если *mc* имеет значение `undefined`, *getBytesLoaded( )* возвращает `undefined`.
- После обнаружения в исходном коде команд *loadMovie( )* или *unloadMovie( )*, но до того, как они будут выполнены или будет загружен какой-либо внешний файл, *getBytesLoaded( )* возвращает размер существующего (до команд *loadMovie( )* или *unloadMovie( )*) в *mc* содержимого. Пустой клип, созданный на этапе разработки, имеет размер 4 байта. Пустой клип, созданный с помощью *MovieClip.createEmptyMovieClip( )*, имеет размер 0 байтов.
- После того как извлечены какие-то байты, *getBytesLoaded( )* возвращает количество кадров, загруженных к данному времени, от 0 до `_totalframes`, если только файл на *url* найден.
- Если файл на *url* не найден, *getBytesLoaded( )* возвращает 0 (даже если *mc* является документом `_level`, который ранее не существовал). Однако *getBytesTotal( )* возвращает –1.
- После выполнения команды *unloadMovie( ) getBytesLoaded( )* возвращает 0, если *mc* является клипом, и возвращает `undefined`, если *mc* является уровнем документа, например `_level0`. То же относится к *getBytesTotal( )*.

Как и `_framesloaded`, *getBytesLoaded( )* обычно применяется для создания предзагрузчиков. Его можно использовать в сочетании с *getBytesTotal( )* для создания более точного индикатора загрузки, чем получается с `_framesloaded` и `_totalframes` (потому что размеры кадров могут быть неодинаковы: фильм с 10 кадрами не будет загружен на 30% после загрузки трех кадров, если кадры сильно отличаются по размеру).

Применение

К сожалению, во Flash 6 значение, возвращаемое *getBytesLoaded( )*, показывает размер файлов Flash *.swf* в разархивированном виде, а не размер файлов Flash после сжатия. Это означает, что *getBytesLoaded( )* для файла *.swf* с размером после сжатия 2000 байт может вернуть, например, значение 10 000 байт.

Пример

```
// КОД В КАДРЕ 1 ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЫ ФИЛЬМА
if (this._framesloaded > 0 && this._framesloaded == this._totalframes) {
  this.gotoAndPlay("beginMovie");
} else {
  // Показать ход загрузки в текстовых полях. Разделить на 1024 для преобразования в Кбайт.
  loadProgressOutput_txt.text = Math.floor(this.getBytesLoaded()/1024);
  loadTotalOutput_txt.text = Math.floor(this.getBytesTotal()/1024);
}

// КОД В КАДРЕ 2
this.gotoAndPlay(1);
```

См. также

loadMovie( ), *MovieClip.*`_framesloaded`, *MovieClip.getBytesTotal( )*, *MovieClip.loadMovie( )*, *MovieClip.*`_totalframes`, *MovieClip.unloadMovie( )*, *unloadMovie( )*

MovieClip.getBytesTotal(), метод

Flash 5

возвращает размер клипа или фильма в байтах

`mc.getBytesTotal()`

Возвращаемое значение

Целое число, представляющее размер не подвергнутого сжатию *mc* в байтах. Разделить на 1024 для преобразования в килобайты.

Описание

Метод *getBytesTotal( )* сообщает размер не подвергнутого сжатию экземпляра клипа или главного фильма в байтах. При вызове с главным фильмом *getBytesTotal( )* сообщает размер всего файла *.swf*. Обычно применяется вместе с *getBytesLoaded( )* при создании предварительных загрузчиков для главных фильмов и файлов *.swf* или *.jpg*, загружаемых в экземпляры или на уровни с помощью *loadMovie( )*

Функции *loadMovie( )* и *unloadMovie( )* выполняются асинхронно, поэтому значение *getBytesTotal( )* будет различным в зависимости от того, когда происходит вызов, и от значения `mc`:

- Если *mc* имеет значение `undefined`, *getBytesTotal( )* возвращает `undefined`.
- После обнаружения в исходном коде команд *loadMovie( )* или *unloadMovie( )*, но до того, как они будут выполнены и прежде чем будет определен размер внешнего файла, *getBytesTotal( )* возвращает размер существующего (до команд *loadMovie( )* или *unloadMovie( )*) в *mc* содержимого. Пустой клип, созданный на этапе разработки, имеет размер 4 байта. Пустой клип, созданный с помощью *MovieClip.createEmptyMovieClip( )*, имеет размер 0 байт.
- После того как размер файла определен, *getBytesTotal( )* возвращает полный размер *mc*, если только файл на *url* найден.
- Если файл на *url* не найден, *getBytesTotal( )* возвращает –1 (даже если *mc* представляет собой документ `_level`, который ранее не существовал), а *getBytesLoaded( )* возвращает 0.
- Если файл на *url* не найден, *getBytesTotal( )* возвращает 0 (даже если *mc* – это документ `_level`, который ранее не существовал). Однако *getBytesLoaded( )* возвращает –1.

После выполнения команды *unloadMovie( ) getBytesTotal( )* возвращает 0, если *mc* – это клип, и возвращает `undefined`, если *mc* является уровнем документа, например `_level0`. То же относится к *getBytesLoaded( )*.

Применение

Как и в случае *getBytesLoaded( )*, во Flash 6 значение, возвращаемое *getBytesTotal( )*, показывает размер файлов Flash *.swf* в разархивированном виде, а не размер файлов Flash после сжатия. Это означает, что *getBytesTotal( )* для файла *.swf* с размером после сжатия 2000 байт может вернуть, например, значение 10 000 байт. По этой причине его нельзя использовать для предсказания времени, оставшегося до конца загрузки сжатого файла *.swf*. Он может предложить только относительную меру той части файла, которая загружена, в случае применения совместно с *getBytesLoaded( )*.

См. также

loadMovie( ), *MovieClip.*`_framesloaded`, *MovieClip.getBytesLoaded( )*, *MovieClip.loadMovie( )*, *MovieClip.*`_totalframes`, *MovieClip.unloadMovie( )*, *unloadMovie( )*

MovieClip.getDepth(), метод

Flash 6

возвращает позицию клипа в стеке визуального содержимого

mc.getDepth()

Возвращаемое значение

Целое число, указывающее глубину *mc* в стеке визуального содержимого его родительского объекта.

Описание

Метод *getDepth( )* возвращает целое число – позицию *mc* в стеке содержимого его родительского клипа. Клипы с большей глубиной закрывают клипы с меньшей глубиной, поэтому клип с глубиной 6 находится перед клипом с глубиной 5, который, в свою очередь, расположен перед клипом с глубиной 4, и т. д. Глубины, превосходящие –1, зарезервированы для динамически создаваемого содержимого. Глубины от –16 383 до –1 включительно зарезервированы для содержимого, создаваемого на этапе разработки. Глубина –16 384 зарезервирована для динамического содержимого, которое должно оказаться в каждом клипе ниже любого содержимого среды разработки (например, для фона этапа исполнения).

Глубина каждого экземпляра инициализируется при создании клипа (в среде разработки Flash либо посредством *attachMovie( )*, *createEmptyMovieClip( )* или *duplicateMovieClip( )*). Однако экземпляр можно переместить на другую глубину с помощью *swapDepths( )*. Содержимое, помещенное на заданную глубину, замещает любое содержимое, ранее помещенное на эту глубину.

Важные подробности приведены в разделе главы 13 «Порядок размещения фильмов и экземпляров в стеке».

Пример

Следующий код создает три клипа и чертит в каждом прямую линию. Первый клип с красной линией создается за вторым клипом с синей линией. Затем третий клип с зеленой линией помещается на глубину клипа с красной линией, в результате чего заменяет его. Глубина клипа с красной линией `redline_mc` определяется динамически с помощью *getDepth( )*.

```
// Красная линия оказывается сзади(глубина 1)
this.createEmptyMovieClip("redline_mc", 1);
redline_mc.lineStyle(10, 0xFF0000);
redline_mc.lineTo(100, 10);

// Синяя линия оказывается спереди (глубина 2)
this.createEmptyMovieClip("blueline_mc", 2);
blueline_mc.lineStyle(10, 0x0000FF);
blueline_mc.lineTo(100, 0);

// Зеленая линия замещает красную, потому что была создана на глубине redline_mc
this.createEmptyMovieClip("greenline_mc", redline_mc.getDepth());
greenline_mc.lineStyle(10, 0x00FF00);
greenline_mc.lineTo(100, 10);
```

См. также

MovieClip.swapDepths( ); «Порядок размещения фильмов и экземпляров в стеке» в главе 13

MovieClip.getURL(), метод

Flash 5

загружает документ в окно броузера

```
mc.getURL(url)
mc.getURL(url, window)
mc.getURL(url, window, method)
```

Аргументы

url Строка, задающая местонахождение загружаемого документа или сценария, который должен быть выполнен.

window Необязательная строка, задающая имя окна или фрейма броузера, в которые должен быть загружен документ. Может быть пользовательским именем или одним из предустановленных имен: `"_blank"`, `"_parent"`, `"_self"` или `"_top"`.

method Необязательный строковый литерал, задающий метод отправки переменных из *mc* во внешний сценарий. Должен быть литералом `"GET"` или `"POST"`, другие выражения недопустимы.

Описание

Метод *MovieClip.getURL( )* служит альтернативой глобальной функции *getURL( )*. Этот формат метода *MovieClip* полезен только при отправке переменных, когда *getURL( )* посылает переменные из *mc*, необязательно являющегося текущей временной диаграммой. Общие инструкции по применению см. в описании глобальной функции *getURL( )*. Объектно-ориентированный способ отправки переменных серверу предоставляет *LoadVars.send( )*.

См. также *getURL( )*, *LoadVars.send( )*

MovieClip.globalToLocal(), метод

Flash 5

преобразует точку на главной сцене в координаты клипа

```
mc.globalToLocal(point)
```

Аргументы

point Ссылка на объект с двумя свойствами `x` и `y`, описывающими точку на главной сцене проигрывателя (то есть на `_level0`). Как `x`, так и `y` могут быть любыми числами с плавающей точкой.

Описание

Метод *globalToLocal( )* преобразует свойства `x` и `y` точки *point* из координат главного сцены в пространство координат *mc*. Обратите внимание, что *globalToLocal( )* не возвращает новый объект, а изменяет существующие значения `x` и `y` точки. Он не оказывает никакого действия при вызове с объектом главного фильма (то есть `_root`), поскольку исходное и целевое пространства координат совпадают.

Для применения *globalToLocal( )* надо сначала создать объект «точки» или «координат» со свойствами `x` и `y`. Для этого создадим общий объект класса *Object* и добавим в него свойства `x` и `y`. Например:

```
var myPoint = new Object();
myPoint.x = 10;
myPoint.y = 20;
```

Свойства x и y этого объекта являются положениями на горизонтальной и вертикальной осях главного сцены и измеряются относительно левого верхнего угла. Например, x, равный 10, означает 10 пикселов вправо от левого края стола, а y, равный 20, означает 20 пикселов вниз от верхнего края стола. После того как объект создан и свойства x и y установлены, объект передается методу *globalToLocal( )*:

```
myClip_mc.globalToLocal(myPoint);
```

После выполнения *globalToLocal( )* значения свойств x и y оказываются преобразованными так, что представляют точку в пространстве myClip при измерении от точки регистрации myClip. Новые значения свойств объекта myPoint позволяют узнать, где именно в myClip появляется точка *(x, y)* главной сцены. Например:

```
xInClip = myPoint.x;
yInClip = myPoint.y;
```

Пример

В следующем примере вычисляется смещение точки регистрации текущего клипа относительно левого верхнего угла главной сцены:

```
pt = new Object();          // Создать общий объект, содержащий точку
pt.x = 0;                   // Левая граница главной сцены
pt.y = 0;                   // Верхняя граница главной сцены
this.globalToLocal(pt);     // Преобразовать pt в локальные координаты

trace("From the current clip, the top-left corner of the main Stage is at ");
trace("x: " + pt.x + "y: " + pt.y);
```

См. также *MovieClip.localToGlobal( )*

MovieClip.gotoAndPlay(), метод

Flash 5

переход в указанный кадр и воспроизведение фильма или клипа

mc.gotoAndPlay(*frameNumber*)
mc.gotoAndPlay(*frameLabel*)

Аргументы

frameNumber Положительное целое, указывающее номер кадра, в который должна перейти головка воспроизведения *mc* перед воспроизведением. Если *frameNumber* меньше 1, то вызов *MovieClip.gotoAndPlay( )* игнорируется. Обратите внимание, что в глобальной версии *gotoAndPlay( )* значение, меньшее 1, обрабатывается как 1. Если *frameNumber* больше количества кадров во временной диаграмме *mc*, то головка воспроизведения направляется в последний кадр.

frameLabel Строка, указывающая метку кадра, в который должна перейти головка воспроизведения *mc* перед воспроизведением. Если *frameLabel* не найдена, операция не выполняется и ошибка не генерируется, а в глобальной версии *gotoAndPlay( )* несуществующая метка кадра обрабатывается как кадр 1.

Описание

Метод *MovieClip.gotoAndPlay( )* представляет собой альтернативу глобальной функции *gotoAndPlay( )* и позволяет управлять удаленными клипами и фильмами, задаваемыми *mc*.

Общие рекомендации по применению можно найти в описании глобальной функции *gotoAndPlay( )*.

Пример

```
// Отправить клип part1_mc на метку intro, затем воспроизвести part1_mc
part1_mc.gotoAndPlay("intro");
```

См. также *gotoAndPlay( )*, *MovieClip.gotoAndStop( )*

MovieClip.gotoAndStop(), метод

Flash 5

выполняет переход в указанный кадр и останавливает головку воспроизведения

mc.gotoAndStop(*frameNumber*)
mc.gotoAndStop(*frameLabel*)

Аргументы

frameNumber Положительное целое, указывающее номер кадра, в который должна перейти головка воспроизведения *mc*. Если *frameNumber* меньше 1, то вызов *MovieClip.gotoAndPlay( )* игнорируется. Обратите внимание, что в глобальной версии *gotoAndPlay( )* значение, меньшее 1, обрабатывается как 1. Если *frameNumber* больше количества кадров во временной диаграмме *mc*, то головка воспроизведения направляется в последний кадр.

frameLabel Строка, указывающая метку кадра, в который должна перейти головка воспроизведения *mc*. Если *frameLabel* не найдена, операция не выполняется и ошибка не генерируется, а в глобальной версии *gotoAndPlay( )* несуществующая метка кадра обрабатывается как кадр 1.

Описание

Метод *MovieClip.gotoAndStop( )* представляет собой альтернативу глобальной функции *gotoAndStop( )* и применяется для управления удаленными клипами и главными фильмами, задаваемыми *mc*.

Общие рекомендации по применению приведены в описании глобальной функции *gotoAndStop( )*.

Пример

```
// Направить клип mainMenu_mc в кадр 6 и остановить там головку воспроизведения
mainMenu_mc.gotoAndStop(6);
```

См. также *gotoAndStop( )*, *MovieClip.gotoAndPlay( )*

MovieClip._height, свойство

Flash 4; усилено во Flash 5

высота клипа или фильма в пикселах

только для чтения во Flash 4; чтение/запись во Flash 5

mc._height

Описание

Свойство _height, имеющее тип числа с плавающей точкой, представляет собой неотрицательное число, указывающее высоту *mc* в пикселах. Если в *mc* нет содержимого,

_height равно 0. Свойство _height измеряет содержимое клипа как расстояние от самого верхнего занятого пиксела до самого нижнего занятого пиксела, даже если между этими пикселами есть пустое пространство. *Занятый пиксел* – это пиксел, являющийся частью фигуры, изображения, кнопки, клипа, текстовоого поля или другого элемента содержимого. Свойство _height отражает изменения, внесенные в клип в среде разработки или через _yscale.

Присваивание значения свойству _height изменяет размер клипа по вертикали. Попытка присвоить _height отрицательное значение игнорируется. Присваивание свойству _height значения 0 не делает клип невидимым, а превращает его в горизонтальную линию толщиной в один пиксел. Во Flash базовой единицей измерения является одна двадцатая часть пиксела («twip»), поэтому минимально допустимое изменение _height составляет 0,05 пиксела. Меньшие изменения игнорируются:

```
trace(ball_mc._height);   // Выводит: 100.
ball_mc._height += .01;
trace(ball_mc._height);   // По-прежнему выводит 100. Присваивание не имело результата.
```

Значение _height для главного фильма (то есть root._height или _level*n*._height) является не высотой сцены, указанной в Modify → Document → Dimensions (или Modify → Movie → Dimensions во Flash 5) в среде разработки, а высотой содержимого главного фильма. Во Flash 6 заданные для фильма размеры можно получить через свойства Stage.height и Stage.width.

Для фильма, находящегося в процессе загрузки, свойство _height возвращает undefined, пока фильм не будет загружен полностью.

Применение

Обратите внимание, что при установке высоты клипа пропорционально масштабируются линии, утрачивая первоначальную толщину, установленную в панели Stroke. Однако если в панели Stroke для линий установлена толщина Hairline, то она не меняется при изменении размеров клипа. Таким образом, чтобы штрихи не становились толстыми и не искажались при изменении масштаба клипа, следует задавать для них толщину Hairline.

Пример

```
ball_mc._height = 20;          // Установить высоту ball_mc равной 20 пикселам
ball_mc._height /= 2;          // Уменьшить высоту ball_mc в 2 раза
```

См. также *MovieClip.*_width, *MovieClip.*_yscale, Stage.height

MovieClip.hitArea, свойство

Flash 6

задает активную область для клипа с поведением кнопки **чтение/запись**

mc.hitArea

Описание

Свойство hitArea применяется только с клипами, в которых определены обработчики событий кнопок (например, *onPress( )*, *onRelease( )* и т. д.). Оно задает активную область кнопки *mc* – физическую область, которая должна находиться под курсором мыши, чтобы кнопка могла быть активизирована щелчком мыши. Если hitArea имеет значение null или undefined, то в качестве активной области выступает форма содержимого *mc*. Если значение hitArea представляет собой ссылку на клип, активной

областью *mc* становится форма содержимого этого клипа. Клип hitArea можно перемещать, делать невидимым и присваивать вместо него другой, что обеспечивает большую гибкость при динамическом создании кнопок. Например, у анимированной кнопки может быть активная область, изменяющая свой размер в соответствии с анимацией, либо активная область кнопки в состояниях Over, Up и Down может быть разной.

Клип, играющий роль hitArea для *mc*, может располагаться на временной диаграмме любого клипа, включая *mc*. Например:

```
// Установить в качестве активной области button_mc клип, находящийся внутри него
button_mc.hitArea = button_mc.shape_mc;
// Установить в качестве активной области button_mc клип, находящийся вне его
button_mc.hitArea = _root.shapeClips.shape_mc;
```

Форма курсора мыши, попадающего в активную область, изменяется со стрелки на ладонь. Чтобы подавить отображение курсора в виде ладони, присвойте свойству useHandCursor значение false.

Дополнительные сведения о реализации поведения кнопки для клипа приводятся в описаниях класса *MovieClip*, метода *MovieClip.hitTest( )* и в разделе «Применение клипов в качестве кнопок» главы 13.

Пример

Следующий код создает клип, выполняющий функции кнопки, и рисует для него совершенно новую активную область. Он автоматически изменяет размер активной области, чтобы он вдвое превышал размер текста в кнопке. Благодаря явному определению свойства hitArea содержимое, добавляемое впоследствии к кнопке, не включается в активную область.

```
// Создать клип, который будет действовать как кнопка
this.createEmptyMovieClip("button_mc", 1);

// Добавить в клип некоторый текст
button_mc.createTextField("button_txt", 1, 0, 0, 0, 0);
button_mc.button_txt.autoSize = true;
button_mc.button_txt.text = "click here!";
button_mc.button_txt.selectable = false;

// Определить обработчик кнопки для клипа кнопки
button_mc.onPress = function () {
  trace("thank you for clicking");
}

// Создать клип, который станет активной областью клипа кнопки,
// и поместить его внутрь клипа кнопки
button_mc.createEmptyMovieClip("hitarea_mc", 2);
// Сделать прямоугольник активной области вдвое большим, чем текст кнопки
var x1 = -button_mc.button_txt._width / 2;
var y1 = -button_mc.button_txt._height / 2;
var x2 = x1 + button_mc.button_txt._width * 2;
var y2 = y1 + button_mc.button_txt._height * 2;
// Draw the hit area rectangle
button_mc.hitarea_mc.moveTo(x1, y1);
button_mc.hitarea_mc.beginFill(0x000000);
button_mc.hitarea_mc.lineTo(x2, y1);
button_mc.hitarea_mc.lineTo(x2, y2);
button_mc.hitarea_mc.lineTo(x1, y2);
```

```
button_mc.hitarea_mc.lineTo(x1, y1);
button_mc.hitarea_mc.endFill();

// Сделать клип активной области невидимым
button_mc.hitarea_mc._visible = false;

// Назначить клип активной областью кнопки
button_mc.hitArea = button_mc.hitarea_mc;
```

См. также

Класс *Button*, *MovieClip*.enabled, *MovieClip.hitTest( )*, *MovieClip*.useHandCursor; «Применение клипов в качестве кнопок» главы 13

MovieClip.hitTest(), метод

Flash 5

проверяет принадлежность точки клипу или пересечение двух клипов

mc.hitTest(*x*, *y*, *shapeFlag*)
mc.hitTest(*target*)

Аргументы

x Горизонтальная координата проверяемой точки.

y Вертикальная координата проверяемой точки.

shapeFlag

Необязательное булево значение, указывающее, надо ли проверять пересечение точки с ограничивающим прямоугольником *mc* (false) или фактическими пикселами *mc* (true). Значение по умолчанию false. Обратите внимание, что *shapeFlag* применяется только в синтаксисе с аргументами *x* и *y*, а не в синтаксисе с аргументом *target*. Он имеет смысл, только если у *mc* неправильные очертания или есть дырка, как в бублике; флаг не действует, если *mc* является сплошным прямоугольным объектом.

target

Строка, указывающая путь к клипу, проверяемому на коллизию с *mc*. Поскольку ссылки на клипы преобразуются в пути при использовании в строковом контексте, *target* может также являться ссылкой на объект клипа, например *mc.hitTest(*ball*)*, а не *mc.hitTest(*"ball"*)*.

Возвращаемое значение

Булево значение, указывающее результат проверки коллизии. Равно true, если выполняется любое из следующих условий:

- Точка *(x,y)* на главной сцене пересекается с каким-либо занятым пикселом *mc*. Занятым считается пиксел, содержащий визуальный элемент, например фигуру или текст.
- Свойство *shapeFlag* имеет значение false и точка *(x,y)* на главной сцене пересекается с каким-либо пикселом в ограничивающем прямоугольнике *mc*. Ограничивающим прямоугольником *mc* считается наименьший прямоугольник, в который входят все занятые пикселы *mc* (такой прямоугольник возвращает *MovieClip.getBounds( )*).
- Какой-либо пиксел в ограничивающем прямоугольнике *target* пересекается с каким-либо пикселом в ограничивающем прямоугольнике *mc*.

Результат имеет значение `false` в любом из следующих случаев:

- Свойство *shapeFlag* имеет значение `true` и точка *(x, y)* на главной сцене не пересекается с каким-либо занятым пикселом *mc*. Занятым считается пиксел, содержащий визуальный элемент, например фигуру или текст.
- Точка *(x,y)* на главной сцене не пересекается с каким-либо пикселом в ограничивающем прямоугольнике *mc*. Ограничивающим прямоугольником *mc* считается наименьший прямоугольник, в который входят все занятые пикселы *mc*.
- Никакие пикселы ограничивающей рамки *target* не пересекаются ни с какими пикселами в ограничивающем прямоугольнике *mc*.

Описание

Метод *hitTest( )* позволяет определить, пересекается ли клип или конкретная точка с *mc*.

При проверке пересечения точки с *mc* в *hitTest( )* передаются координаты *x* и *y* проверяемой точки (относительно главной сцены). Можно также передать необязательный аргумент *shapeFlag*, определяющий, должны ли при проверке коллизии использоваться фактические пикселы *mc* или ограничивающий прямоугольник *mc* (наименьший прямоугольник, в который входят все занятые пикселы *mc*). Проверка фактических пикселов *mc* позволяет определить, является ли точка *(x, y)* занятым пикселом внутри контура *mc*, а не только ее нахождение внутри ограничивающего прямоугольника *mc*.

При проверке пересечения клипа с *mc* в *hitTest( )* передается аргумент *target*, задающий проверяемый клип. Обнаружение коллизий между *target* и *mc* всегда осуществляется по ограничивающим прямоугольникам клипов: *hitTest( )* не поддерживает попиксельную проверку пересечения двух клипов. Ручные процедуры обнаружения коллизий на уровне пикселов трудно разрабатывать, и они интенсивно загружают процессор. Во многих случаях, например в простой космической игре, часто проверяется ограничивающий круг, а не конкретные пикселы (см. следующий пример).

На значение, возвращаемое *hitTest( )*, не оказывают влияния свойства `_visible` или `_alpha`, благодаря которым клип может сделаться невидимым. Невидимый клип можно так же проверять на пересечение с точкой или другим невидимым клипом.

Применение

Обратите внимание, что коллизия всегда проверяется относительно расположения *mc* на главной сцене проигрывателя, поэтому при задании в *hitTest( )* одной точки аргументы `x` и `y` всегда должны описывать точку в координатах сцены. О том, как преобразовывать координаты клипа в координаты сцены, рассказано в описании *MovieClip.localToGlobal( )*. При проверке коллизии между двумя клипами координаты клипов в различных временных диаграммах автоматически преобразуются в глобальные координаты (то есть координаты сцены).

Пример

В этом примере показано, как вручную определить пересечение двух окружностей, не прибегая к *hitTest( )*:

```
// Получить расстояние между двумя круглыми клипами по обеим осям
var deltaX = clip1._x - clip2._x;  // Расстояние по горизонтали
var deltaY = clip1._y - clip2._y;  // Расстояние по вертикали

// Сохранить радиус каждого круга во вспомогательном свойстве
var radius1 = clip1._width / 2;
var radius2 = clip2._width / 2;
```

```
// Если квадрат расстояния между центрами окружностей меньше
// или равен квадрату суммы длин радиусов, то происходит пересечение.
if ((deltaX * deltaX) + (deltaY * deltaY)
    <= (radius1 + radius2) * (radius1 + radius2)) {
  trace("intersecting");
} else {
  trace("not intersecting");
}
```

А так можно проверить пересечение paddle_mc и ball_mc с помощью *hitTest( )*:

```
if (paddle_mc.hitTest("ball_mc")) {
  trace("The paddle hit the ball.");
}
```

Теперь проверим, находится ли курсор мыши над занятым пикселом внутри контура домика house_mc. Обратите внимание, что координаты курсора даны относительно _root (главной сцены). Пробелы в клипе учитываются, если *shapeFlag* имеет значение true. Например, если окна в house_mc представлены пустым пространством и курсор находится над окном, то *hitTest( )* возвратит false:

```
if (house_mc.hitTest(_root._xmouse, _root._ymouse, true)) {
  trace("You're pointing to the tractor.");
}
```

См. также *MovieClip.getBounds( )*, *MovieClip.localToGlobal( )*

MovieClip.lineStyle(), метод

Flash 6

задает толщину, цвет и прозрачность штриха для черчения на этапе исполнения

mc.lineStyle(*thickness*, *RGB*, *alpha*)

Аргументы

thickness

Целое число – толщина линии в точках в диапазоне от 0 до 255, причем 0 задает линию с толщиной Hairline. Если аргумент не задан, линии не чертятся. Значения, меньшие 0, интерпретируются как 0, а большие 255 – как 255.

RGB

Целое число, характеризующее цвет линии, обычно в формате шестнадцатеричной тройки RGB – 0x*RRGGBB*, где *RR*, *GG* и *BB* – это двузначные шестнадцатеричные числа, представляющие красную, зеленую и синюю составляющие. Например, шестнадцатеричное число 0xFF0000 задает красный цвет. Если аргумент не задан, то по умолчанию принимается черный цвет. Формат троек RGB подробно обсуждается в описании класса *Color*.

alpha

Целое число в диапазоне от 0 до 100, задающее степень непрозрачности (или, наоборот, прозрачности) фигуры в процентах: 0 соответствует полной прозрачности, а 100 – полной непрозрачности. По умолчанию принимается равным 100. Значения, большие 100, интерпретируются как 100, а меньшие 0 интерпретируются как 0.

Описание

Метод *lineStyle( )* устанавливает толщину, цвет и прозрачность штриха для всех последующих вычерчиваний в *mc* с помощью методов *lineTo( )* и *curveTo( )*. По умолчанию при черчении в *mc* обводка не применяется. Метод *lineStyle( )* следует вызывать явно для каждого нового клипа и после каждого вызова *clear( )*, иначе обводка не появится (но можно вычерчивать закрашенные области без обводки).

В следующих примерах представлены допустимые способы модифицирования стиля линии для drawing_mc:

```
drawing_mc.lineStyle()               // Сбрасывает стиль (прямые и кривые не проводятся)
drawing_mc.lineStyle(1)              // Сплошная черная линия толщиной 1 pt
drawing_mc.lineStyle(1,0x00FF00)     // Сплошная зеленая линия толщиной 1 pt
drawing_mc.lineStyle(1,0x00FF00, 50) // Полупрозрачная зеленая линия толщиной 1 pt
```

Стиль линии можно в любой момент изменить, вызвав *lineStyle( )* снова. Например, следующий код чертит квадрат утолщающимися каждый раз линиями черного, красного, зеленого и синего цвета:

```
this.createEmptyMovieClip("drawing_mc", 2);

drawing_mc.lineStyle(1, 0x000000, 100);
drawing_mc.lineTo(100, 0);
drawing_mc.lineStyle(5, 0xFF0000, 100);
drawing_mc.lineTo(100, 100);
drawing_mc.lineStyle(10, 0x00FF00, 100);
drawing_mc.lineTo(0, 100);
drawing_mc.lineStyle(15, 0x0000FF, 100);
drawing_mc.lineTo(0, 0);
```

Значение *thickness*, равное 0, задает черчение «волосяными» линиями (линия толщиной в один пиксел, которая не меняется при увеличении масштаба клипа или фильма). Для того чтобы убрать обводку линий полностью, установите для толщины значение undefined или вызовите *lineStyle( )* без параметров. Например:

```
drawing_mc.lineStyle(undefined);   // Отключить линии в drawing_mc
drawing_mc.lineStyle();            // То же
```

Даже если для клипа определен стиль линии, можно переместить перо в новую точку, ничего при этом не вычерчивая, с помощью *MovieClip.moveTo( )*. При вызове метода *clear( )* для удаления всего, что начерчено в клипе, стиль линии клипа вновь принимает значение undefined (без линий).

См. также

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineTo( )*, *MovieClip.moveTo( )*

MovieClip.lineTo(), метод

Flash 6

вычерчивает прямую линию на этапе исполнения

mc.lineTo(*x*, *y*)

Аргументы

x Число с плавающей точкой. Горизонтальная координата конечной точки линии, которую требуется провести. Измеряется относительно точки регистрации *mc*.

y Число с плавающей точкой. Вертикальная координата конечной точки линии, которую требуется провести. Измеряется относительно точки регистрации *mc*.

Описание

Метод *lineTo( )* проводит прямую линию от текущего положения пера до точки (*x*,*y*) (описание пера см. в *MovieClip.moveTo( )*). Характеристики штриха линии (толщина, цвет и прозрачность) определяются последним обращением к *lineStyle( )*. Если толщина линии задана в *lineStyle( )* как `undefined`, то *lineTo( )* перемещает перо, но не вычерчивает линию на экране (как при вызове *moveTo( )*). Начерченная прямая появляется над всем содержимым, уже созданным Drawing API в *mc*, но ниже всего остального содержимого в *mc* (например, клипов, фигур, текста и кнопок, размещенных на этапе разработки, или содержимого, прикрепленного на этапе выполнения с помощью *attachMovie( )*, *duplicateMovieClip( )* или *createTextField( )*). Если отсутствует какой-либо из аргументов *lineTo( )*, операция не выполняется и ошибка не генерируется.

Следующий код создает клип с именем `drawing_mc` и проводит черную линию толщиной 2 пункта от текущей позиции пера (0,0) вниз до точки (0, 100). Пока положение пера не изменилось в результате вызова *moveTo( )*, *lineTo( )* или *curveTo( )*, оно совпадает с точкой регистрации `mc` или с левым верхним углом сцены, если *mc* представляет собой `_root`.

```
this.createEmptyMovieClip("drawing_mc", 1);  // Создать клип для черчения в нем
drawing_mc.lineStyle(2);             // Задать черчение черным штрихом толщиной 2 пункта
drawing_mc.lineTo(0, 100);           // Начертить линию
```

После того как кривая или прямая начерчена, перо остается в ее конечной точке, поэтому с помощью последовательности вызовов *lineTo( )* и/или *curveTo( )* легко начертить непрерывную ломаную линию:

```
// Создать клип для черчения в нем
this.createEmptyMovieClip("drawing_mc", 1);

// Задать черчение красным штрихом толщиной 1 пункт
drawing_mc.lineStyle(1, 0xFF0000);

// Начертить ломаную линию как ряд отрезков прямой
drawing_mc.lineTo(50, 100);
drawing_mc.lineTo(-20, 150);
drawing_mc.lineTo(75, 200);
drawing_mc.lineTo(50, 300);
```

Аналогично с помощью последовательности вызовов *lineTo( )* и/или *curveTo( )* можно чертить замкнутые фигуры. Подробности см. в описании *MovieClip.beginFill( )*.

Разумеется, клипами, в которых есть содержимое Drawing API, можно управлять как обычно. Например, следующий код заставит наш клип `drawing_mc` вращаться:

```
drawing_mc.onEnterFrame = function () {
  this._rotation++;
}
```

Упражнение. Обратите внимание, что *lineTo( )* принимает абсолютные, а не относительные координаты. Напишите пользовательский метод *lineToDelta( )*, принимающий относительные координаты. Например, *mc.lineToDelta(5, 10)* должен провести линию в точку, находящуюся на 5 пикселов правее и на 10 пикселов ниже текущей позиции пера. (Совет: посмотрите пример C.3 в описании *MovieClip.moveTo( )*, в котором сохраняется текущее положение пера.)

См. также

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.moveTo( )*

MovieClip.loadMovie(), метод

Flash 5; во Flash 6 добавлены загрузка .jpg и ограничения, связанные с безопасностью

загружает в проигрыватель внешний файл .swf

```
mc.loadMovie(url)
mc.loadMovie(url, method)
```

Аргументы

url

Строка, задающая местонахождение внешнего файла *.swf* или *.jpg*, который должен быть загружен.

method

Необязательный строковый литерал, задающий метод пересылки переменных во внешний сценарий. Должен быть литералом `"GET"` или `"POST"`; другие выражения недопустимы.

Описание

Метод *MovieClip.loadMovie( )* является альтернативой глобальной функции *loadMovie( )*. При вызове в качестве метода *MovieClip loadMovie( )* не принимает параметр `target`; он загружает в `mc` файл *.swf*, или *.jpg*, находящийся по указанному `url`. Метод меньше подвержен возможным синтаксическим ошибкам пользователя, чем соответствующая глобальная функция, поскольку не нужен параметр `target`.

Рекомендации по применению см. в описании глобальной функции *loadMovie( )*.

См. также

loadMovie( ), *loadMovieNum( )*, *MovieClip.getURL( )*; «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13

MovieClip.loadVariables(), метод

Flash 5

возвращает группу внешних переменных

```
mc.loadVariables(url)
mc.loadVariables(url, method)
```

Аргументы

url

Строка, указывающая путь к одному из двух источников переменных: сценарию, создающему на выходе переменные, или текстовому файлу, содержащему переменные.

method

Необязательный строковый литерал, определяющий метод отправки переменных из `mc` во внешний сценарий. Если этот аргумент задан, осуществляются как отправка, так и загрузка переменных. Если опущен, переменные только загружаются. Может быть литералом `"GET"` или `"POST"`, другие выражения недопустимы.

Описание

Метод *MovieClip.loadVariables( )* представляет собой альтернативу глобальной функции *loadVariables( )*. При вызове в качестве метода *MovieClip loadVariables( )* не принимает параметр *target*; он загружает переменные из *url* в *mc*. Синтаксис метода *MovieClip* меньше подвержен возможным ошибкам пользователя, чем у соответствующей глобальной функции, поскольку не нужен параметр *target*.

Рекомендации по применению приведены в описании глобальной функции *loadVariables( )*. Во Flash 6 предпочтительней использовать класс *LoadVars*, а не метод *MovieClip.loadVariables( )*, поскольку этот класс предоставляет более гибкие и мощные средства импорта и экспорта переменных.

См. также

loadVariables( ), класс *LoadVars*; «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13

MovieClip.localToGlobal(), метод

Flash 5

преобразует точку в клипе в координаты главной сцены

mc.localToGlobal(*point*)

Аргументы

point

Ссылка на объект с двумя свойствами x и y, описывающими точку в пространстве координат *mc*. Как x, так и y могут быть любыми числами с плавающей точкой.

Описание

Метод *localToGlobal( )* преобразует свойства x и y объекта *point* из координат, заданных в пространстве координат *mc*, в координаты на главной сцене проигрывателя. Обратите внимание, что *localToGlobal( )* не возвращает новый объект, а лишь модифицирует существующие значения x и y в *point*. Он не оказывает никакого действия при вызове с объектом главного фильма (то есть _root), поскольку исходное и целевое пространства координат совпадают.

Прежде чем вызывать *localToGlobal( )*, надо создать объект «точки», или «координатный», со свойствами x и y. Для этого создадим общий объект класса *Object* и добавим в него свойства x и y:

```
myPoint = new Object();
myPoint.x = 10;
myPoint.y = 20;
```

Свойства x и y этого объекта являются положениями на горизонтальной и вертикальной осях *mc* относительно точки регистрации *mc* (отображаемой как перекрестие в библиотечном символе *mc*). Например, x, равный 10, означает 10 пикселов вправо от точки регистрации *mc*, а y, равный 20, означает 20 пикселов вниз от точки регистрации *mc*. После того как объект создан и свойства x и y установлены, объект передается методу *localToGlobal( )*:

```
myClip.localToGlobal(myPoint);
```

В результате выполнения *localToGlobal( )* значения свойств x и y у myPoint оказываются преобразованными так, что представляют соответствующую точку на главной сцене, отсчитанную от его левого верхнего угла. Новые значения свойств объекта

myPoint позволяют узнать, «в каком месте главной сцены появляется точка клипа *(x, y)*». Например:

```
mainX = myPoint.x;
mainY = myPoint.y;
```

Пример

В следующем примере определяется расположение точки регистрации клипа относительно главной сцены:

```
pt = new Object();          // Создать общий объект, содержащий точку
pt.x = 0;                   // Положение точки регистрации клипа по горизонтали
pt.y = 0;                   // Положение точки регистрации клипа по вертикали
this.localToGlobal(pt);     // Преобразовать pt в координаты главной сцены

trace("On the main Stage, the registration point of the current clip is at: ");
trace("x: " + pt.x + "y: " + pt.y);
```

Посредством метода *localToGlobal( )* можно преобразовывать точку в координаты сцены – в случае совместного применения с методом *hitTest( )*, которому надо передавать точки в пространстве координат сцены. Он также позволяет сравнивать точки в двух различных клипах с помощью общего пространства координат.

См. также *MovieClip.globalToLocal( )*, *MovieClip.hitTest( )*

MovieClip.moveTo(), метод

Flash 6

перемещает перо в новое положение, не вычерчивая линию

mc.moveTo(*x*, *y*)

Arguments

x Новая горизонтальная координата пера (число с плавающей точкой). Измеряется относительно точки регистрации *mc*.

y Новая вертикальная координата пера (число с плавающей точкой). Измеряется относительно точки регистрации *mc*.

Описание

Вычерчивание прямых и кривых во Flash основано на так называемом теоретическом «чертежном пере». Во всех новых клипах перо первоначально помещается в точку (0,0) – точку регистрации клипа, которая обычно находится в его центре и представлена перекрестием в библиотечном символе клипа. При вычерчивании кривых и прямых (с помощью *lineTo( )* и *curveTo( )*) оно перемещается в пространстве координат клипа, останавливаясь в конечной точке последней начерченной прямой или кривой линии. Текущее положение пера всегда указывает начальную точку прямой или кривой линии, которая будет вычерчиваться следующей.

Метод *moveTo( )* позволяет произвольно устанавливать начальную точку прямой или кривой, «поднимая» перо и перемещая его в новое положение без проведения линии в указанную точку.

Допустим, например, что требуется начертить одну прямую линию из точки (100,100) в точку (200,200) в новом клипе. Сначала надо переместить перо из точки (0,0) в (100,100), а затем провести оттуда прямую в точку (200,200):

```
this.createEmptyMovieClip("drawing_mc", 1); // Создать клип для черчения в нем
```

```
drawing_mc.lineStyle(1);                // Задать черчение черным штрихом толщиной 1 пункт
drawing_mc.moveTo(100, 100);            // Переместить перо без вычерчивания линии
drawing_mc.lineTo(200, 200);            // Начертить линии (при этом также перемещается перо)
```

При вызове метода *clear( )* для удаления всего, что начерчено в клипе, перо возвращается в точку (0,0). Если отсутствует один из аргументов *moveTo( )*, операция не выполняется и ошибка не генерируется.

Применение

Во Flash нет встроенной поддержки текущего положения пера, но методы *moveTo( )*, *lineTo( )*, *curveTo( )* и *clear( )* можно модифицировать, чтобы они сохраняли положение пера по горизонтали и вертикали в свойствах, как показано в примере C.3. Прежде чем заменять каждый встроенный метод, мы сохраняем старую версию в свойстве, например `origLineTo`. Это считается самой правильной практикой на тот случай, если понадобится восстановить функциональность, существовавшую по умолчанию, хотя здесь мы ее не применяем.

Пример C.3. Пользовательские свойства положения пера

```
MovieClip.prototype.origLineTo = MovieClip.prototype.lineTo;
MovieClip.prototype.lineTo = function (x, y) {
  this.penx = x;
  this.peny = y;
  this.origLineTo(x, y);
};

MovieClip.prototype.origMoveTo = MovieClip.prototype.moveTo;
MovieClip.prototype.moveTo = function (x, y) {
  this.penx = x;
  this.peny = y;
  this.origMoveTo(x, y);
};

MovieClip.prototype.origCurveTo = MovieClip.prototype.curveTo;
MovieClip.prototype.curveTo = function (controlX, controlY, anchorX, anchorY) {
  this.penx = anchorX;
  this.peny = anchorY;
  this.origCurveTo(controlX, controlY, anchorX, anchorY);
};

MovieClip.prototype.origClear = MovieClip.prototype.clear;
MovieClip.prototype.clear = function () {
  this.penx = 0;
  this.peny = 0;
  this.origClear();
};

// Применение:
this.createEmptyMovieClip("drawing_mc", 1);
drawing_mc.lineStyle(1, 0xFF0000);
drawing_mc.lineTo(50, 100);
trace(drawing_mc.penx);                     // Выводит: 50
drawing_mc.moveTo(-20, 150);
trace(drawing_mc.penx);                     // Выводит: -20
drawing_mc.curveTo(200, 200, 70, 300);
trace(drawing_mc.peny);                     // Выводит: 300
```

См. также

MovieClip.beginFill( ), *MovieClip.beginGradientFill( )*, *MovieClip.clear( )*, *MovieClip.curveTo( )*, *MovieClip.endFill( )*, *MovieClip.lineStyle( )*, *MovieClip.lineTo( )*

MovieClip._name, свойство

Flash 4

идентификатор экземпляра клипа в виде строки **чтение/запись**

mc.`_name`

Описание

Свойство `_name` указывает имя экземпляра клипа в виде строки. Изначально оно отражает имя экземпляра, установленное в панели инспектора свойств на этапе разработки, и поэтому неприменимо к главным фильмам (к главному фильму проще всего обращаться посредством глобального свойства `_root`).

Пример

Свойство `_name` позволяет определить, надо ли осуществлять какие-то действия над данным клипом, как мы это делали, генерируя серию клипов со звездами в примере для *MovieClip.onLoad( )*. Свойство `_name` можно применять и для присвоения клипу нового идентификатора. Например:

```
// Поменять ball на circle
ball._name = "circle";

// Теперь управлять прежним ball как circle
circle._x = 500;
```

См. также

MovieClip._target, *targetPath( )*; «Имена экземпляров» в главе 13

MovieClip.nextFrame(), метод

Flash 5

перемещает головку воспроизведения клипа или фильма на один кадр вперед и останавливает ее

mc.`nextFrame()`

Описание

Метод *MovieClip.nextFrame( )* представляет собой альтернативу глобальной функции *nextFrame( )* и позволяет управлять удаленными клипами или главными фильмами, заданными *mc*.

Общие рекомендации по применению можно найти в описании глобальной функции *nextFrame( )*.

Пример

```
// Переместить slideshow_mc на один кадр вперед и остановить головку воспроизведения
slideshow_mc.nextFrame();
```

См. также

gotoAndStop( ), *MovieClip.gotoAndStop( )*, *MovieClip.prevFrame( )*, *nextFrame( )*

MovieClip.onData(), обработчик события

Flash 5; во Flash 6 добавлен формат функции обратного вызова

обработчик, выполняемый при получении клипом переменных или части загружаемого клипа

```
mc.onData()

onClipEvent(data) {
 инструкции
}
```

Описание

Обработчик события *onData( )* представляет собой форму старого обработчика события *onClipEvent (data)* в виде функции обратного вызова и является его более современным аналогом. Обработчик *onData( )* выполняется, когда в клип загружены внешние данные. Он может запускаться в двух различных ситуациях в зависимости от типа загружаемых данных, которыми могут быть внешние переменные либо файл *.swf* или *.jpg*. Обратите внимание, что в качестве загрузчика данных событие *onData( )* устарело во Flash 6 благодаря методам контроля загрузки класса *LoadVars*. В качестве загрузчика данных событие *onData( )*, по моему опыту, содержит слишком много ошибок, чтобы пользоваться им во Flash 5, хотя во Flash 6.0.29.0 его работа представляется правильной.

Когда мы запрашиваем у сервера группу переменных с помощью *loadVariables( )*, необходимо дождаться их полной загрузки, прежде чем работать с содержащейся в них информацией. Когда клип получает конец пакета загружаемых переменных, генерируется событие *onData( )*, сообщая о том, что можно выполнять код, обращающийся к этим переменным.

Допустим, что у нас есть фильм гостевой книги, в которую посетители вводят комментарии, сохраняемые нами на сервере. Когда пользователь пытается прочесть комментарий, мы запрашиваем его у сервера с помощью *loadVariables( )*. Но прежде чем можно будет показать эти комментарии, придется задержаться на экране загрузки, пока не станет известно, что запрашиваемые данные стали доступны. Обработчик события *onData( )* сообщит нам, когда переменные станут доступны, и тогда мы сможем показать комментарии пользователю.

Следующий код показывает упрощенный фрагмент гостевой книги, загружающий переменные, когда пользователь щелкает по кнопке. В этом примере кнопка загружает две переменные в кодировке URL из текстового файла в клип с помощью *loadVariables( )*. Клип определяет обработчик события *onData( )*, который будет выполнен, когда переменные загрузятся. В этом обработчике мы показываем значения переменных. Мы знаем, что переменные можно показывать, поскольку код в обработчике выполняется только тогда, когда он запускается событием *onData( )* (то есть после получения данных):

```
// СОДЕРЖИМОЕ ФАЙЛА guestbook.txt
name=judith&message=hello

// КНОПКА ВНУТРИ КЛИПА
guestBook_mc.go_btn.onRelease = function {
  this._parent.loadVariables("guestbook.txt");
};

// ОБРАБОТЧИК В НАШЕМ КЛИПЕ
guestBook_mc.onData = function {
```

```
    trace(name);
    trace(message);
  };
```

Второй случай применения события *onData( )* относится к загрузке в клипы внешних файлов *.swf* и *.jpg* с помощью функции *loadMovie( )*. Когда файл *.swf* загружается в клип, он по умолчанию начинает автоматически воспроизводиться, даже если была загружена лишь его часть. Такой режим не всегда желателен, поскольку иногда требуется обеспечить загрузку всего или определенной части *.swf*-файла, прежде чем начинать воспроизведение. Это достигается применением обработчика события *onData( )* и некоторого кода предзагрузки.

Событие *onData( )* происходит каждый раз, когда клип, в который происходит загрузка, получает часть внешнего файла *.swf*. Определить, что такое «часть», не так просто, как можно было бы предположить. Для того чтобы было сгенерировано событие *onData( )*, должен быть загружен хотя бы один новый кадр внешнего файла *.swf* с момента последнего срабатывания события *onData( )* или начала загрузки файла *.swf*. Фактически за это время может загрузиться несколько кадров файла *.swf*, но один кадр – это минимальное количество, необходимое для того, чтобы вызвать событие *onData( )*.

Выполнение обработчиков событий *onData( )* привязано к воспроизведению кадров в проигрывателе. Отображая очередной кадр, интерпретатор проверяет, загружена ли какая-то часть внешнего файла *.swf* в клип, содержащий обработчик события *onData( )*. Если часть внешнего файла *.swf* загружена в такой клип и содержит хотя бы один новый кадр, то выполняется обработчик события *onData( )*. Эта процедура происходит один и только один раз на каждый отсчет частоты смены кадров (даже если головка воспроизведения остановлена).

Заметьте, что поскольку событие *onData( )* связано с кадрами, у фильмов с более высокой частотой смены кадров предзагрузчики должны выглядеть более гладкими, поскольку в них чаще происходит обновление статуса загрузки файлов *.swf* (или *.jpg*).

Точное количество событий *onData( )*, генерируемых во время операции *loadMovie( )*, зависит от распределения содержимого в загружаемом файле *.swf* и скорости соединения. Файл *.swf* с одним кадром, как бы велик он ни был, сгенерирует всего одно событие *onData( )*. С другой стороны, файл *.swf* с сотней кадров может сгенерировать до 100 отдельных событий *onData( )* в зависимости от частоты смены кадров в фильме, размера каждого кадра в байтах и скорости соединения. Если кадры большие, а соединение медленное, будет сгенерировано больше событий *onData( )* (вплоть до одного события для каждого кадра). Если кадры маленькие, а соединение быстрое, будет сгенерировано меньше событий *onData( )* (все 100 кадров могут быть переданы в промежутке между сменой в проигрывателе двух кадров, что даст только одно событие *onData( )*).

Как воспользоваться событием *onData( )* для создания предзагрузчика? Прежде всего, следует воспользоваться форматом обработчика *onClipEvent( )* в стиле Flash 5:

```
onClipEvent (data) {
  // код предварительной загрузки
}
```

Применение *onClipEvent(data)* вместо *onData( )* гарантирует, что обработчик не будет удален, когда мы загрузим наш файл *.swf*, поскольку первый прикрепляется к клипу на этапе разработки, а не создается в виде переменной, которая будет потеряна, когда клип загрузит свое новое содержимое. Тогда, если при вызове функции *loadMovie( )* будет происходить событие *onData( )*, мы будем знать, что идет загрузка внешнего файла *.swf*. Поэтому, находясь внутри обработчика события *onClip-*

Event(data), мы сможем проверить, достаточная ли часть файла загрузилась, чтобы можно было начать его воспроизведение. Мы делаем это с помощью методов *getBytesLoaded()* и *getBytesTotal()*, как показано в следующем примере. (Можно также воспользоваться свойствами клипа _framesloaded и _totalframes.)

```
onClipEvent (data) {
    trace("data received");            // Кино вот-вот начнется!

    // Включить индикатор передачи данных
    _root.transferIndicator.gotoAndStop("on");

    // Если загрузка завершена, выключить индикатор передачи данных и запустить показ фильма
    if (getBytesTotal() > 0 && getBytesLoaded() == getBytesTotal()) {
      _root.transferIndicator.gotoAndStop("off");
      play();
    }

    // Вывести некоторые подробности загрузки в текстовых полях на _root
    _root.bytesLoaded = getBytesLoaded();
    _root.bytesTotal = getBytesTotal();
    _
root.clipURL = _url.substring(_url.lastIndexOf("/") + 1, _url.length);
}
```

Этот пример также обеспечивает обратную связь во время загрузки фильма. Обратите внимание, что в первом кадре загружаемого файла *.swf* должно быть обращение к функции *stop()*, чтобы не началось его автоматическое воспроизведение до момента полной загрузки. Разновидность этого примера есть в сетевом хранилище кода.

Ошибки реализации

Во Flash 5 при высокой скорости соединения событие *onData()* может не возникнуть, когда файл *.swf* будет загружен целиком. Эта неофициальная ошибка не подтверждается Macromedia, но, по-видимому, исправлена в релизе 29 Flash Player 6 (6.0.29.0).

См. также

loadMovie(), *loadVariables()*, класс *LoadVars*, *MovieClip.getBytesLoaded()*, *MovieClip.getBytesTotal()*, *MovieClip._framesloaded*, *MovieClip._framestotal*, *MovieClip.onLoad()*; глава 10

MovieClip.onDragOut(), обработчик события

Flash 6

событие Button; происходит, когда кнопка мыши нажимается над клипом, а затем курсор уводится с клипа

```
mc.onDragOut()
```

```
on (dragOut) {
 инструкции
}
```

Описание

Обработчик события *onDragOut()* идентичен событию *Button.onDragOut()* за исключением того, что его можно также применять к экземплярам клипов (хотя и не к главным фильмам, таким как _root, _level0 и т. д.). Подробности см. с описании *Button.onDragOut()* и разделе «Применение клипов в качестве кнопок» в главе 13.

См. также

Button.onDragOut(), *MovieClip.onDragOver()*; глава 13

MovieClip.onDragOver(), обработчик события

Flash 6

событие Button; происходит, когда кнопка мыши нажимается над клипом, а затем курсор уводится с клипа и снова возвращается в него

```
mc.onDragOver()
```

```
on (dragOver) {
 инструкции
}
```

Описание

Обработчик события *onDragOver( )* идентичен событию *Button.onDragOver( )*, за исключением того, что его можно также применять к экземплярам клипов (хотя и не к главным фильмам, таким как `_root`, `_level0` и т. д.). Подробности см. с описании *Button.onDragOver( )* и в разделе «Применение клипов в качестве кнопок» в главе 13.

См. также *Button.onDragOver( )*, *MovieClip.onDragOut( )*; глава 13

MovieClip.onEnterFrame(), обработчик события

Flash 5; во Flash 6 добавлен формат функции обратного вызова

обработчик выполняется на каждый такт смены кадров во Flash Player

```
mc.onEnterFrame()
```

```
onClipEvent (enterFrame) {
  инструкции
}
```

Описание

Обработчик события *onEnterFrame( )* представляет собой форму старого обработчика события *onClipEvent (enterFrame)* в виде функции обратного вызова и является его более современным аналогом. Обработчик *onEnterFrame( )* выполняется один раз на каждый такт смены кадров фильма, например при перемещении вперед головки воспроизведения или при повторном выводе текущего кадра проигрывателем при остановленной головке воспроизведения. В любом случае частота события *onEnterFrame( )* напрямую связана с частотой смены кадров, реализуемой проигрывателем. Если частота смены кадров, установленная на этапе разработки в окне Modify → Document составляет 20 кадров в секунду, то теоретически отсчеты частоты смены кадров должны возникать через каждую 1/20 долю секунды. Но если проигрыватель не в состоянии воспроизводить фильм с заданной частотой смены кадров (например, из-за недостаточной мощности компьютера или слишком большого количества открытых приложений), фактическая частота смены кадров окажется ниже, а промежуток между отсчетами – больше. Например, если фактическая частота составит 12 кадров в секунду, один «тик» (промежуток между отсчетами) составит 1/12 секунды. Обратите внимание, что «тик» не является постоянной измерения времени и даже не определен как формальный термин ActionScript (в отличие от Lingo); он просто обозначает один цикл частоты смены кадров во Flash Player.

Следующий обработчик увеличивает размер `cloud_mc` на 10 пикселов с каждым тактом смены кадров:

```
cloud_mc.onEnterFrame = function () {
  this._height += 10;
  this._width += 10;
};
```

При воспроизведении во Flash Player все фильмы Flash непрерывно выполняются, даже если ничего не перемещается по экрану или головка фильма остановлена в кадре. По этой причине обработчик *onEnterFrame( )* отдельного клипа периодически выполняется, пока клип находится на сцене, независимо от того, воспроизводится клип или остановлен. Если головка воспроизведения фильма перемещается в результате вызова функции *gotoAndStop( )*, обработчик события *onEnterFrame( )* клипа продолжает запускаться с каждым новым кадром. И если все головки воспроизведения всего фильма остановлены с помощью функции *stop( )*, то все обработчики событий *onEnterFrame( )* всех клипов будут по-прежнему выполняться.

Событие *onEnterFrame( )* обычно применяется для периодического обновления состояния клипа. Но обработчик события *onEnterFrame( )* не обязательно применять непосредственно к тому клипу, в котором он находится. Его можно использовать на главной временной диаграмме фильма или на пустом клипе из одного кадра для периодического выполнения кода. Этот прием, называемый *циклом событий клипа* (*clip event loop*), или, более вольно, процессом, демонстрируется в разделе «Циклы событий временной диаграммы и клипа» в главе 8. Полезным инструментом для периодического выполнения кода является также функция *setInterval( )*.

Для того чтобы прекратить обработку функцией обратного вызова события *onEnterFrame( )*, применяется оператор *delete*, удаляющий обработчик, например:

```
// Не ставьте оператор () после имени функции!
delete theClip_mc.onEnterFrame;
```

Обратите внимание, что обработчик события *onEnterFrame( )* выполняется раньше, чем любой код на временной диаграмме клипа, содержащего обработчик.

Пример

Следующий код развивает наш предшествующий пример, вызывая увеличение размера клипа `cloud_mc` до заданной величины:

```
cloud_mc.maxHeight = 300

cloud_mc.onEnterFrame = function () {
  if (this._height < this.maxHeight) {
    this._height += 10;
  } else {
    delete this.onEnterFrame;
  }
}
```

См. также

setInterval( ); «Циклы событий временной диаграммы и клипа» в главе 8

MovieClip.onKeyDown(), обработчик события

Flash 5; во Flash 6 добавлен формат функции обратного вызова

обработчик, выполняемый при нажатии клавиши во время нахождения клипа в фокусе ввода

```
mc.onKeyDown()

onClipEvent(keyDown) {
  инструкции
}
```

Описание

Обработчик события *onKeyDown( )* представляет собой форму старого обработчика события *onClipEvent (keyDown)* в виде функции обратного вызова и является его более современным аналогом. Однако это один из немногих обработчиков, режим которого изменился в формате функции обратного вызова по сравнению с форматом *onClipEvent( )* в стиле Flash 5. Будучи реализованным в стиле функции обратного вызова, *onKeyDown( )* выполняется только тогда, когда нажата клавиша, и клип, определенный `mc`, находится в фокусе ввода с клавиатуры. По умолчанию фокус ввода с клавиатуры обозначается желтым прямоугольником. Однако во Flash 6 использование *onKeyDown( )* в формате функции обратного вызова таит пару коварных ловушек. Во-первых, клип без обработчиков кнопок не может получить фокус ввода, если явным образом не установить в `true` одно из свойств: `focusEnabled` или `tabEnabled`. Во-вторых, клип потеряет фокус ввода с клавиатуры, как только будет обнаружено перемещение мыши над Flash Player. В следующем коде мы задаем функцию обратного вызова *onKeyDown( )*, затем включаем возможность получения фокуса и, наконец, программно помещаем клип в фокус. Только после этого клип начинает вращаться при нажатии клавиш.

```
theClip_mc.onKeyDown = function () {
  this._rotation++;
}
theClip_mc.focusEnabled = true;
Selection.setFocus(theClip_mc);
```

Если попытаться выполнить этот код в режиме Test Movie, он будет работать только тогда, когда курсор мыши не находится над сценой фильма, в противном случае фокус будет немедленно утрачен, и событие не будет генерироваться. Такие особенности в целом ограничивают полезность события клипа *onKeyDown( )*. В действительности событие приемника *Key.onKeyDown( )*, а не *MovieClip.onKeyDown( )*, является, по замыслу Macromedia, универсальным инструментом реагирования на события клавиатуры. Однако *MovieClip.onKeyDown( )* приобретает значение, когда клипы действуют в качестве элементов интерфейса, управляемых с клавиатуры. Например, реализовав *onKeyDown( )* в двух отдельных клипах «меню», нетрудно заставить одну и ту же горячую клавишу запускать выполнение различных операций в зависимости от того, которое из меню находится в фокусе. К сожалению, такой сценарий может быть применен исключительно к интерфейсам, взаимодействующим только с клавиатурой, так как малейшее движение мыши снимает текущий фокус. Macromedia планирует совершенствовать свою реализацию фокуса клавиатуры в будущих версиях Flash, поэтому во Flash 7 может появиться другая модель для такой ситуации.

Напротив, версия обработчика в виде *onClipEvent(keyDown)* интуитивно понятна и ведет себя, как во Flash 5; он выполняется, когда нажимается любая клавиша без каких-либо исключений. Необходимо лишь, чтобы сам Flash Player находился в фокусе. Во Flash 6 событие приемника *Key.onKeyDown( )* обычно предпочтительнее, чем *MovieClip.onKeyDown( )*; см. подробную документацию в описании *Key.onKeyDown( )*. Однако в этой смеси с *onKeyDown( )* есть еще один и последний ингредиент – функция *updateAfterEvent( )* (которая обновляет экран в промежутке между кадрами) действует с *MovieClip.onKeyDown( )*, но не с *Key.onKeyDown( )*. Поэтому общие правила применения таковы:

- Если требуется общий механизм обработки нажатий клавиш, но не требуется обновление экрана между кадрами, применяйте *Key.onKeyDown( )*.

- Если требуется общий механизм обработки нажатий клавиш и приложение должно обновлять экран между кадрами, используйте *onClipEvent (keyDown)* в стиле Flash 5.
- Если необходимо управлять клипом с помощью клавиатуры, только когда он находится в фокусе ввода с клавиатуры, используйте *MovieClip.onKeyDown( )*.

Применение

Обработчик *onKeyDown( )* нельзя применять с временными диаграммами главного фильма (например, `_root`, `_level0`, `_level1`), поскольку главные временные диаграммы не могут получать фокус ввода.

См. также

Key.onKeyDown( ), *Key.onKeyUp( )*, *MovieClip*.`focusEnabled`, *MovieClip.onKeyUp( )*, *MovieClip*.`tabEnabled`

MovieClip.onKeyUp(), обработчик события

Flash 5; во Flash 6 добавлен формат функции обратного вызова

обработчик, выполняемый при освобождении клавиши во время нахождения клипа в фокусе

mc.`onKeyUp()`

```
onClipEvent(keyUp) {
  инструкции
}
```

Описание

Обработчик события *onKeyUp( )* представляет собой форму старого обработчика события *onClipEvent (keyUp)* в виде функции обратного вызова и является его более современным аналогом. Однако его поведение зависит от формы. При реализации в стиле функции обратного вызова *onKeyUp( )* выполняется только тогда, когда отпускается клавиша при нахождении клипа, определенного *mc*, в фокусе ввода с клавиатуры. Клип может получить фокус с помощью клавиши Tab или метода *Selection.setFocus( )*, но только если в нем определен хотя бы один обработчик кнопки или одно из свойств `tabEnabled`, или если `focusEnabled` имеет значение `true`. Например:

```
// Задать функцию обратного вызова onKeyUp()
theClip_mc.onKeyUp = function () {
  this._xscale++;  // Увеличить ширину клипа при освобождении клавиши.
}
// Разрешить установку фокуса
theClip_mc.focusEnabled = true;
// Поместить клип в фокус
Selection.setFocus(mc);
```

Учтите, что фокус ввода мгновенно утрачивается клипом при обнаружении малейшего движения мыши над Flash Player. Подробности см. в описании *MovieClip.onKeyDown( )*.

В форме *onClipEvent(keyUp)* обработчик *onKeyUp( )* выполняется всегда, когда освобождается любая клавиша. Для этого требуется только, чтобы сам Flash Player находился в фокусе. Например:

```
// Код в экземпляре клипа
onClipEvent (keyUp) {
  trace("A key was pressed!");
}
```

Во Flash 6, как правило, лучше работать с приемником событий *Key.onKeyUp( )*, а не *MovieClip.onKeyUp( )*; см. подробную информацию в описании объекта *Key*.

Применение

Во Flash 6 функция *updateAfterEvent( )*, обновляющая экран в промежутке между кадрами, может работать с *MovieClip.onKeyUp( )*, но не может работать с *Key.onKeyUp( )*.

Обработчик *onKeyUp( )* нельзя применять с временными диаграммами главного фильма (например, `_root`, `_level0`, `_level1`), поскольку главные временные диаграммы не могут получать фокус ввода.

См. также

Key.onKeyDown( ), *Key.onKeyUp( )*, *MovieClip*.focusEnabled, *MovieClip.onKeyDown( )*, *MovieClip*.tabEnabled

MovieClip.onKillFocus (), обработчик события

Flash 6

обработчик, выполняемый при потере клипом фокуса ввода с клавиатуры

mc.onKillFocus(*newFocus*)

Аргументы

newFocus

Объект *TextField*, *MovieClip* или *Button*, обладающий в данный момент фокусом, или `null`, если ни один объект не получил фокуса.

Описание

Обработчик события *onKillFocus( )* выполняется, когда *mc* теряет фокус ввода с клавиатуры. Клип может находиться в фокусе при соблюдении одного или нескольких из следующих условий:

- Он определяет хотя бы один обработчик события кнопки (*onPress( )*, *onRollOver( )* и т. д.).
- Его свойство `tabEnabled` имеет значение `true`, и тогда его можно поместить в фокус с помощью клавиши Tab.
- Его свойство `focusEnabled` имеет значение `true`, и тогда его можно поместить в фокус с помощью *Selection.setFocus( )*.

Потеря фокуса клипом происходит так же, как и потеря фокуса кнопкой, и описывается в *Button.onKillFocus( )*. Обработчик *onKillFocus( )* может применяться для снятия выделения с элемента интерфейса, находящегося в фокусе. В ряде простых случаев этого можно также достичь с помощью специальных меток кадров `_up` и `_over`. См. *MovieClip*.focusEnabled и *MovieClip*.`_focusrect`.

Метод *Selection.onSetFocus( )* позволяет организовать централизованный перехват всех событий фокуса, а не для отдельного экземпляра.

Применение

Обратите внимание, что *onKillFocus( )* нельзя использовать в блоке *onClipEvent( )* в стиле Flash 5.

Обработчик *onKillFocus( )* нельзя применять с временными диаграммами главного фильма (например, `_root`, `_level0`, `_level1`), поскольку главные временные диаграммы не могут получать фокус ввода.

См. также

Button.onKillFocus( ), *MovieClip*.focusenabled, *MovieClip*.tabEnabled, *MovieClip*.`_focusrect`, *Selection.setFocus( )*, *Selection.onSetFocus( )*

MovieClip.onLoad(), обработчик события

Flash 5; во Flash 6 добавлен формат функции обратного вызова

обработчик, выполняемый при первом появлении клипа на сцене или по окончании загрузки в клип файла .swf

```
mc.onLoad()

onClipEvent (load) {
  инструкции
}
```

Описание

Обработчик события *onLoad( )* представляет собой форму старого обработчика события *onClipEvent(load)* в виде функции обратного вызова и является его более современным аналогом. Обработчик события *onLoad( )* выполняется при «рождении» клипа – его первом появлении на сцене. Он не применяется к главным фильмам (например, `_root`, `_level1` и т. д.). Клип появляется на сцене (то есть генерирует событие *onLoad( )*) одним из следующих способов:

- Головка воспроизведения перемещается в ключевой кадр, содержащий новый экземпляр клипа, помещенный туда в среде разработки.
- Клип создан как копия другого клипа с помощью функции *duplicateMovieClip( )* или *MovieClip. duplicateMovieClip( )*.
- Клип помещен на сцену программно с помощью функции *attachMovie( )*.
- Внешний файл *.swf* загружен в клип с помощью функции *loadMovie( )* или *MovieClip. loadMovie( )*.

Тело обработчика события *onLoad( )* выполняется после любого кода на временной диаграмме, в которой впервые появляется клип.

При реализации в виде блока *onClipEvent(load)* в стиле Flash 5 обработчик события *onLoad( )* часто применяется для инициализации в клипе переменных или для выполнения некоторой задачи настройки (такой как задание размеров или положения динамически созданного клипа). С помощью обработчика события *onClipEvent(load)* удобно также предотвращать автоматическое воспроизведение клипа:

```
// Код, помещенный непосредственно в объект клипа на этапе разработки
onClipEvent (load) {
  stop();
}
```

С помощью обработчика *onClipEvent(load)* можно также запускать какую-нибудь функцию, правильное выполнение которой зависит от существования конкретного клипа или загруженного файла *.swf*.

В формате функции обратного вызова *onLoad( )* действует совершенно иначе, чем *onClipEvent(load)*: он действует только при назначении на уровне класса, а не на уровне экземпляра. В последнем случае он игнорируется.

Допустим, мы присваиваем функцию обратного вызова *onLoad( )* клипу, созданному с помощью *attachMovie( )* или *duplicateMovieClip( )*:

```
this.attachMovie("BallSymbol", "theBall_mc", 1);
// ЭТОТ КОД НИКОГДА НЕ ВЫПОЛНЯЕТСЯ:
theBall_mc.onLoad = function () {
  trace("loaded");
};
```

Когда выполняется первая строка, `theBall_mc` прикрепляется к текущему клипу, и проходит стадия *onLoad( )*. Когда выполняется вторая строка, функция обратного вызова присваивается `theBall_mc.onLoad`, но она никогда не выполнится, потому что клип оказывается уже загруженным, прежде чем назначается функция обратного вызова. Такой же неудачной окажется и попытка присвоить *onLoad( )* из кадра 1 клипа:

```
// ЭТОТ КОД НИКОГДА НЕ ВЫПОЛНЯЕТСЯ:
this.onLoad = function() {
  trace("loaded");
};
```

В отличие от своего родственника в стиле Flash 5 *onClipEvent(load)*, обработчик события *onLoad( )* не применяется к экземплярам клипов отдельно. Вместо этого он применяется на уровне класса для инициализации экземпляров подкласса *MovieClip*. Поэтому местом присваивания функции обратного вызова *onLoad( )* предполагается блок `#initclip` подкласса клипа. Например:

```
// КОД В КАДРЕ 1 СИМВОЛА КЛИПА "ballSymbol"
#initclip
// Конструктор Ball
function Ball () {
  // Здесь выполняется инициализация экземпляра ball...
}

// Задать MovieClip в качестве надкласса Ball
Ball.prototype = new MovieClip();

// Связать класс Ball с символом библиотеки по имени "ballSymbol"
Object.registerClass("ballSymbol", Ball);

// Присвоить обработчик onLoad всем экземплярам
Ball.prototype.onLoad = function () {
  trace(this._name + " loaded.");
}
#endinitclip
```

Подробные сведения о подклассах клипов приведены в описании прагмы `#initclip` и в главе 14.

Присваивание *onLoad( )* на уровне класса, приведенное в коде *Ball*, должно работать, но по существу это дублирование ролей. Код инициализации экземпляра обычно следует помещать внутрь функции конструктора класса.

Обратите внимание, что функция обратного вызова в клипе *onLoad( )* уничтожается, когда в клип загружается внешний файл *.swf* или *.jpg*, но его обработчик *onClipEvent(load)* сохраняется и будет выполняться в разных вызовах *loadMovie( )*. (Функция обратного вызова *onLoad( )* уничтожается, потому что свойство обработчика событий *onLoad( )* технически представляет собой функцию, хранящуюся в свойстве клипа; свойства клипа уничтожаются при удалении клипа из проигрывателя.)

Применение

Обработчик *onClipEvent(load)* в стиле Flash 5 выполняется, когда содержимое клипа выгружается с помощью функции *unloadMovie( )* (потому что при удалении содержимого клипа в клип загружается пустой клип-заместитель). Напротив, его собственная функция обратного вызова *onLoad( )* не выполняется при выгрузке содержимого клипа. Достаточно странно, что когда к одному и тому же клипу применяются оба типа обработчиков, они оба выполняются при вызове *unloadMovie( )*.

Пример

В следующем коде *onLoad( )* вызывается в формате *onClipEvent(load)*, чтобы генерировать клипы `star`, использующие один обработчик события в каскадной цепочке. Обработчик *onClipEvent(load)* прикрепляется к экземпляру клипа с именем `star0`. При загрузке клипа он дублирует себя и свой обработчик *onClipEvent(load)*, заставляя дубликат, в свою очередь, дублировать себя. Процесс останавливается, когда дублируется 100-й клип. Обратите внимание, что добиться такого же эффекта с помощью *onLoad( )* в форме функции обратного вызова нельзя, потому что функции обратного вызова не копируются в дублированные клипы.

Файл *.fla* для этого примера можно загрузить из сетевого хранилища кода:

```
// Прикрепить этот обработчик onLoad() к клипу с именем star0
onClipEvent (load) {
  // Поместить текущий клип в случайно выбранную точку
  _x = Math.floor(Math.random() * 550);
  _y = Math.floor(Math.random() * 400);

  // Сбросить масштаб клипа, чтобы он не наследовался от предыдущего
  _xscale = 100;
  _yscale = 100;

  // Случайным образом выбрать масштаб текущего клипа в интервале от 50% до 150%
  randScale = Math.floor(Math.random() * 100) - 50;
  _xscale += randScale;
  _yscale += randScale;

  // Если это не 100-й клип, сделать еще один
  if (_name != "star100") {
    nextStarNumber = Number(_name.substring(4, _name.length)) + 1;
    this.duplicateMovieClip("star" + nextStarNumber, nextStarNumber);
  }
}
```

MovieClip.onMouseDown(), обработчик события

Flash 5; во Flash 6 добавлен формат функции обратного вызова

обработчик, выполняемый, если во время нахождения клипа на сцене нажата главная кнопка мыши

```
mc.onMouseDown()

onClipEvent(mouseDown) {
 инструкции
}
```

Описание

Обработчик события *onMouseDown( )* представляет собой форму старого обработчика события *onClipEvent (mouseDown)* в виде функции обратного вызова и является его более современным аналогом. Обработчик события *onMouseDown( )* идентичен приемнику события *Mouse.onMouseDown( )* за исключением того, что он допускает обновление экрана между кадрами с помощью *updateAfterEvent( )*. Он выполняется, если во время нахождения *mc* на сцене нажата главная кнопка мыши. Обработчик *MovieClip.onMouseDown( )* оказался бы устаревшим во Flash 6, если бы *Mouse.onMouseDown( )* поддерживал *updateAfterEvent( )*. Подробные сведения можно найти в описании *Mouse.onMouseDown( )*.

Пример

```
// Формат функции обратного вызова
_root.onMouseDown = function () {
  trace("The mouse was pressed");
}

// Блок onClipEvent() в формате Flash 5 помещается непосредственно в экземпляр клипа
onClipEvent (mouseDown) {
  trace("The mouse was pressed");
}
```

См. также

Mouse.onMouseDown( ), *MovieClip.onMouseMove( )*, *MovieClip.onMouseUp( )*

MovieClip.onMouseMove(), обработчик события

Flash 5; во Flash 6 добавлен формат функции обратного вызова

обработчик, выполняемый при перемещении курсора мыши во время нахождения клипа на сцене

```
mc.onMouseMove()

onClipEvent(mouseMove) {
 инструкции
}
```

Описание

Обработчик события *onMouseMove( )* представляет собой форму старого обработчика события *onClipEvent (mouseMove)* в виде функции обратного вызова и является его более современным аналогом. Обработчик события *onMouseMove( )* идентичен приемнику события *Mouse.onMouseMove( )* за исключением того, что он допускает обновление экрана между кадрами с помощью *updateAfterEvent( )*. Он выполняется,

если во время нахождения *mc* на сцене перемещается курсор мыши. Обработчик *MovieClip.onMouseMove( )* оказался бы устаревшим во Flash 6, если бы *Mouse.onMouseMove( )* поддерживал *updateAfterEvent( )*. Подробные сведения можно найти в описании *Mouse. onMouseMove( )*.

Пример

```
// Формат функции обратного вызова
_root.onMouseMove = function () {
  trace("The mouse moved");
}

// Блок onClipEvent() в формате Flash 5 помещается непосредственно в экземпляр клипа
onClipEvent (mouseMove) {
  trace("The mouse moved");
}
```

См. также

Mouse.onMouseMove( ), *MovieClip.onMouseDown( )*, *MovieClip.onMouseUp( )*

MovieClip.onMouseUp(), обработчик события

Flash 6

обработчик, выполняемый во время освобождения главной кнопки мыши во время нахождения клипа на сцене

```
mc.onMouseUp()

onClipEvent (mouseUp) {
 инструкции
}
```

Описание

Обработчик события *onMouseUp( )* представляет собой форму старого обработчика события *onClipEvent (mouseUp)* в виде функции обратного вызова и является его более современным аналогом. Обработчик события *onMouseUp( )* идентичен приемнику события *Mouse.onMouseUp( )* за исключением того, что он допускает обновление экрана между кадрами с помощью *updateAfterEvent( )*. Он выполняется, если во время нахождения *mc* на сцене отпущена главная кнопка мыши. Обработчик *MovieClip.onMouseUp( )* оказался бы устаревшим во Flash 6, если бы *Mouse.onMouseUp( )* поддерживал *updateAfterEvent( )*. Подробные сведения можно найти в описании *Mouse.onMouseUp( )*.

Пример

```
// Формат функции обратного вызова
_root.onMouseUp = function () {
  trace("The mouse was released");
}

// Блок onClipEvent() в формате Flash 5 помещается непосредственно в экземпляр клипа
onClipEvent (mouseUp) {
  trace("The mouse was released");
}
```

См. также

Mouse.onMouseUp( ), *MovieClip.onMouseDown( )*, *MovieClip.onMouseMove( )*

MovieClip.onPress(), обработчик события

Flash 6

событие Button; происходит, когда кнопка мыши нажата над клипом

```
mc.onPress()
```

```
on (press) {
 инструкции
}
```

Описание

Обработчик события *onPress( )* идентичен событию *Button.onPress( )* за исключением того, что может также применяться к экземплярам клипов (хотя и не к главным фильмам, таким как `_root`, `_level0` и т. д.).

См. также *Button.onPress( )*; «Применение клипов в качестве кнопок», глава 13

MovieClip.onRelease(), обработчик события

Flash 6

событие Button; происходит, когда кнопка мыши нажата, а затем отпущена над клипом

```
mc.onRelease()
```

```
on (release) {
 инструкции
}
```

Описание

Обработчик события *onRelease( )* идентичен событию *Button.onRelease( )* за исключением того, что может также применяться к экземплярам клипов (хотя и не к главным фильмам, таким как `_root`, `_level0` и т. д.).

См. также

Button.onRelease( ); «Применение клипов в качестве кнопок», глава 13

MovieClip.onReleaseOutside(), обработчик события

Flash 6

событие Button; происходит, когда кнопка мыши нажата над клипом, а затем отпущена за его пределами

```
mc.onReleaseOutside()
```

```
on (releaseOutside) {
  инструкции
}
```

Описание

Обработчик события *onReleaseOutside( )* идентичен событию *Button.onReleaseOutside( )* за исключением того, что может также применяться к экземплярам клипов (хотя и не к главным фильмам, таким как `_root`, `_level0` и т. д.).

См. также

Button.onReleaseOutside( ); «Применение клипов в качестве кнопок», глава 13

MovieClip.onRollOut(), обработчик события

Flash 6

событие Button; происходит, когда курсор мыши уводится с клипа при ненажатой кнопке мыши

```
mc.onRollOut()
```

```
on (rollOut) {
  инструкции
}
```

Описание

Обработчик события *onRollOut( )* идентичен событию *Button.onRollOut( )* за исключением того, что может также применяться к экземплярам клипов (хотя и не к главным фильмам, таким как `_root`, `_level0` и т. д.).

См. также

Button.onRollOut( ); «Применение клипов в качестве кнопок», глава 13

MovieClip.onRollOver(), обработчик события

Flash 6

событие Button; происходит, когда курсор мыши перемещается на клип

```
mc.onRollOver()
```

```
on (rollOver) {
  инструкции
}
```

Описание

Обработчик события *onRollOver( )* идентичен событию *Button.onRollOver( )* за исключением того, что может также применяться к экземплярам клипов (хотя и не к главным фильмам, таким как `_root`, `_level0` и т. д.).

См. также

Button.onRollOver( ); «Применение клипов в качестве кнопок», глава 13

MovieClip.onSetFocus(), обработчик события

Flash 6

функция обратного вызова, выполняемая при получении клипом фокуса

```
mc.onSetFocus(oldFocus)
```

Аргументы

oldFocus

Объект *TextField*, *MovieClip* или *Button*, у которого прежде был фокус, либо `null`, если ни один объект прежде не находился в фокусе.

Описание

Обработчик события *onSetFocus( )* выполняется, когда *mc* получает фокус. Клип получает фокус в следующих случаях:

- Пользователь переходит на клип, нажимая клавишу Tab.
- Фокус назначается программно, при помощи *Selection.setFocus( )*.

Клип может находиться в фокусе при соблюдении одного или нескольких из следующих условий:

- Он определяет хотя бы один обработчик события кнопки (*onPress( )*, *onRollOver( )* и т. д.).
- Его свойство `tabEnabled` имеет значение `true`, и тогда его можно поместить в фокус с помощью клавиши Tab.
- Его свойство `focusEnabled` имеет значение `true`, и тогда его можно поместить в фокус с помощью *Selection.setFocus( )*.

Следующий код регистрирует функцию обратного вызова, отвечающую на *onSetFocus( )*:

```
// Присвоить функцию обратного вызова onSetFocus() с помощью литерала функции
theClip_mc.onSetFocus = function (newFocus) {
  trace(this + " now has focus. Old focus was " + newFocus);
};
```

Обратите внимание, что из тела функции обратного вызова можно обращаться к `theClip_mc` с помощью ключевого слова `this`. Прекратить обработку события *onSetFocus( )* функцией обратного вызова позволяет оператор *delete*, например так:

```
// Не ставьте оператор () после имени функции!
delete clip_mc.onSetFocus;
```

Обычно обработчик *onSetFocus( )* применяется для выделения элемента интерфейса, находящегося в фокусе. Например, темно-синий циферблат можно окрасить в светло-синий цвет и обвести черной окружностью.

Для того чтобы перехватывать все события фокуса централизованно, а не для отдельного экземпляра, применяется *Selection.onSetFocus( )*.

Применение

Обратите внимание, что *onSetFocus( )* нельзя применять в блоке *onClipEvent( )* в стиле Flash 5.

Обработчик *onSetFocus( )* не работает с временными диаграммами главного фильма (например, `_root`, `_level0`, `_level1`), поскольку главные временные диаграммы не могут получать фокус ввода.

См. также

Button.onSetFocus( ), *MovieClip*.`focusEnabled`, *MovieClip.onKillFocus( )*, *MovieClip*.`tabEnabled`, *Selection.onSetFocus( )*, *Selection.setFocus( )*, *TextField.onSetFocus( )*; глава 10

MovieClip.onUnload(), обработчик события

Flash 5; во Flash 6 добавлен формат функции обратного вызова

функция обратного вызова, выполняемая при удалении клипа со сцены или при выполнении unloadMovie()

```
mc.onUnload()

onClipEvent (unload) {
  инструкции
}
```

Описание

Обработчик события *onUnload( )* представляет собой форму старого обработчика события *onClipEvent (unload)* в виде функции обратного вызова и является его более современным аналогом. Обработчик события *onUnload( )* противоположен событию *onLoad( )*; он выполняется, когда *mc* «испускает последний вздох» – сразу вслед за последним кадром, в котором *mc* присутствует на сцене (но *перед* первым кадром, в котором он отсутствует). Событие *onUnload( )* не применяется к главным фильмам (например, `_root`, `_level1` и т. д.).

Событие *onUnload( )* генерируется в следующих случаях:

- Головка воспроизведения достигает конца группы кадров, в которых располагается *mc*.
- *mc* удаляется с помощью функции *removeMovieClip( )* (которая удаляет клипы, созданные функциями *attachMovie( )* и *duplicateMovieClip( )*).
- Для *mc* вызывается функция *unloadMovie( )*.
- Произошла загрузка в *mc* внешнего файла *.swf*.

Последний случай может показаться несколько необычным, но на самом деле это естественное следствие способа загрузки фильмов во Flash. Всякий раз, когда файл *.swf* или *.jpg* загружается в клип, предшествующее содержимое этого клипа удаляется, что запускает событие *onUnload( )*. Вот пример, иллюстрирующий поведение событий *onLoad( )* и *onUnload( )* в связи с *loadMovie( )*:

1. В среде разработки Flash поместим на сцену пустой клип в кадре 1 главной временной диаграммы фильма. Дадим этому клипу имя `emptyClip`.
2. В кадре 5 главной временной диаграммы будем загружать фильм *test.swf* в `emptyClip` с помощью кода:

```
emptyClip.loadMovie("test.swf");
```

3. Запустим фильм с помощью Control → Play movie.

Результаты будут такими:

1. В кадре 1 появляется `emptyClip`, вызывая событие *onLoad( )*.
2. В кадре 5 в два этапа выполняется функция *loadMovie( )*:
   a. Содержимое-заполнитель клипа `emptyClip` удаляется, чтобы освободить место для *test.swf*, что вызывает событие *onUnload( )*.
   b. Загружается *test.swf*, вызывая событие *onLoad( )*.

Событие *onUnload( )* обычно применяется для того, чтобы выполнить служебный код, который может очистить сцену или как-то привести в исходный вид программное окружение. Обработчик *onUnload( )* позволяет также выполнить какое-нибудь действие (например, запустить другой фильм) после того, как закончится клип.

В отличие от своего собрата *onLoad( )*, возможность применения которого в форме функции обратного вызова ограничена, *onUnload( )* действует одинаково и в форме функции обратного вызова, и в блоке *onClipEvent(unLoad)*. Однако при этом сохраняются обычные различия между обработчиками обратного вызова и обработчиками *onClipEvent( )*:

- При выгрузке содержимого *mc* функция обратного вызова *onUnload( )* теряется, а обработчик *onClipEvent(unload)* сохраняется.

- При дублировании *mc* с помощью *duplicateMovieClip( )* функция обратного вызова *onUnload( )* не копируется в дубликат в отличие от обработчика *onClipEvent(unLoad)*.

Применение

Достаточно удивительно, что во Flash 6 *onUnload( )* не генерируется при закрытии Flash Player или при удалении `level0` (главного фильма) из проигрывателя. Возможно, в будущем Macromedia займется этой странностью.

См. также *MovieClip.onLoad( )*, *MovieClip.unloadMovie( )*

MovieClip._parent, свойство

Flash 4

ссылка на клип или фильм, содержащий данный **только для чтения**

mc.`_parent`

Описание

В свойстве `_parent` хранится ссылка на объект клипа, на временной диаграмме которого располагается *mc*. Главные фильмы не поддерживают свойство `_parent`, поскольку находятся на верхнем уровне (на который можно ссылаться как `_root`). Ссылки на `_root._parent` возвращают `undefined`. Свойство `_parent` дает мощную возможность манипулировать клипами относительно текущего клипа.

Ошибки реализации

И во Flash 5, и во Flash 6 можно присвоить свойству `_parent` новое значение, задав другой клип, но смысла в этом мало: меняется только ссылка, а физическая структура клипов остается прежней. Эта непроизвольная особенность будет, вероятно, исправлена.

Пример

Если *mc* находится на главной временной диаграмме фильма, то ее можно воспроизвести из *mc* так:

```
_parent.play();
```

Можно также устанавливать свойства родительской временной диаграммы, например:

```
_parent._alpha = 50;
```

Свойство `_parent` можно также применять последовательно, то есть обращаться к родителю родителя, например:

```
_parent._parent.play();  // Воспроизвести клип, на два клипа выше текущего
```

См. также `_root`; «Ссылка на вложенные экземпляры», глава 13

MovieClip.play(), метод

Flash 5

начинает последовательный вывод кадров клипа или фильма

mc.`play()`

Описание

Метод *MovieClip.play( )* представляет собой альтернативу глобальной функции *play( )*. Вызов метода применяется для управления удаленными клипами или главными фильмами, заданными *mc*.

Общие рекомендации по применению см. в описании глобальной функции *play( )*.

Пример

```
// Начать воспроизведение клипа intro_mc
intro_mc.play();
```

См. также *MovieClip.stop( )*, *play( )*

MovieClip.prevFrame(), метод

Flash 5

перемещает головку воспроизведения клипа или фильма на один кадр назад и останавливает ее

```
mc.prevFrame()
```

Описание

Метод *MovieClip.prevFrame( )* представляет собой альтернативу глобальной функции *prevFrame( )*. Вызов метода применяется для управления удаленными клипами или главными фильмами, заданными *mc*.

Общие рекомендации по применению см. в описании глобальной функции *prevFrame( )*.

Пример

```
// Перемотать slideshow_mc на один кадр назад и остановить
slideshow_mc.prevFrame();
```

См. также *MovieClip.nextFrame( )*, *prevFrame( )*

MovieClip.removeMovieClip(), метод

Flash 5

удаляет клип из Flash Player

```
mc.removeMovieClip()
```

Описание

MovieClip.removeMovieClip( ) представляет собой альтернативу глобальной функции *removeMovieClip( )*. При вызове в качестве метода *MovieClip removeMovieClip( )* не принимает параметр *target*; он всегда удаляет *mc*. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции.

Рекомендации по применению приведены в описании глобальной функции *removeMovieClip( )*.

См. также

duplicateMovieClip( ), *MovieClip.attachMovie( )*, *MovieClip.duplicateMovieClip( )*, *removeMovieClip( )*; «Проблемы, связанные с перекрытием методов и глобальных функций», глава 13

MovieClip._rotation, свойство

Flash 4

угол поворота клипа или фильма в градусах **чтение/запись**

```
mc._rotation
```

Описание

Свойство _rotation, значение которого представляет собой число с плавающей точкой, задает, на сколько градусов *mc* поворачивается относительно первоначального направления (если *mc* – это не главный фильм, то его *первоначальное направление* – это направление символа *mc* в библиотеке). В _rotation отражаются как действия в среде разработки, так и программные изменения. Числа в диапазоне от 0 до 180,0 поворачивают клип по часовой стрелке. Числа в диапазоне от 0 до –180,0 поворачивают клип против часовой стрелки. При повороте клипа на *n* градусов и на *n-360* градусов (где *n* больше нуля) получается одинаковый результат. Например, нет разницы между поворотом клипа на +299,4 градуса и на –60,6 градуса. Аналогично, если *n* отрицательно, то нет разницы между *n* градусами и *n+360* градусами. Например, поворот клипа на –90 градусов ничем не отличается от поворота на +270 градусов.

Если свойству _rotation присваивается значение за пределами диапазона от –180 до 180, то оно приводится к правильному диапазону путем следующего пересчета:

```
x = newRotation % 360;
if (x > 180) {
  x -= 360;
} else if (x < -180) {
  x += 360;
}
_rotation = x;
```

Ошибки реализации

Во Flash Player 4 установка значения _rotation для клипа немного уменьшает масштаб последнего. Если изменить значение _rotation несколько раз, клип заметно сожмется. Для того чтобы учесть эту ошибку, задавайте _xscale и _yscale клипа равными 100 (или тем значениям, которые они имели прежде) при каждом изменении _rotation.

Пример

Следующий обработчик *onEnterFrame( )* заставляет клип box_mc поворачиваться на 5° по часовой стрелке при каждом выводе кадра:

```
box_mc.onEnterFrame = function {
  this._rotation += 5;
}
```

MovieClip.setMask(), метод

Flash 6

назначает клип маской другого клипа

mc.setMask(*maskMovieClip*)

Аргументы

maskMovieClip

Строка, задающая путь к клипу, который должен выступать в качестве маски для *mc*. Поскольку в строковом контексте ссылка на клип преобразуется в путь, *maskMovieClip* может быть и ссылкой на объект клипа, например *mc*.setMask(maskClip) вместо *mc*.setMask("maskClip").

Описание

Метод *setMask( )* назначает *maskMovieClip* маской для *mc*. В результате содержимое *maskMovieClip* становится отверстием, через которое виден *mc*. Отображаются фрагмен-

ты *mc*, находящиеся под занятыми пикселами *maskMovieClip*, а фрагменты, находящиеся под пустыми пикселами *maskMovieClip*, оказываются скрыты. Поэтому обычно маски создают с черными пикселами в тех областях, через которые должен просвечивать маскируемый клип *mc*. Например, *maskMovieClip* может содержать закрашенный черный круг, который будет действовать как прожектор, направленный на содержимое *mc*. Чтобы удалить маску, которая в данный момент применяется к mc, вызовите *setMask( )* с параметром *maskMovieClip*, равным null.

Следующий код устанавливает flashlight_mc в качестве маски darkroom_mc:

```
darkroom_mc.setMask(flashlight_mc);
```

А этот код удаляет маску flashlight_mc:

```
darkroom_mc.setMask(null);
```

В масках могут использоваться фигуры, вычерченные с помощью методов черчения в клипе (*beginFill( )*, *lineTo( )*, *curveTo( )* и т. д.). Начиная с Flash 6 временные диаграммы клипов, применяемых в качестве масок, воспроизводятся нормально. Содержимое каждого кадра, когда оно воспроизводится, становится маской. Клипы масок можно также перетаскивать, масштабировать и перемещать программно. У них могут быть обработчики событий клипов и кнопок. Например, клип маски может определять обработчик *onEnterFrame( )*, изменяющий его положение в каждом новом кадре. Либо в клипе маски можно определить обработчики *onPress( )* и *onRelease( )*, которые осуществляют его перетаскивание, например:

```
darkroom_mc.setMask("spotlight_mc");

spotlight_mc.onPress = function () {
  trace("Dragging spotlight");
  this.startDrag();
}

spotlight_mc.onRelease = spotlight_mc.onReleaseOutside = function () {
  trace("Dropping spotlight");
  this.stopDrag();
}
```

Если *maskMovieClip* определяет хотя бы один обработчик события кнопки или содержит объект кнопки, то никакие события кнопок не достигают *mc*. Это означает, что маскированный клип не может реагировать на события кнопок, если в его маске определено событие кнопки или содержится кнопка.

Если применить маску, создаваемую на этапе исполнения, к клипу, использованному в маскирующем слое этапа разработки, Flash удалит маску этапа разработки и поместит вновь замаскированный клип ниже уровней, замаскированных на этапе разработки.

Применение

На маски Flash распространяется ряд ограничений:

- В версиях Flash Player до 6.0.40.0 нельзя маскировать клипом текстовые поля без встроенных шрифтов. Во Flash Player 6.0.40.0 прямоугольная граничная рамка клипа маски, создаваемой на этапе исполнения, может маскировать текстовые поля, даже если их шрифты не являются встроенными. Для маскирования текстовых полей в более старых версиях Player требуется сделать их шрифты встроенными.
- Текстовые поля без встроенных шрифтов нельзя применять в качестве масок.
- Текстовые поля без встроенных шрифтов нельзя маскировать с помощью маскирующего слоя этапа разработки.

- Линии не могут применяться в качестве масок. В клипах масок следует использовать заливки вместо линий. Преобразовать линии в заливки можно с помощью Modify → Shape → Convert Lines to Fills.
- Метод *setMask( )* не может назначить один клип в качестве маски для нескольких клипов. Для того чтобы применить одинаковую маску к нескольким клипам, создайте необходимое количество копий клипа-маски или воспользуйтесь маской этапа разработки, которую можно применять одновременно к нескольким клипам.
- Во Flash 6 внутренние отверстия в фигурах, находящихся в клипах, не отражаются в масках. Так, нельзя с помощью клипа создать маску в виде бублика (дырка будет заполнена, и маска окажется сплошным кругом). Однако с помощью элементарных фигур этапа разработки в маскирующем слое можно создавать фигуры масок с внутренними дырками. Метод *beginFill( )* тоже позволяет вычерчивать фигуры с дырками, правильно функционирующие в качестве масок.

См. также *MovieClip.beginFill( )*

MovieClip.startDrag(), метод

Flash 5

заставляет фильм или клип следовать за курсором мыши

```
mc.startDrag()
mc.startDrag(lockCenter)
mc.startDrag(lockCenter, left, top, right, bottom)
```

Аргументы

lockCenter Булево значение, определяющее, должна ли точка регистрации `mc` быть размещена в центре курсора мыши (`true`) или она должна перемещаться относительно своего первоначального положения (`false`).

left Число, задающее координату x, левее которой нельзя перетаскивать точку регистрации `mc`.

top Число, задающее координату y, выше которой нельзя перетаскивать точку регистрации `mc`.

right Число, задающее координату x, правее которой нельзя перетаскивать точку регистрации `mc`.

bottom Число, задающее координату y, ниже которой нельзя перетаскивать точку регистрации `mc`.

Описание

Метод *MovieClip.startDrag( )* представляет собой альтернативу глобальной функции *startDrag( )*. При вызове в качестве метода *MovieClip startDrag( )* не принимает параметр `target`; он буксирует `mc`. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции.

Рекомендации по применению приведены в описании глобальной функции *startDrag( )*.

Ошибки

Обратите внимание, что правильный порядок координат ограничивающего прямоугольника таков: левая, верхняя, правая, нижняя, но в ActionScript Dictionary из Flash 5 они перечислены для *MovieClip.startDrag( )* в неправильном порядке.

Пример

```
// Код кнопки для перетаскивания текущего клипа или фильма
this.onPress = function () {
  this.startDrag(true);
};

this.onRelease = function () {
  stopDrag();
}
```

См. также

MovieClip.stopDrag( ), *startDrag( )*, *stopDrag( )*; «Проблемы, связанные с перекрытием методов и глобальных функций», глава 13

MovieClip.stop(), метод

Flash 5

сделать паузу в клипе или фильме

mc.stop()

Описание

Метод *MovieClip.stop( )* представляет собой альтернативу глобальной функции *stop( )* и применяется для управления удаленными клипами или главными фильмами, заданными *mc*. Flash Player продолжает воспроизводить содержимое сцены и обрабатывать события, даже когда головка воспроизведения остановлена.

Общие рекомендации по применению см. в описании глобальной функции *stop( )*.

Пример

```
// Остановить воспроизведение spinner_mc
spinner_mc.stop();
```

См. также

MovieClip.nextFrame( ), *MovieClip.play( )*, *MovieClip.prevFrame( )*, *stop( )*

MovieClip.stopDrag(), метод

Flash 5

заканчивает текущую операцию буксировки

mc.stopDrag()

Описание

Метод *MovieClip.stopDrag( )* представляет собой альтернативу глобальной функции *stopDrag( )*. Однако необходимости в применении формата метода нет, так как *stopDrag( )* прекращает любую текущую операцию буксировки, будь она вызвана через клип или через глобальную функцию.

См. также

MovieClip.startDrag( ), *stopDrag( )*

MovieClip.swapDepths(), метод

Flash 5

изменяет графический слой экземпляра в стеке визуального содержимого

mc.swapDepths(*target*)
mc.swapDepths(*depth*)

Аргументы

target

Строка, указывающая путь к клипу, который надо поменять местами с `mc`. Поскольку в строковом контексте ссылки на клипы преобразуются в пути, `target` может также являться ссылкой на объект клипа, например `mc.swapDepths(window2)` вместо `mc.swapDepths("window2")`.

depth

Целое число от –16 384 до 1 048 575 включительно, задающее уровень в клипе стеков родительского объекта `mc`. Клипы с большей глубиной закрывают клипы с меньшей глубиной, поэтому клип с глубиной 6 находится перед клипом с глубиной 5, а он, в свою очередь, перед клипом с глубиной 4 и т. д. Глубины, превосходящие по модулю –1, зарезервированы для динамически создаваемого содержимого. Глубины от –16 383 до –1 включительно зарезервированы для содержимого, создаваемого на этапе разработки, а глубина –16 384 зарезервирована для динамического содержимого, которое должно оказаться в `mc` ниже любого содержимого среды разработки (например, для фона этапа исполнения). См. важные подробности в разделе главы 13 «Добавление в стек клипов, созданных с помощью attachMovie()».

Описание

Все экземпляры клипов в документе *.swf* располагаются в стеке, управляющем зрительным размещением клипов по слоям в проигрывателе. Стек имеет структуру колоды карт: клипы с более высоким положением в стеке оказываются перед клипами с более низким положением. Положение экземпляра в стеке первоначально устанавливается при создании клипа (в среде разработки Flash или посредством *attachMovie( )*, *duplicateMovieClip( )* или *createEmptyMovieClip( )*). Метод *swapDepths( )* позволяет изменять положение экземпляра в стеке.

Метод *swapDepths( )* имеет два формата. Если аргумент `target` задан, то метод *swapDepths( )* обменивает позиции в стеке `mc` и `target` при условии, что родительский клип для `mc` и `target` один и тот же (то есть они находятся на одной временной диаграмме). Если задан аргумент `depth`, то метод *swapDepths( )* помещает `mc` в новую позицию в стеке родителя `mc`. Если эта позиция занята, то ее владелец перемещается в прежнюю позицию `mc`. Обратите внимание, что с помощью *swapDepths( )* нельзя переместить клип за пределы стека, поддерживаемого его родителем. Сведения о расположении фильмов и клипов по стекам в проигрывателе можно найти в разделе «Порядок расположения фильмов и экземпляров в стеке» главы 13. Узнать текущую глубину клипа можно с помощью *MovieClip.getDepth( )*.

Ошибки реализации

Если задан аргумент `depth`, то перемена мест между дублированным или прикрепленным экземпляром, с одной стороны, и созданным вручную экземпляром, с другой, может вызвать во Flash 5 Player проблемы с перерисовкой. Осторожно обращайтесь с аргументом `depth` и всегда тщательно тестируйте код, обменивающий уровни.

См. также

MovieClip.attachMovie( ), *MovieClip.createEmptyMovieClip( )*, *MovieClip.createTextField( )*, *MovieClip.duplicateMovieClip( )*, *MovieClip.getDepth( )*; «Порядок расположения фильмов и экземпляров в стеке», глава 13

MovieClip.tabChildren, свойство

Flash 6

управляет включением вложенных объектов в автоматический обход при табуляции **чтение/запись**

mc.tabChildren

Описание

Булево свойство tabChildren устанавливает, должны ли текстовые поля и кнопки внутри *mc* включаться в автоматический порядок обхода при нажатии клавиши Tab. Оно применимо только к клипам, но не к главным фильмам (например, _root, _level1). Если tabChildren имеет значение true (по умолчанию), то все кнопки, клипы с обработчиками кнопок и текстовые поля, содержащиеся в *mc*, автоматически включаются в автоматический обход по нажатию клавиши Tab, даже если mc.tabEnabled имеет значение false. Если tabChildren имеет значение false, то содержащиеся в *mc* объекты не доступны по клавише Tab.

У свойства tabChildren есть два типичных применения:

- С его помощью удобно отключать табуляцию для целой группы объектов. Например, можно отключить группу кнопок панели инструментов, поместив кнопки внутрь клипа и присвоив свойству tabChildren этого клипа значение false.
- Оно позволяет избежать помех взаимодействию клавиатуры с клипом со стороны содержимого клипа. Например, чтобы клавиша Tab не захватывала ручку внутри панели ползунка, установим для панели ползунка tabEnabled равным true и tabChildren равным false. Благодаря этому вся панель учитывается в порядке обхода по клавише Tab как единая позиция.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

Пример

Следующий код разрешает sliderBar_mc получать фокус по нажатию клавиши Tab, но запрещает установку фокуса на любые элементы его содержимого:

```
sliderBar_mc.tabEnabled = true;
sliderBar_mc.tabChildren = false;
```

См. также *MovieClip*.tabEnabled, *MovieClip*.tabIndex

MovieClip.tabEnabled, свойство

Flash 6

включает или исключает клип из текущего порядка обхода при табуляции **чтение/запись**

mc.tabEnabled

Описание

Булево свойство tabEnabled управляет вхождением *mc* в порядок обхода по клавише табуляции для текстовых полей, клипов и кнопок, находящихся в данный момент на экране. Порядок табуляции – это последовательность, в которой объекты получают фокус при нажатии в фильме клавиши Tab. Подробное описание см. в *TextField*.tabEnabled.

Для клипов свойство tabEnabled может:

- Исключать клип с определенным свойством tabIndex из пользовательского порядка табуляции
- Исключать любой клип с событиями кнопок из автоматического порядка табуляции
- Включать клип в автоматический порядок табуляции (по умолчанию автоматический порядок табуляции не включает клипы с событиями кнопок)

По умолчанию tabEnabled имеет значение undefined, и *mc* исключается из автоматического порядка табуляции, если в нем не определен хотя бы один обработчик события кнопки (*onPress( )*, *onRelease( )* и т. д.), и тогда он включается в автоматический порядок табуляции.

Клип, в котором не определены обработчики событий кнопок, обычно добавляется в автоматический или пользовательский порядок табуляции, если он является элементом интерфейса с доступом с клавиатуры. Так, клип окна списка должен иметь возможность получать фокус с помощью клавиши Tab и управляться клавишами со стрелками, поэтому его свойству tabEnabled надо присвоить значение true.

По умолчанию, когда клип находится в фокусе, Flash отображает вокруг него желтую прямоугольную рамку. Можно организовать собственный способ выделения, для чего сначала надо отключить желтый прямоугольник, присвоив *MovieClip.*_focusrect значение false, а затем либо создать выделенное состояние внутри обработчика *onSetFocus( )* для клипа, либо определить в клипе кадры _up и _over для состояний вне фокуса и в фокусе соответственно. (Flash автоматически воспроизводит эти специальные кадры, когда клип приобретает или теряет фокус.)

Для того чтобы разрешить помещение клипа в фокус программным образом с помощью *Selection.setFocus( )*, мы должны присвоить его свойству focusEnabled значение true. Обратите внимание, что свойство tabEnabled применимо только к клавише Tab, а свойство focusEnabled применимо только к *Selection.setFocus( )*. Для многих графических элементов интерфейса уместно отключение автоматической табуляции для всех дочерних объектов клипа (содержащихся в нем вложенных клипов, кнопок и текстовых полей). Это легко сделать, присвоив свойству tabChildren значение false. Отключение автоматической табуляции дочерних объектов позволяет ясно управлять взаимодействием клипа с клавиатурой, не беспокоясь о вмешательстве каких-либо подчиненных элементов интерфейса, которые могут содержаться в клипе.

В общем, чтобы разрешить получение фокуса клипом – графическим элементом интерфейса, надо объединить следующие настройки свойств:

```
// Отключить автоматический желтый прямоугольник фокуса
theClip_mc._focusrect = false;
// Разрешить помещение клипа в фокус клавишей Tab
theClip_mc.tabEnabled = true;
// Разрешить помещение клипа в фокус с помощью Selection.setFocus()
theClip_mc.focusEnabled = true;
// Запретить автоматическую табуляцию клипов, текстовых полей и кнопок внутри клипа
theClip_mc.tabChildren = false;
```

Обратите внимание, что если свойство клипа tabEnabled имеет значение false, то для любых содержащихся в нем текстовых полей, кнопок или клипов табуляция может быть включена. Чтобы запретить табуляцию для клипа и всех содержащихся в нем объектов, надо установить оба свойства tabEnabled и tabChildren равными false.

У клипов с поведением кнопок (в которых определены обработчики событий) свойство tabEnabled обычно устанавливается равным false, когда они отключены (то есть их свойство enabled имеет значение false) или когда доступ к ним осуществляется только с помощью мыши, например в играх, когда для обнаружения действий мыши применяются прозрачные кнопки.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

См. также

MovieClip.enabled, *MovieClip*.focusEnabled, ***MovieClip.onSetFocus( )***, *MovieClip*.tabChildren, *MovieClip*.tabIndex, ***Selection.setFocus( )***, *TextField*.tabEnabled

MovieClip.tabIndex, свойство

Flash 6

задает индекс клипа в пользовательском порядке табуляции **чтение/запись**

mc.tabIndex

Описание

Свойство tabIndex представляет собой целое число, задающее место *mc* в пользовательском порядке табуляции. Если нет текстовых полей, клипов или кнопок, для которых определено свойство tabIndex, Flash определяет порядок табуляции автоматически и при нажатии клавиши Tab перемещает фокус между клипами с обработчиками событий кнопок, кнопками и текстовыми полями справа налево, а затем сверху вниз. Свойство tabIndex применяется для того, чтобы перемещение фокуса при нажатии клавиши Tab происходило в порядке, заданном пользователем и отменяющем автоматический. Подробное описание см. в *TextField*.tabIndex.

Объекты со свойством _visible, равным false, не включаются в порядок табуляции. Если свойство tabIndex задано хотя бы для одного объекта, то ни к одному объекту, свойству tabIndex которого не присвоено значение, нельзя обратиться с помощью клавиши Tab.

По умолчанию клипы не включаются в порядок табуляции, если только в них не определены обработчики событий кнопок. Для того чтобы включить такой клип в автоматический порядок табуляции, присвойте его свойству tabEnabled значение true. Клип включается в пользовательский порядок табуляции, если задать для его свойства tabIndex значение, равное целому числу (задавать значение свойства tabEnabled при этом не требуется). Клип, в котором не определены обработчики событий кнопок, обычно добавляется в автоматический или пользовательский порядок табуляции, если он является элементом интерфейса с доступом с клавиатуры (и тогда управляется клавиатурой с помощью *Key.onKeyUp( )* или *MovieClip.onKeyUp( )*).

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

Пример

Следующий код ограничивает доступ клавиши Tab к клипу окна списка, текстовому полю и кнопке Submit:

```
phoneListBox_mc.tabIndex = 1;
firstName_txt.tabIndex = 2;
submit_btn.tabIndex = 3;
```

См. также *MovieClip*.tabChildren, *MovieClip*.tabEnabled, *TextField*.tabIndex

MovieClip._target, свойство

Flash 4

путь к клипу или фильму с использованием слэш-синтаксиса **только для чтения**

mc._target

Описание

Свойство _target представляет путь к *mc* как строку в слэш-нотации в стиле Flash 4. Например, если клип ball располагается на главной временной диаграмме фильма, то свойство _target для ball имеет значение "/ball". Для клипа stripe внутри клипа ball свойство _target будет иметь значение "/ball/stripe".

Строка, содержащая путь к клипу в точечной нотации, возвращается функцией *targetPath( )*. Строка со слэш-нотацией преобразуется в объектную ссылку функцией *eval( )*.

См. также *eval( )*, *targetPath( )*

MovieClip._totalframes, свойство

Flash 4

суммарное количество кадров на временной диаграмме клипа или фильма **только для чтения**

mc._totalframes

Описание

Целочисленное свойство _totalframes представляет количество кадров на временной диаграмме *mc*, где *mc* – это клип или главный фильм. Свойство _totalframes чаще всего применяется вместе с _frameloaded для создания предзагрузчиков, как показано в описании *MovieClip*._framesloaded.

Функции *loadMovie( )* и *unloadMovie( )* выполняются асинхронно, поэтому значение _framesloaded зависит от времени вызова и значения mc:

- Если *mc* имеет значение undefined, _totalframes возвращает undefined.
- После обнаружения в исходном коде команд *loadMovie( )* или *unloadMovie( )*, но до того, как они будут выполнены или будет определено количество кадров во внешнем файле, _totalframes возвращает количество кадров в прежнем *mc* (существовавшем до команд *loadMovie( )* или *unloadMovie( )*).
- После того как определено количество кадров, _totalframes возвращает количество кадров в загружающемся фильме, если только файл на *url* найден.
- Если файл на *url* не найден и *mc* представляет собой клип, _totalframes возвращает 0, а _framesloaded возвращает –1. Однако если *mc* – это документ _level, то _framesloaded и _ totalframes возвращают undefined.

- После выполнения команды *unloadMovie( )* `_totalframes` возвращает 0, если *mc* — это клип, и `undefined`, если `mc` является уровнем документа, например `_level0`.

См. также *MovieClip.*`_framesloaded`

MovieClip.trackAsMenu, свойство

Flash 6

модифицирует требования к обработчику onRelease(), включая поведение в стиле меню **чтение/запись**

mc.`trackAsMenu`

Описание

Булево свойство `trackAsMenu` модифицирует условия, вызывающие выполнение для *mc* обработчика события *onRelease( )*. Обычно применяется для того, чтобы превратить группу кнопок в меню, и только для клипов, определяющих хотя бы один обработчик события кнопки. Подробное описание см. в *Button*.`trackAsMenu`.

См. также *Button*.`trackAsMenu`, *MovieClip.onRelease( )*

MovieClip.unloadMovie(), метод

Flash 5

удаляет фильм или клип из проигрывателя

mc.`unloadMovie()`

Описание

Метод *MovieClip.unloadMovie( )* представляетс собой альтернативу глобальной функции *unloadMovie( )*. При вызове в качестве метода *MovieClip*, *unloadMovie( )* не принимает параметр *target*: он всегда выгружает *mc*. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции, поскольку не нужен параметр *target*.

Рекомендации по применению можно найти в описании глобальной функции *unloadMovie( )*.

Пример

```
// Удаляет загруженный документ с уровня 1
_level1.unloadMovie();
```

См. также

MovieClip.loadMovie( ), *unloadMovie( )*; «Порядок расположения фильмов и экземпляров в стеке», глава 13

MovieClip._url, свойство

Flash 4

сетевой адрес, с которого был загружен клип или фильм **только для чтения**

mc.`_url`

Описание

Свойство `_url` имеет строковый тип и представляет URL (унифицированный указатель информационного ресурса), задающий адрес в Интернете или на диске, с которо-

го было загружено содержимое *mc*. Свойство _url всегда содержит абсолютный, а не относительный URL. Для главных фильмов _url просто представляет адрес текущего файла *.swf*. Для всех клипов файла *.swf* свойство _url совпадает с главным фильмом этого файла, если только в отдельные клипы не загружались внешние файлы *.swf* посредством *MovieClip.loadMovie( )*. Свойство _url для клипа, хранящего внешний загруженный файл *.swf*, содержит адрес внешнего загруженного файла.

Иногда свойство _url применяется для создания простых систем защиты, предотвращающих воспроизведение фильмов в нежелательном месте.

Пример

Значение _url в фильме, загруженном с веб-сайта, выглядит так:

```
"http://www.moock.org/gwen/meetgwen.swf"
```

Значение _url в фильме, загруженном с жесткого диска локальной машины, выглядит так:

```
"file:///C|/data/flashfiles/movie.swf"
```

Ниже приведена проверка, позволяющая убедиться, что фильм загружен из указанного места в сети (если нет, выводится кадр с сообщением об ошибке):

```
if (this._url != "http://www.moock.org/gwen/meetgwen.swf") {
  trace ("This movie is not running from its intended location.";
  this.gotoAndStop("accessDenied");
}
```

См. также *MovieClip.loadMovie( )*, *System.Security.allowDomain( )*

MovieClip.useHandCursor, свойство

Flash 6

управляет показом курсора мыши в виде ладони при нахождении над клипом (только для клипов в режиме кнопок) **чтение/запись**

mc.useHandCursor

Описание

Если булево свойство useHandCursor имеет значение false, то отключается отображение курсора в виде ладони, форму которой он обычно имеет, находясь над активной областью *mc*. Это относится только к клипам с обработчиками кнопок. Подробности см. в описании *Button*.useHandCursor.

Пример

Следующий код отключает отображение курсора в виде ладони над checkBox_mc:

```
checkBox_mc.useHandCursor = false;
```

А этот – отображение курсора в виде ладони над всеми клипами в фильме:

```
MovieClip.prototype.useHandCursor = false;
```

См. также *Button*.useHandCursor, *Mouse.hide( )*

MovieClip.valueOf(), метод

Flash 5

возвращает ссылку на клип или фильм

mc.valueOf()

Возвращаемое значение

Объектная ссылка на *mc*.

Описание

Метод *valueOf( )* возвращает объектную ссылку на *mc*. Ссылка синонимична собственно *mc*, поэтому необходимость в явном вызове *valueOf( )* возникает редко.

См. также *Object.valueOf( )*, *targetPath( )*

MovieClip._visible, свойство

Flash 4

определяет, виден или скрыт клип или фильм **чтение/запись**

mc._visible

Описание

Булево свойство _visible указывает, что в данный момент *mc* виден (true) или скрыт (false). Свойство _visible применяется как средство, позволяющее быстро скрыть из видимости клип или фильм. Обратите внимание, что события кнопок и свойства-обработчики событий *onKeyUp( )* и *onKeyDown( )* не действуют для невидимых клипов, но в остальном они ведут себя, как обычные клипы. Ими можно по-прежнему управлять средствами ActionScript, и они по-прежнему могут воспроизводиться, останавливаться, быть обнаруженными с помощью *hitTest( )*, а также получать другие события клипов. Скрытые клипы просто не выводятся на экран.

Начальным значением свойства _visible для всех клипов является true, даже для полностью прозрачных или находящихся за пределами сцены. Свойство _visible изменяется, только если специально модифицируется сценарием. Его следует рассматривать как программный способ вывода на экран и сокрытия клипов, а не как воплощение всех факторов, влияющих на видимость клипа, таких как положение на сцене и прозрачность.

Для того чтобы сделать клип невидимым, предпочтительнее воспользоваться свойством _visible, чем делать его полностью прозрачным или удалять со сцены, потому что Flash Player не пытается рисовать графику в клипах, у которых _visible имеет значение false, что увеличивает производительность вывода изображений.

Пример

Следующие обработчики событий скрывают текущий клип при нажатии кнопки мыши и показывают его вновь, когда кнопка отпущена:

```
this.onMouseDown = function () {
  this._visible = false;
}

this.onMouseUp = function () {
  this._visible = true;
}
```

См. также *MovieClip*._alpha

MovieClip._width, свойство

Flash 4; усилено во Flash 5

возвращает ширину клипа или фильма в пикселах

только для чтения во Flash 4; чтение/запись начиная с Flash 5

mc._width

Описание

Свойство _width, имеющее тип числа с плавающей точкой, представляет собой неотрицательное число, указывающее текущую шрину *mc* в пикселах. Если *mc* не имеет содержимого, то _width равно 0. Свойство _width измеряет содержимое клипа как расстояние от самого левого занятого пиксела до самого правого занятого пиксела, даже если между этими пикселами есть пустое пространство. *Занятый пиксел* – это пиксел, представляющий собой часть фигуры, изображения, кнопки, клипа, текстового поля или другого элемента содержимого. Свойство _width отражает изменения, внесенные в клип в среде разработки или через _xscale.

Присваивание значения свойству _width изменяет размер клипа по горизонтали. Попытки присвоить _width отрицательное значение игнорируются. Присваивание _width значения 0 не делает его невидимым, а превращает в вертикальную линию толщиной в один пиксел. Во Flash базовой единицей измерения является одна двадцатая часть пиксела (twip), поэтому минимально допустимое изменение _width составляет 0,05 пиксела. Меньшие изменения игнорируются:

```
trace(ball_mc._width);    // Выводит: 100
ball_mc._width += .01;
trace(ball_mc._width);    // По-прежнему выводит 100. Присваивание не имело результата.
```

Значение _width для главного фильма (то есть root._width или _level*n*._width) – это не высота сцены, указанная в Modify → Document → Dimensions (или Modify → Movie → Dimensions во Flash 5) в среде разработки, а ширина содержимого главного фильма. Во Flash 6 заданные для фильма размеры можно получить, прочитав свойства Stage.height и Stage.width.

Для фильма, находящегося в процессе загрузки, свойство _width возвращает undefined, пока фильм не будет загружен полностью.

Применение

Обратите внимание, что при изменении ширины клипа пропорционально масштабируются линии, утрачивая первоначальную толщину, установленную в панели Stroke. Но если в панели Stroke для линии установлена толщина Hairline, то при изменении размеров клипа этот параметр не изменяется. Поэтому, во избежание утолщения или других искажений штрихов при изменении масштаба клипа, следует задавать для линий толщину Hairline.

Пример

```
ball_mc._width = 20;          // Установить ширину ball_mc равной 20 пикселам
ball_mc._width *= 2;          // Удвоить ширину ball_mc
```

См. также *MovieClip*._height, *MovieClip*._xscale, Stage.width

MovieClip._x, свойство

Flash 4

возвращает горизонтальное положение клипа или фильма в пикселах

чтение/запись

mc._x

Описание

Свойство _x, представляемое числом с плавающей точкой, всегда указывает на положение *точки регистрации* *mc* по горизонтали, но измеряется относительно одного из трех возможных пространств координат:

- Если *mc* находится на главной временной диаграмме, _x измеряется относительно левого края *сцены* (*stage*). Так, _x, равное 20, указывает, что точка регистрации *mc* находится на 20 пикселов правее левого края сцены, а –20 обозначает 20 пикселов влево.
- Если *mc* располагается на временной диаграмме другого клипа, то _x измеряется относительно *точки регистрации* этого родительского экземпляра. Например, _x, равное 20, указывает, что точка регистрации *mc* находится на 20 пикселов правее точки регистрации родительского экземпляра, а –20 обозначает 20 пикселов влево.
- Если *mc* является главным фильмом, то _x представляет смещение по горизонтали всего документа *.swf* относительно левого края сцены. Например, _x, равное 200, указывает, что содержимое сцены смещено на 200 пикселов вправо от позиции этапа разработки; –200 обозначает 200 пикселов влево.

Свойство _x (и все горизонтальные координаты во Flash) возрастает в правую сторону и уменьшается в левую. Дробные значения _x аппроксимируются во Flash с помощью сглаживания. Базовая единица измерения во Flash – одна двадцатая часть пиксела (twip), поэтому минимально допустимое перемещение клипа составляет 0,05 пиксела. Меньшие изменения игнорируются:

```
trace(ball_mc._x);    // Выводит: 0
ball_mc._x += .01;
trace(ball_mc._x);    // По-прежнему выводит 0. Присваивание не имело результата.
```

Если *mc* содержится в масштабированном и/или повернутом экземпляре, то пространство координат, в котором он находится, тоже масштабировано и/или повернуто. Так, если родитель *mc* имеет масштаб 200% и повернут на 90° по часовой стрелке, то _x будет возрастать вниз, а не вправо, и одна единица измерения _x будет фактически равна 2 пикселам вместо 1.

Поскольку переключение между пространствами координат экземпляра и главного фильма может быть утомительным, объект *MovieClip* предоставляет для преобразований пространств координат методы *localToGlobal( )* и *globalToLocal( )*.

Пример

Размещение в клипе следующего обработчика *onEnterFrame( )* вызывает его перемещение вправо на 5 пикселов в каждом новом кадре (если считать, что его пространство координат не изменилось в результате преобразований, произведенных над его родителем):

```
ball_mc.onEnterFrame = function () {
  this._x += 5;
}
```

Расположение клипов с помощью _x и _y является базовой задачей визуального программирования на ActionScript. В следующем примере ко всем клипам добавляется метод *follow( )*, вызывающий перемещение клипа за курсором мыши с фиксированной скоростью, заданной в пользовательском свойстве speed (в сетевом хранилище кода можно найти много других примеров перемещений):

```
// Метод: MovieClip.follow()
// Перемещает клип в сторону точки (leaderX, leaderY)
```

```
MovieClip.prototype.follow = function (leaderX, leaderY) {
  // Перемещаться, только если не достигнута точка назначения
  if (this._x != leaderX || this._y != leaderY) {
    // Определить расстояние между clip и leader
    var deltaX = this._x - leaderX;
    var deltaY = this._y - leaderY;
    var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));

    // Разложить speed по осям X и Y
    var moveX = this.speed * (deltaX / dist);
    var moveY = this.speed * (deltaY / dist);

      // Если у клипа достаточно скорости, чтобы перескочить точку назначения, просто
      //  перейти в точку назначения. Иначе переместиться в соответствии со скоростью клипа.
    if (this.speed >= dist) {
      this._x = leaderX;
      this._y = leaderY;
    } else {
      this._x -= moveX;
      this._y -= moveY;
    }
  }
}

// Применение...
butterfly_mc.speed = 10;
butterfly_mc.onEnterFrame = function () {
  this.follow(_root._xmouse, _root._ymouse);
}
```

См. также

MovieClip.globalToLocal( ), *MovieClip.localToGlobal( )*, *MovieClip.*_xscale, *MovieClip.*_y

MovieClip._xmouse, свойство

Flash 5

горизонтальное расположение курсора мыши **только для чтения**

*mc.*_xmouse

Описание

Свойство _xmouse, представляемое числом с плавающей точкой, указывает горизонтальное расположение активной точки курсора мыши относительно пространства координат *mc*. Если *mc* — это главный фильм, то _xmouse измеряется от левого края сцены. Если же *mc* представляет собой экземпляр, то _xmouse измеряется от точки регистрации экземпляра. Для получения согласованных значений координаты _xmouse, всегда измеряемой относительно сцены, следует применять _root._xmouse.

Пример

Размещение следующего кода в клипе заставляет последний отражать горизонтальное положение курсора мыши относительно сцены (перемещаясь влево и вправо по прямой линии):

```
this.onEnterFrame = function () {
  this._x = _level0._xmouse;
}
```

См. также

*MovieClip.*_ymouse

MovieClip._xscale, свойство

Flash 4

возвращает ширину клипа или фильма в процентах **чтение/запись**

mc._xscale

Описание

Свойство _xscale, представляемое числом с плавающей точкой, указывает ширину *mc* по отношению к исходной ширине (в процентах). Если *mc* – это экземпляр, то его «исходная ширина» равна ширине символа экземпляра в библиотеке. Если же *mc* представляет собою главный фильм, то его «исходная ширина» равна ширине фильма на этапе разработки.

Если текущая ширина *mc* совпадает с исходной, то значение _xscale равно 100. А если _xscale равно 200, значит, ширина *mc* удвоена по сравнению с исходной. При _xscale, равном 50, ширина *mc* вдвое меньше исходной.

Свойство _xscale масштабирует клип относительно точки регистрации пропорционально влево и вправо от нее. Чтобы размер клипа изменялся только в одну сторону, разместите все его содержимое по одну сторону от точки регистрации символа клипа (этот прием полезен при создании горизонтального указателя в предварительном загрузчике). Если присвоить свойству клипа _xscale отрицательное значение, клип переворачивается по горизонтали, как если бы через точку регистрации проходило вертикальное зеркало (то есть становится зеркальным отражением самого себя), после чего с отрицательным значением обращаются, как с положительным. Для горизонтального переворота клипа без изменения его размеров задайте значение свойства _xscale, равным -_xscale.

Пример

```
// Удвоить ширину ball_mc (не изменяя высоты)
ball_mc._xscale *= 2;

// Создать зеркальное отражение ball_mc
ball_mc._xscale = -ball_mc._xscale;
```

См. также

MovieClip._yscale

MovieClip._y, свойство

Flash 4

возвращает вертикальное положение клипа или фильма в пикселах **чтение/запись**

mc._y

Описание

Свойство _y, представляемое числом с плавающей точкой, всегда указывает на положение *точки регистрации* *mc* по вертикали, но измеряется относительно одного из трех возможных пространств координат:

- Если *mc* находится на главной временной диаграмме, то _y измеряется относительно верхнего края *сцены*. Так, _y, равное 20, указывает, что точка регистрации *mc* находится на 20 пикселов ниже верхнего края сцены, а –20 обозначает 20 пикселов вверх.
- Если *mc* располагается на временной диаграмме другого экземпляра клипа, то _y измеряется относительно *точки регистрации* этого родительского экземпляра. Например, _y, равное 20, указывает, что точка регистрации *mc* находится на

20 пикселов ниже точки регистрации родительского экземпляра, а –20 обозначает 20 пикселов вверх.

- Если *mc* – это главный фильм, то _y представляет смещение по вертикали всего документа *.swf* относительно верхнего края сцены. Так, _y, равное 200, указывает, что содержимое сцены смещено на 200 вниз от позиции во время редактирования, а –200 обозначает 200 пикселов вверх.

Свойство _y (и все вертикальные координаты во Flash) возрастает вниз и уменьшается вверх – противоположно декартовой системе координат. Дробные значения _y аппроксимируются во Flash с помощью сглаживания. Базовая единица измерения во Flash – одна двадцатая часть пиксела (twip), поэтому минимально допустимое перемещение клипа составляет 0,05 пиксела. Меньшие изменения игнорируются:

```
trace(ball_mc._y);    // Выводит: 0
ball_mc._y += .01;
trace(ball_mc._y);    // По-прежнему выводит 0. Присваивание не имело результата.
```

Если *mc* содержится в масштабированном и/или повернутом экземпляре, то пространство координат, в котором он находится, тоже масштабировано и/или повернуто. Так, если родитель *mc* имеет масштаб 200% и повернут на 90° по часовой стрелке, то _y будет возрастать влево, а не вниз, а одна единица измерения _y будет фактически равна 2 пикселам вместо 1.

Поскольку переключение между пространствами координат экземпляра и главного фильма может быть утомительным, объект *MovieClip* предоставляет для преобразований пространств координат методы *localToGlobal( )* и *globalToLocal( )*.

Пример

Следующий обработчик *onEnterFrame( )* вызывает перемещение клипа вниз на 5 пикселов в каждом новом кадре (если считать, что его пространство координат не изменилось в результате преобразований, произведенных над его родителем):

```
ball_mc.onEnterFrame = function () {
  this._y += 5;
}
```

См. также *MovieClip.globalToLocal( )*, *MovieClip.localToGlobal( )*, *MovieClip._x*

MovieClip._ymouse, свойство

Flash 5

возвращает вертикальную координату курсора мыши **только для чтения**

mc._ymouse

Описание

Свойство _ymouse, представляемое числом с плавающей точкой, указывает вертикальное расположение курсора мыши относительно пространства координат *mc*. Если *mc* – это главный фильм, то _ymouse измеряется от верхнего края сцены. Если *mc* – экземпляр, то _ymouse измеряется от точки регистрации экземпляра. Для получения согласованных значений координаты _ymouse, всегда измеряемой относительно сцены, следует применять _root._ymouse.

Пример

Клип, в котором будет помещен следующий код, будет перемещаться по вертикали относительно сцены вслед за курсором мыши (вверх и вниз по прямой линии):

```
onEnterFrame = function () {
  this._y = _level0._ymouse;
}
```

См. также *MovieClip*._xmouse

MovieClip._yscale, свойство

Flash 4

возвращает высоту клипа или фильма в процентах **чтение/запись**

mc._yscale

Описание

Свойство _yscale, представляемое числом с плавающей точкой, указывает (в процентах) высоту *mc* по отношению к исходной высоте. Если *mc* – это экземпляр, то его «исходная высота» равна высоте символа экземпляра в библиотеке. Если же *mc* – это главный фильм, то его «исходная высота» равна высоте фильма на этапе разработки.

Если текущая высота *mc* совпадает с исходной, то _yscale равно 100. Если _yscale равно 200, то высота *mc* удвоена по сравнению с исходной. А если _yscale равно 50, то высота *mc* вдвое меньше исходной.

Свойство _yscale масштабирует клип пропорционально вверх и вниз относительно точки регистрации клипа. Чтобы размер клипа изменялся только вверх или только вниз, разместите все его содержимое по одну сторону от точки регистрации в символе клипа (этот прием полезен при создании вертикального указателя в предзагрузчике). Если присвоить свойству клипа _yscale отрицательное значение, он переворачивается по вертикали, как если бы через точку регистрации проходило горизонтальное зеркало (то есть становится зеркальным отражением самого себя по вертикали), после чего с отрицательным значением обращаются как с положительным. Для осуществления вертикального переворота клипа без изменения его размеров надо задать значение свойства _yscale, равным -_yscale.

Пример

```
// Удвоить высоту ball_mc (не изменяя ширины)
ball_mc._yscale *= 2;
// Перевернуть ball_mc в вертикальном направлении
ball_mc._yscale = -ball_mc._yscale;
```

См. также *MovieClip*._xscale

NaN, глобальное свойство

Flash 5

константа, представляющая неверные числовые данные **только для чтения**

NaN

Описание

NaN – это особая числовая константа, представляющая неверные числовые данные (NaN – акроним выражения «Not-a-Number»). Числовые операции, результатом которых не может быть допустимое число, дают значение NaN. Например:

```
Math.sqrt(-1);       // Мнимое число не может быть представлено
15 - "coffee cup";   // "coffee cup" нельзя преобразовать в число.
```

Заметьте, что NaN все же представляет числовые данные, но участвовать в вычислениях не может:

```
typeof NaN;    // Дает значение"number"
```

Применение

Значение NaN редко непосредственно используется в исходном коде, но дает возможность сообщить о возникновении ошибки в ходе выполнения операции. Сравнение NaN с самим собой не дает равенства, поэтому обнаружить его можно только с помощью глобальной функции *isNaN( )*. NaN представляет собой сокращенный эквивалент записи Number.NaN.

См. также

isNaN( ), Number.NaN; «Специальные значения числового типа данных», глава 4

new, оператор

Flash 5

создает новый экземпляр объекта

new *constructor()*

Операнды

constructor Функция-конструктор класса, инициализирующая новый объект.

Возвращаемое значение

Ссылка на вновь созданный объект.

Описание

Оператор *new* создает экземпляр класса, который задает *constructor*. Роль *new* в объектно-ориентированном программировании подробно рассмотрена в главе 12.

См. также Object.__proto__; «Оператор new», глава 5; глава 12

newline, константа

Flash 4

вставляет символ перевода строки

newline

Возвращаемое значение

Символ перевода строки.

Описание

Константа newline представляет стандартный символ перевода строки (ASCII 10). Этот синоним escape-последовательности "\n" или тега HTML
 применяется для принудительного переноса строки в блоке текста (обычно при выводе, в переменной текстового поля).

Пример

```
myOutput = "hello" + newline + "world";
```

См. также

TextField.htmlText, *TextField*.text; «Escape-последовательности», глава 4

nextFrame(), глобальная функция

Flash 2

перемещает головку воспроизведения фильма или клипа вперед на один кадр и останавливает ее

```
nextFrame()
```

Описание

Функция *nextFrame( )* перемещает головку воспроизведения *текущей временной диаграммы* вперед на один кадр и останавливает ее там. Под «текущей временной диаграммой» понимается та, откуда вызвана функция *nextFrame( )*. Функция *nextFrame( )* – это синоним вызова *gotoAndStop(_currentFrame + 1);*. При вызове из последнего кадра временной диаграммы *nextFrame( )* просто останавливает в этом кадре головку воспроизведения, если только за текущей сценой не следует другая, и тогда *nextFrame( )* перемещает головку воспроизведения в кадр 1 следующей сцены.

См. также *gotoAndStop( )*, *MovieClip.nextFrame( )*, *nextScene( )*, *prevFrame( )*

nextScene(), глобальная функция

Flash 2

перемещает головку воспроизведения в кадр 1 следующей сцены

```
nextScene()
```

Описание

Функция *nextScene( )* перемещает главную головку воспроизведения фильма в кадр 1 следующей сцены и останавливает временную диаграмму главного фильма. «Текущей сценой» считается та, из которой вызывается функция *nextScene( )*. Чтобы эта функция возвращала какой-либо результат, ее следует вызывать из главной временной диаграммы фильма, а это значит, что *nextScene( )* не работает внутри экземпляра клипа. При вызове из последней сцены фильма *nextScene( )* отправляет головку воспроизведения в кадр 1 этой сцены и останавливает воспроизведение фильма.

Не рекомендуется применять сцены в фильмах, активно задействующих ActionScript. В отличие от клипов, сцены не представляются объектами, и ими нельзя непосредственно управлять с помощью большинства встроенных функций. Как правило, вместо сцен Flash лучше использовать метки и клипы в качестве псевдосцен на временной диаграмме.

См. также *nextFrame( )*, *prevScene( )*

Number(), глобальная функция

Flash 5

преобразует значение к числовому типу

```
Number(value)
```

Аргументы

value

Выражение, содержащее значение, которое должно быть преобразовано в число.

Возвращаемое значение

Результат преобразования `value` в элементарный числовой тип.

Описание

Функция *Number( )* преобразует полученный ею аргумент в элементарное числовое значение и возвращает это преобразованное значение. Результаты преобразования различных типов данных в элементарный числовой тип описаны в табл. 3.1 главы 3. Необходимость в применении функции *Number( )* возникает редко: ActionScript, если требуется, автоматически преобразует значения к типу *number*.

Не путайте глобальную функцию *Number( )* с конструктором класса *Number*. Первая из них представляет собой функцию, преобразующую величину в тип *number*, тогда как последний является классом, служащим оболочкой для элементарного числового элемента данных в объекте, который может иметь свойства и методы.

Применение

Обратите внимание, что функция *Number( )* часто появляется в файлах *.fla* Flash 4, преобразованных в формат Flash 5. Сведения о действиях с типами данных при конвертировании файлов Flash 4 в файлы Flash 5 можно найти в разделе «Преобразование типов данных Flash 4 во Flash 5» приложения C.

См. также

Класс *Number*, *parseFloat( )*, *parseInt( )*; «Явное преобразование типов данных», глава 3

Number, класс

Flash 5

класс-оболочка для примитивных числовых данных

Конструктор

`new Number(value)`

Аргументы

value Выражение, значение которого необходимо получить и при необходимости преобразовать в числовое, а затем заключить в объект *Number*.

Свойства класса

Доступ к следующим свойствам осуществляется непосредственно как к свойствам класса *Number*, в виде `Number.propertyName`. Для обращения к ним не надо создавать новый экземпляр объекта класса *Number* (то есть не нужен конструктор). Для некоторых из этих свойств, таких как `NaN`, даже не требуется обозначение в виде `Number.propertyName`. Можно просто использовать `NaN` как сокращение для `Number.NaN` (подробно о каждом свойстве рассказано ниже).

MAX_VALUE

Самое большое положительное число, которое можно представить в ActionScript.

MIN_VALUE

Самое маленькое положительное число, которое можно представить в ActionScript.

NaN

Специальное значение «нечисло», указывающее на неверное числовое значение.

NEGATIVE_INFINITY

Любое число, более отрицательное, чем -MAX_VALUE.

POSITIVE_INFINITY

Любое число, большее MAX_VALUE.

Методы

toString( )

Преобразует число в строку.

Описание

У класса *Number* есть два назначения:

- Предоставляет доступ к встроенным свойствам, представляющим специальные числовые значения – MAX_VALUE, MIN_VALUE, NaN, NEGATIVE_INFINITY и POSITIVE_INFINITY, – которые можно использовать для проверки действительности числовых данных или нахождения их в допустимом диапазоне значений.
- Позволяет преобразовывать числа из одной системы счисления в другую, например числа по основанию 10 (десятичные) в числа по основанию 16 (шестнадцатеричные). Подробности см. в описании метода *Number.toString( )*.

Применение

Не требуется создавать новый объект *Number*, если нужен лишь доступ к определенным в нем числовым свойствам. На самом деле проще использовать эквиваленты в виде глобальных свойств, когда они есть (NaN, Infinity и -Infinity). Честно говоря, необходимость в применении свойств класса *Number* вообще возникает редко.

С другой стороны, метод класса *Number toString( )* применяется с созданным экземпляром объекта *Number*. Однако интерпретатор берет на себя создание объекта *Number*, когда вызывается метод для элементарного числового значения. Ниже показано, как с помощью *toString( )* десятичное число преобразуется в шестнадцатеричный эквивалент:

```
x = 102;
x.toString(16);  // Автоматически создается временный объект Number,
                 // служащий "оболочкой" x в этой операции.
```

Такая задача возникает довольно редко, и вряд ли вам придется часто обращаться к классу *Number*.

См. также Объект *Math*; «Тип данных Number», глава 4

Number.MAX_VALUE, свойство

Flash 5

самое большое положительное число, которое можно представить в ActionScript

только для чтения

```
Number.MAX_VALUE
```

Описание

Свойство MAX_VALUE возвращает самое большое положительное число, которое можно представить в ActionScript (1.79769313486231e+308). Им удобно пользоваться, когда желательно начать с очень большого значения, как показано в следующем примере. Никакого числа, большего MAX_VALUE, в ActionScript представить нельзя, поэтому оно считается «плюс бесконечностью» – POSITIVE_INFINITY.

Пример

Ищем минимальное значение, содержащееся в массиве. Переменной minVal присваивается начальное значение MAX_VALUE, поэтому любое последующее значение будет меньше него:

```
var myArray = [-10, 12, 99];
var minVal = Number.MAX_VALUE;
for (var thisValue in myArray) {
  if (myArray[thisValue] < minVal) {
    minVal = myArray[thisValue];
  }
}
trace ("The minimum value is " + minVal);
```

См. также

Inifinity, Number.MIN_VALUE; Number.POSITIVE_INFINITY; «Специальные значения числового типа данных», глава 4

Number.MIN_VALUE, свойство

Flash 5

самое маленькое положительное число, которое можно представить в ActionScript — **только для чтения**

Number.MIN_VALUE

Описание

Свойство MIN_VALUE хранит самое маленькое положительное число, которое можно представить в ActionScript, 4.94065645841247e-324 (не путать с -Number.MAX_VALUE, самым отрицательным числом, допускаемым ActionScript).

См. также

Number.MX_VALUE; «Специальные значения числового типа данных», глава 4

Number.NaN, свойство

Flash 5

константа, представляющая неверные числовые данные (нечисло) — **только для чтения**

Number.NaN

Описание

Свойство Number.NaN введено для совместимости с ECMA-262. См. описание более удобного эквивалентного глобального свойства NaN.

См. также

isNaN( ), NaN

Number.NEGATIVE_INFINITY, свойство

Flash 5

константа, представляющая любое число, более отрицательное, чем –Number.MAX_VALUE

только для чтения

`Number.NEGATIVE_INFINITY`

Описание

Свойство `NEGATIVE_INFINITY` введено для совместимости с ECMA-262. Вместо него следует применять более удобное эквивалентное глобальное свойство `-Infinity`.

См. также `Infinity`, `-Infinity`, `Number.POSITIVE_INFINITY`

Number.POSITIVE_INFINITY, свойство

Flash 5

константа, представляющая любое число, большее чем Number.MAX_VALUE

только для чтения

`Number.POSITIVE_INFINITY`

Описание

Свойство `POSITIVE_INFINITY` введено для совместимости с ECMA-262. Вместо него следует применять более удобное эквивалентное глобальное свойство `Infinity`.

Пример

```
if (score == Number.POSITIVE_INFINITY) {
  trace ("You've achieved the highest possible score.");
}
```

См. также `Infinity`, `-Infinity`, `Number.NEGATIVE_INFINITY`

Number.toString(), метод

Flash 5

преобразует число в строку

`numberObject``.toString(`*`radix`*`)`

Аргументы

radix

Целое число от 2 до 36, задающее основание числовой системы, используемой для представления *`numberObject`* в строковом формате. Этот аргумент может отсутствовать. Если он опущен, принимается значение 10.

Возвращаемое значение

Значение *`numberObject`*, преобразованное в строку.

Описание

Метод *toString( )* извлекает значение из объекта *Number*, преобразует это значение в строку и возвращает полученную строку. Необязательный аргумент *`radix`* указывает, в какую систему счисления преобразуется число (например, двоичную, восьмеричную, десятичную или шестнадцатеричную). Буквы A–Z предназначены для представления цифр со значениями от 10 до 35 соответственно, хотя обычно используются только буквы от A до F (для представления шестнадцатеричных цифр от 10 до 15).

Для того чтобы применить *Number.toString( )* к элементарному числовому значению, заключите его в круглые скобки, например:

```
(204).toString(16);
```

Пример

```
x = new Number(255);
trace(x.toString());        // Выводит: "255" (то есть десятичное)
trace(x.toString(16));      // Выводит: "ff" (то есть шестнадцатеричное)
trace(x.toString(2));       // Выводит: "11111111" (то есть двоичное)

// Преобразовать шестнадцатеричный литерал в десятичную строку
trace((0xFFFFFF).toString(10));        // Выводит: "16777215" (строку)
// Преобразовать шестнадцатеричный литерал в десятичный
x = parseInt(0xFFFFFF).toString(10); // x равно 16777215 (числу), а не "16777215"
```

См. также

Number ( ), *Object.toString( )*, *parseInt( )*; «Целочисленные литералы», глава 4

Object, класс

Flash 5

основа для всех остальных классов и родовых объектов

Конструктор

```
new Object()
```

Свойства

constructor

Ссылка на функцию конструктора класса, создающую объект.

__proto__

Ссылка на свойство `prototype` функции конструктора класса.

Методы класса

Следующий метод вызывается через сам класс *Object*, а не через экземпляр класса *Object*:

registerClass( )

Назначить конструктор для символа клипа.

Методы

Следующие методы представляют собой методы экземпляров класса *Object*:

addProperty( )

Определяет свойство доступа для объекта или класса.

hasOwnProperty( )

Проверяет, определено ли свойство в самом объекте.

toString( )

Преобразует значение объекта в строку.

unWatch( )

Удаляет имеющуюся контрольную точку.

valueOf( )

Извлекает из объекта значение элементарного типа, если оно существует.

watch( )

Создает контрольную точку для фильтрации присваивания свойствам.

Описание

В объектной модели ActionScript класс *Object* – базовый и применяется с двумя главными целями: как конструктор для создания новых общих объектов и как надкласс, на котором основываются новые классы. Все классы ActionScript, как определяемые пользователем, так и встроенные, являются потомками класса *Object*. Поэтому все объекты всех классов наследуют свойства и методы *Object* (хотя некоторые классы переопределяют эти свойства). Однако обращение к методу класса *Object.registerClass( )* происходит непосредственно как к методу конструктора класса *Object*, а не через экземпляр класса.

Чтобы создать общий объект класса *Object* непосредственно в коде, не применяя конструктор, можно воспользоваться литералом объекта, подобно литералу строки или литералу массива. Литерал объекта состоит из ряда пар имя/значение свойства, разделенных запятыми и заключенных в фигурные скобки. Вот общий синтаксис:

```
{ property1: value1, property2: value2, property3: value3 }
```

Имена свойств в литерале объекта должны быть допустимыми идентификаторами (см. главу 15). Значениями могут быть любые допустимые выражения. Например:

```
// Литерал объекта с двумя числовыми свойствами
myObject = { x: 30, y: 23 };
// Присвоить свойству x  значение с помощью сложного выражения
myOtherObject = { x: Math.floor(Math.random() * 50 + 1) };
```

Поскольку литералы объектов всегда создают общие анонимные объекты, они применяются обычно в случае временной потребности в данных, имеющих формат объекта, например при вызове *Sound.setTransform( )*, *Color.setTransform( )* или *MovieClip.localToGlobal( )*.

См. также Глава 12

Object.addProperty(), метод

Flash 6

определяет свойство доступа для объекта или класса

someObject.addProperty(*propertyName*, *getterFunction*, *setterFunction*)

Аргументы

propertyName

Строка с именем свойства, которое должно быть добавлено. Имя должно соответствовать правилам создания идентификаторов, описанным в разделе «Индентификаторы» главы 15.

getterFunction

Функция чтения («getter»), которая возвращает текущее значение свойства *propertyName*. Принято давать ей имя *getPropertyName*, где *PropertyName* соответствует параметру *propertyName*. Это может быть пользовательская функция или встроенный метод объекта, как показано ниже в разделе «Описание».

setterFunction

Функция записи («setter»), которая должна внутренне присваивать значение свойству *propertyName*. Принято давать ей имя *setPropertyName*, где *PropertyName* соответствует параметру *propertyName*. Это может быть пользовательская функция или встроенный метод объекта, как показано ниже в разделе «Описание». Если свойство должно быть доступно только для чтения, то в качестве значения *setterFunction* должно выступать `null`.

Возвращаемое значение

Булево значение `true`, если операция была успешной, и `false`, если *propertyName* не является допустимым идентификатором или *getterFunction* или *setterFunction* не являются объектами функции.

Описание

Метод *addProperty()* определяет так называемое свойство доступа («getter/setter») с именем *propertyName* объекта *someObject*. Внешне свойство доступа ведет себя как обычное свойство, за исключением того, что получение и задание его значения осуществляются автоматически двумя функциями: *getterFunction* и *setterFunction*. Это позволяет разработчикам обеспечивать удобное обращение к свойствам, получение и установка которых требуют сложных операций. Когда *someObject.propertyName* участвует в любом выражении (кроме левой части операции присваивания), интерпретатор должен установить и возвратить значение свойства. Для этого интерпретатор выполняет функцию, заданную параметром *getterFunction*, и использует возвращаемое ею значение в качестве значения *propertyName*. Например:

```
// Создать объект.
var gamePlayer = new Object();
// Добавить свойство доступа. Сделаем свойство доступным только для чтения,
// передав null в качестве значения параметра setFunction.
gamePlayer.addProperty("score", getScore, null);
// Определим функцию, которая должна выполняться при чтении значения gamePlayer.score.
// Этот пример надуман, а к реальной практике более близкое отношение имеет следующий пример.
function getScore () {
  return 1500;
}
// Чтение gamePlayer.score, которое автоматически вызывает функцию getScore()
// и использует возвращаемое ею значение (1500) для вывода в окне Output.
trace(gamePlayer.score);
```

Если *someObject.propertyName* находится в левой части оператора присваивания (слева от знака равенства), интерпретатор должен задать значение свойства. Для этого он выполняет функцию, заданную параметром *setterFunction*, передавая ей новое значение (указанное в инструкции присваивания значения свойству) в качестве аргумента. Часто аргумент просто сохраняется в переменной и в дальнейшем извлекается функцией *getterFunction*:

```
// Создать объект.
var gamePlayer = new Object();
// Добавить свойство доступа, на этот раз указав setScore() в качестве
// функции, выполняемой при записи в gamePlayer.score.
gamePlayer.addProperty("score", getScore, setScore);
// Вот функция чтения("getter").
function getScore () {
  return gamePlayer.currentScore;
}
```

```
// Вот функция записи ("setter"). Присваиваемое новое значение свойства
// передается в качестве единственного параметра, который мы назвали newVal.
function setScore (newVal) {
  // Сохранить присвоенное свойству значение внутренне в отдельном свойстве,
  // чтобы в дальнейшем извлекать его с помощью getScore().
  gamePlayer.currentScore = newVal;
}
gamePlayer.score = 2000;  // Здесь вызывается setScore() с newVal, равным 2000.
trace(gamePlayer.score);  // Здесь вызывается getScore(). Выводит: 2000
// Чтобы убедиться, что все работает, проверим значение внутреннего свойства currentScore.
trace(gamePlayer.currentScore);  // Выводит: 2000
```

Значение, передаваемое *setterFunction*, нельзя записывать в свойство с таким же именем, как *propertyName*.

В следующем коде `newVal` ошибочно присваивается свойству `score`, что вызывает конфликт с одноименным свойством доступа:

```
function setScore (newVal) {
  // ЭТО НАРУШАЕТ РАБОТУ СВОЙСТВА ДОСТУПА!
  gamePlayer.score = newVal;
}
```

Чтобы удалить свойство доступа, достаточно удалить его и вспомогательные функции чтения/записи:

```
delete someObject.propertyName
delete getterFunc;
delete setterFunc;
```

Свойства доступа обычно применяются, когда значение свойства должно внутренне рассчитываться каким-то методом или когда предъявляемое значение свойства отличается от значения, хранящегося внутри класса. Допустим, например, что мы хотим определить для объекта *Ball* свойство `color` (цвет). Мы хотим, чтобы извне `color` принимал в качестве значений строки естественного языка «blue», «red» и «green». Но внутри значение свойства `color` должно храниться как целое число RGB. Мы хотим задавать свойство `color` так:

```
redBall.color = "red";
```

Еще мы хотим, чтобы внутри свойства эквивалент «red» хранился как число `0xFF0000`. Извлекая значение `redBall.color`, мы должны преобразовать число `0xFF0000` обратно в строку «red». Вот как мы создаем наше свойство доступа, на естественном языке называемое `color`:

```
// Создать конструктор класса Ball.
function Ball () {
  // Создать метод записи для свойства color.
  this.setColor = function (newVal) {
    // Преобразовать присваиваемое значение к нижнему регистру.
    var theColor = newVal.toLowerCase();
    // Преобразовать обычное название в эквивалентное число
    // и записать его в свойство colorNum.
    if (theColor == "red") {
      this.colorNum = 0xFF0000;
    } else if (theColor == "green") {
      this.colorNum = 0x00FF00;
```

```
    } else if (theColor == "blue") {
      this.colorNum = 0x0000FF;
    } else {
      // Если указан цвет, отличный от красного, зеленого или синего, считать его черным.
      this.colorNum = 0x000000;
    }
  }

  // Создать метод чтения свойства color.
  this.getColor = function () {
    // Преобразовать внутреннее свойство colorNum в эквивалентное естественное
    // название и возвратить это название в качестве значения свойства color.
    if (this.colorNum == 0xFF0000) {
      return "red";
    } else if (this.colorNum == 0x00FF00) {
      return "green";
    } else if (this.colorNum == 0x0000FF) {
      return "blue";
    } else {
      return "black";
    }
  }

  // Добавить свойство доступа color.
  this.addProperty("color", this.getColor, this.setColor);
}

// Создать экземпляр ball.
redBall = new Ball();

// Присвоить шарику цвет. Обратите внимание, что функция записи
// допускает в названии цвета любую смесь верхнего и нижнего регистров.
redBall.color = "RED";
// Проверим цвет. Функция чтения всегда возвращает названия на нижнем регистре.
trace(redBall.color);  // Выводит: red
```

Обратите внимание, что функции чтения и записи вызываются как методы объекта, содержащего свойство доступа, поэтому они могут обращаться к этому объекту через `this` и получать доступ к его свойствам и методам. Однако в таком виде этот код неэффективен, поскольку присваивает методы *setColor( )* и *getColor( )* непосредственно каждому экземпляру *Ball*. К счастью, свойства доступа можно добавлять к свойству `prototype` конструктора класса, где обычно определяются методы класса. Вот усовершенствованный вариант нашего свойства `color`, в котором определения методов и свойств прикреплены к `prototype`:

```
// Класс Ball
function Ball () {
  // Здесь находится код инициализации экземпляра...
}

// Функция записи
Ball.prototype.setColor = function (newVal) {
  var theColor = newVal.toLowerCase();
  if (theColor == "red") {
    this.colorNum = 0xFF0000;
  } else if (theColor == "green") {
    this.colorNum = 0x00FF00;
  } else if (theColor == "blue") {
```

```
    this.colorNum = 0x0000FF;
  } else {
    this.colorNum = 0x000000;
  }
}

// Функция чтения
Ball.prototype.getColor = function () {
  if (this.colorNum == 0xFF0000) {
    return "red";
  } else if (this.colorNum == 0x00FF00) {
    return "green";
  } else if (this.colorNum == 0x0000FF) {
    return "blue";
  } else {
    return "black";
  }
}

// Свойство доступа color, определенное в Ball.prototype
Ball.prototype.addProperty("color",
                           Ball.prototype.getColor,
                           Ball.prototype.setColor);

// Создать экземпляр ball
redBall = new Ball();

// Присвоить ball значение color и проверить его
redBall.color = "RED";
trace(redBall.color);  // Выводит: red
```

Обратите внимание, что свойства доступа введены исключительно для удобства. Вполне допустимо явно показать методы чтения и записи, не предоставляя интерфейса свойства доступа к значениям, с которыми они работают. Например, наш класс *Ball* мог бы просто потребовать непосредственного применения методов *setColor( )* и *getColor( )*, вместо того чтобы предоставлять свойство доступа `color`. Наоборот, в ActionScript допустимо непосредственное обращение к внутреннему свойству `colorNum`, хотя это и считается плохой практикой объектно-ориентированного программирования. Другими словами, можно получать свойство `colorNum` и присваивать ему значение как `redBall.colorNum`, вообще не прибегая к функциям доступа.

Ошибки реализации

Во Flash 6 свойства доступа нельзя добавлять непосредственно в объект `_global`. Однако их можно добавлять в объекты, определенные в объекте `_global`.

Пример

Следующий остроумный код, предложенный Гаем Уотсоном (Guy Watson), сотрудничающим с *http://www.flashguru.co.uk*, создает свойство доступа `depth` для всех клипов. Предоставляемый им интерфейс проще использовать и запомнить, чем тот, который предоставляют встроенные методы *getDepth( )* и *swapDepths( )*:

```
// Добавить свойство.
MovieClip.prototype.addProperty("depth",
                                MovieClip.prototype.getDepth,
                                MovieClip.prototype.swapDepths);

// Опробуем его. Обратите внимание, что swapDepths() может принимать как число,
// так и целевой клип.
```

```
this.createEmptyMovieClip("empty_mc", 15);
trace(empty_mc.depth);      // Выводит: 15
empty_mc.depth = 20;        // Передать целое число в swapDepths().
                            // Depth становится равным 20.
trace(empty_mc.depth);      // Выводит: 20
empty_mc.depth = ball_mc;   // Обменяться глубиной с ball_mc (переданным в swapDepths())
```

См. также

MovieClip.getDepth( ), *MovieClip.swapDepths( )*, `Object.__proto__`, *Object.watch( )*; глава 12

Object.constructor, свойство

Flash 5

ссылка на функцию конструктора класса, применяемую для создания объекта — **чтение/запись**

`someObject.constructor`

Описание

В свойстве `constructor` хранится ссылка на функцию конструктора, посредством которой был создан *someObject*. Например, свойство `constructor` объекта *Date* содержит ссылку на функцию конструктора *Date*:

```
now = new Date();
now.constructor == Date;       // Дает значение true
```

Обратите внимание, что в JavaScript свойство `constructor` всегда считывается из свойства `prototype` конструктора объекта *someObject* (`someObject.___proto___.constructor`). Во Flash Player версий 6 и 5 свойство `constructor` записывается непосредственно в *someObject*, поэтому оно всегда точно ссылается на фактическую функцию-конструктор *someObject*. Этот нюанс подробно рассмотрен в разделе «Более сложные вопросы стандартного назначения надкласса» главы 12.

Применение

Присваивать новое значение свойству `constructor` объекта не рекомендуется, поскольку в результате изменяется естественная структура наследования классов.

См. также

«Свойство constructor», глава 12 и «Более сложные вопросы стандартного назначения надкласса», глава 12

Object.hasOwnProperty(), метод

Flash 6 (не документирован)

проверяет, определено ли свойство непосредственно в объекте

`someObject.hasOwnProperty(propName)`

Аргументы

propName

Строка с именем проверяемого свойства.

Возвращаемое значение

Булево значение `true`, если *propName* представляет собой прямое свойство или метод *someObject*, в противном случае `false`.

Описание

Недокументированный метод *hasOwnProperty( )* возвращает булево значение, указывающее, что свойство или метод с именем *propName* определяется непосредственно в *someObject*, и тогда оно имеет значение `true`; в противном случае оно имеет значение `false`. Применяется преимущественно во время отладки, чтобы отличить свойство, прикрепленное непосредственно к экземпляру объекта, от свойства, унаследованного от класса по цепочке прототипов. Например:

```
// Класс Rectangle
function Rectangle (w, h) {
  // Прямые свойства
  this.width = w;
  this.height = h;
}
// Унаследованное свойство
Rectangle.prototype.numSides = 4;

// Экземпляр Rectangle
rect = new Rectangle(10, 100);
// Получить прямое свойство:
trace(rect.width);                        // Выводит: 10
trace(rect.hasOwnProperty("width"));      // Выводит: true

// Получить унаследованное свойство:
trace(rect.numSides);                     // Выводит: 4
trace(rect.hasOwnProperty("numSides"));   // Выводит: false
```

Обратите внимание, что `numSides` доступно через `rect`, но не определено в самом `rect`, поэтому *rect.hasOwnProperty("numSides")* возвращает `false`. Однако `false` возвращалось бы и в том случае, если бы это свойство нигде не было определено (ни в `rect`, ни в его классе).

Упражнение 1: Напишите пользовательскую функцию *hasProperty( )*, которая возвращает число 2, если *hasOwnProperty( )* дает значение `true`, возвращает 1, если *hasOwnProperty( )* дает значение `false`, но свойство унаследовано от класса, и возвращает 0, если у объекта нет такого свойства. (Подсказка: даже объявленное свойство может содержать значение `undefined`, поэтому проверьте с помощью *hasOwnProperty( )* все объекты в цепочке прототипов. Если вы доберетесь до класса *Object* и *hasOwnProperty( )* по-прежнему возвратит `false`, значит, свойство не существует.)

Упражнение 2: Напишите пользовательскую функцию *showProperties( )*, которая выводит имена и значения всех свойств объекта, показывая, какие из них получены по цепочке прототипов.

Упражнение 3: Напишите пользовательскую функцию *showMethods( )*, аналогичную *showProperties( )*, но показывающую имена методов объекта, а не его свойств. (Подсказка: воспользуйтесь *instanceof* для определения того, что свойство является экземпляром класса *Function*.)

Метод *hasOwnProperty( )* можно использовать со свойствами доступа, добавленными с помощью *Object.addProperty( )*. Он определен в спецификации ECMA-262, но официально не документирован Macromedia.

См. также *instanceof*, *Object.*`__proto__`; глава 12

Object.__proto__, свойство

Flash 5

ссылка на свойство prototype конструктора объекта **чтение/запись**

someObject.__proto__

Описание

Свойство __proto__ хранит ссылку на автоматически назначаемое свойство prototype функции конструктора *someObject*, служащее для передачи свойств вниз по иерархии классов. Свойство объекта __proto__ в основном используется интерпретатором для поиска унаследованных объектом свойств, но его можно также применять для определения класса объекта, как показано в примере 12.5. Некоторые разработчики посредством __proto__ устанавливают наследование между подклассом и его надклассом, как описывается в разделе «Более сложные вопросы стандартного назначения надкласса» главы 12.

Обратите внимание, что __proto__ начинается и заканчивается *двумя* символами подчеркивания.

Применение

Не рекомендуется присваивать новое значение свойству __proto__ объекта, поскольку в результате изменяется естественная структура наследования классов.

См. также *Function*.prototype, *instanceof*; «Свойство __proto__», глава 12

Object.registerClass(), метод класса

Flash 6

назначает конструктор для символа клипа

Object.registerClass(*symbolID*, *theClass*)

Аргументы

symbolID

Строка, задающая идентификатор связи экспортируемого символа клипа, указанный в Options → Linkage библиотеки.

theClass

Функция конструктора класса, которую надо связать с *symbolID*. Это должна быть ссылка на объект, а не строка. Чтобы разрегистрировать класс, используется значение null.

Возвращаемое значение

Булево значение true, если *registerClass( )* вызван с правильным количеством аргументов, и false в противном случае. Обратите внимание, что Flash 6 возвращает false, только если *registerClass( )* вызван с неверным числом аргументов; проверка успешной регистрации класса не осуществляется.

Описание

Метод *registerClass( )* входит в набор средств для создания подкласса клипа из библиотечного символа. Он соединяет класс ActionScript с экспортированным символом библиотеки. Когда экземпляры символа вручную или программно с помощью мето-

да *attachMovie( )* или *duplicateMovieClip( )* помещаются на временную диаграмму, Flash автоматически вызывает *theClass* в качестве функции-конструктора нового экземпляра и задает класс этого экземпляра как *theClass*. При выполнении функции конструктора ключевое слово this указывает на создаваемый экземпляр. Например, следующий код создает функцию конструктора *Vehicle( )*, а затем связывает ее с символом клипа, экспортированным в библиотеке как carSymbol:

```
// Функция конструктора
function Vehicle () {
  // Задать скорость нового клипа
  this.pixelsPerFrame = this.pixelsPerSecond / 12;
}

// Соединить конструктор с carSymbol
Object.registerClass("carSymbol", Vehicle);
```

Экземпляры клипа carSymbol создаются функцией *Vehicle( )*. Внутри этой функции ключевое слово this указывает на новый экземпляр. Обратите, однако, внимание, что *Vehicle( )* не принимает параметров. Параметры передаются в виде свойств объекта методами *attachMovie( )* или *duplicateMovieClip( )* (с помощью *initObject*). Например, свойство pixelsPerSecond передается с помощью *attachMovie( )* и становится доступно при выполнении *Vehicle( )*. Функция *Vehicle( )* использует pixelsPerSecond для расчета скорости экземпляра в пикселах на кадр (если считать, что частота смены кадров фильма составляет 12 кадров в секунду).

Для того чтобы обеспечить наследование экземплярами нашего класса *Vehicle* всех методов и свойств класса *MovieClip*, надо сделать *Vehicle* подклассом *MovieClip*, например:

```
// Задать MovieClip в качестве надкласса Vehicle
Vehicle.prototype = new MovieClip();
```

Зарегистрированные классы всегда должны быть потомками *MovieClip*. После регистрации нашего класса и задания *MovieClip* в качестве его надкласса можно строить класс дальше, как обычно. Вот полный код класса:

```
// Конструктор не принимает параметры. Их можно передать с помощью
// attachMovie() и duplicateMovieClip() через параметр initObject.
function Vehicle () {
  this.pixelsPerFrame = this.pixelsPerSecond / 12;
}

// Связать класс Vehicle с символом библиотеки.
Object.registerClass("carSymbol", Vehicle);

// Задать MovieClipet в качестве надкласса Vehicle.
Vehicle.prototype = new MovieClip();

// Добавить метод goForward() в класс Vehicle.
Vehicle.prototype.goForward = function () {
  this.onEnterFrame = function () {
    this._x += this.pixelsPerFrame;
  }
}

// Добавить метод brake() в класс Vehicle.
Vehicle.prototype.brake = function () {
  delete this.onEnterFrame;
}
```

```
// Создать экземпляр. Конструктор Vehicle вызывается автоматически.
// Свойства initObject (последнего аргумента attachMovie())
// присваиваются экземпляру fastCar_mc перед выполнением конструктора.
this.attachMovie("carSymbol", "fastCar_mc", 1, { pixelsPerSecond: 100 });

// Вызвать метод.
this.fastCar_mc.goForward();
```

Для того чтобы отменить регистрацию класса, вызовите *registerClass( )*, передав ID клипа, который надо разрегистрировать, в качестве параметра `symbolID` и `null` в качестве *theClass*. Например:

```
// Снова превратить carSymbol в обычный клип. Классом carSymbol
// снова станет MovieClip, а не подкласс MovieClip.
// fastCar_mc, тем не менее, останется экземпляром Vehicle.
Object.registerClass("carSymbol", null);
```

Клипы, существующие во время разрегистрации символа, остаются экземплярами своего зарегистрированного класса. Дерегистрация класса влияет только на создаваемые в последующем экземпляры.

Обратите внимание, что сразу после регистрации класса его экземпляры можно создавать программно с помощью *attachMovie( )* или *duplicateMovieClip( )*, как в нашем примере *Vehicle*. Однако экземпляры, помещенные вручную на временную диаграмму на этапе разработки, должны быть созданы по крайней мере на один кадр позднее регистрации класса. Экземпляры, помещенные вручную в тот же кадр, где происходит регистрация класса, не будут членами зарегистрированного класса. Решить эту проблему можно с помощью прагмы `#initclip`.

Сведения о том, как сделать класс клипа независимым (то есть поместить код класса на временную диаграмму самого клипа), есть в описании прагмы `#initclip` в данном «Справочнике по языку». Подробнее о создании классов из клипов рассказано в главе 14.

Обратите внимание, что один конструктор класса можно зарегистрировать с несколькими символами библиотеки, но один символ библиотеки нельзя зарегистрировать более чем с одним классом.

Обычно *Object.registerClass( )* используется с клипами, но может также присваивать конструктору класса произвольный строковый идентификатор. Посредством этого идентификатора Flash восстанавливает класс объекта при импорте объекта во Flash Player с помощью *SharedObject*, *LocalConnection* или Macromedia Flash Remoting MX (Flash Remoting). Отправляя из проигрывателя экземпляр зарегистрированного класса, Flash сопровождает его идентификатором класса. Позже, считывая экземпляр обратно в проигрыватель, Flash получает идентификатор класса экземпляра и вновь связывает экземпляр с его классом, что называют приведением типа (*casting*) объекта.

Допустим, например, что мы создаем экземпляр `p1` пользовательского класса *GamePlayer*, который определяет свойство `score` и метод *addPoints( )* следующим образом:

```
// Конструктор класса
function GamePlayer () {
  // Свойство score
  this.score = 0;
}
```

```
// Метод addPoints()
GamePlayer.prototype.addPoints = function (points) {
  this.score += points;
}

// Экземпляр p1
p1 = new GamePlayer();
p1.addPoints(100);
```

Как будет видно из кода, `p1` сохраняется на диске с помощью *SharedObject*, и его связь с классом *GamePlayer* будет потеряна, потому что у *GamePlayer* нет идентификатора, связывающего его с сериализуемым объектом `p1`. Извлечь объект `p1` и обнаружить в нем свойство `score` по-прежнему можно (поскольку `score` определяется непосредственно в `p1`), но получить доступ к методу его класса *addPoints( )* по цепочке прототипов будет нельзя. Для того чтобы сохранить связь между `p1` и *GamePlayer*, мы должны добавить регистрацию класса, которая присвоит произвольный идентификатор ("GamePlayerClass") классу *GamePlayer*:

```
Object.registerClass("GamePlayerClass", GamePlayer);
```

После этого все экземпляры *GamePlayer* будут невидимо помечаться как принадлежащие классу с идентификатором «GamePlayerClass» (имя произвольно, но принято использовать имя класса с добавлением слова «Class»). Сериализуя `p1` перед выводом его из проигрывателя, Flash включает в посылаемые данные идентификатор «GamePlayerClass». (Сериализация – процедура преобразования свойств объекта в другой формат для целей передачи.) Когда Flash считывает `p1` из внешнего источника, он обнаруживает в данных идентификатор класса. Тогда он пытается привести `p1` к тому классу, который зарегистрирован с этим идентификатором (*GamePlayer*). Формально нет никакой гарантии, что класс *GamePlayer*, действовавший в тот момент, когда сохранялся *SharedObject*, окажется тем же самым, что и класс *GamePlayer*, действующий в момент извлечения *SharedObject*. Flash просто пытается привести тип извлеченного объекта к классу, зарегистрированный идентификатор которого совпадает с зарегистрированным идентификатором объекта (предполагается, что это тот же конструктор класса, который использовался, когда данные сохранялись первоначально).

Однако в процессе воссоединения объекта со своим классом Flash также выполняет конструктор класса для восстанавливаемого объекта. Обычно строить заново объект не требуется (и не следует), поэтому надо добавить в конструктор флажок, позволяющий предотвратить ненужную повторную инициализацию. Вот базовый код, необходимый для создания зарегистрированного класса:

```
// Класс GamePlayer
function GamePlayer () {
  // Инициализация только при первом выполнении конструктора
  if (this.inited) {
    return;
  }
  // Пометим, что этот объект инициализирован
  this.inited = true;

  // Продолжим инициализацию
  this.score = 0;
}

// Метод addPoints()
GamePlayer.prototype.addPoints = function (points) {
  this.score += points;
}
```

```
// Связать GamePlayer с идентификатором GamePlayerClass
Object.registerClass("GamePlayerClass", GamePlayer);
```

А вот код, создающий экземпляр объекта и сохраняющий его как `player1` с помощью *SharedObject*:

```
// Создать экземпляр GamePlayer
p1 = new GamePlayer();
p1.addPoints(100);
// Создать SharedObject
gameData_so = SharedObject.getLocal("gameData");
// Создать свойство для сохранения нашего объекта GamePlayer
gameData_so.data.player1 = p1;
// Попробуем сохранить SharedObject
gameData_so.flush();
```

Ниже приводится код для извлечения зарегистрированного объекта из *SharedObject*. Не существует способов сохранить конструктор или методы объекта, даже зарегистрировав его класс. Поэтому конструктор класса *GamePlayer* и метод *addPoints( )* должны быть определены в фильме-приемнике, а класс должен быть зарегистрирован с помощью *registerClass( )*. Обратите внимание, что, как уже отмечалось, конструктор *GamePlayer* и его методы не обязательно совпадают с теми, которые существовали в момент сохранения объекта, хотя обычно это так. Поэтому начало кода в принимающем фильме и кода в передающем фильме, который мы видели раньше, совпадают. Этот код может находиться в том же фильме, что и отправитель (и тогда его не надо дублировать), либо он может быть в другом фильме (возможно, даже если это одна машина, поддерживающем связь с фильмом-отправителем через объект *LocalConnection*).

```
// Класс GamePlayer
function GamePlayer () {
  // Инициализация только при первом выполнении конструктора
  if (this.inited) {
    return;
  }
  // Пометим, что этот объект инициализирован
  this.inited = true;

  // Продолжим инициализацию
  this.score = 0;
}

// Метод addPoints()
GamePlayer.prototype.addPoints = function (points) {
  this.score += points;
}

// Связать GamePlayer с идентификатором GamePlayerClass
Object.registerClass("GamePlayerClass", GamePlayer);
```

А вот код для извлечения *SharedObject* и воссоздания экземпляра объекта, сохраненного в `player1`. Обратите внимание, что при выполнении функции *SharedObject.getLocal( )* (и восстановлении сериализованных данных) вызывается конструктор *GamePlayer*, хотя мы не обращаемся к объекту экземпляра `player1` непосредственно, пока он не будет присвоен переменной `p1`.

```
// Извлечь SharedObject.
gameData_so = SharedObject.getLocal("gameData");
// Извлечь объект GamePlayer.
p1 = gameData_so.data.player1;
```

```
// Получить значение score для  p1:
trace(p1.score);
// Добавить к score 50 очков. Получится только потому, что класс GamePlayer зарегистрирован.
p1.addPoints(50);
```

Аналогичная схема действует для объектов, пересылаемых по сети с помощью *LocalConnection* и Flash Remoting. Дополнительные сведения о Flash Remoting см. по адресу:

http://www.macromedia.com/software/flash/flashremoting/

См. также

`#initclip`, *LocalConnection*, *MovieClip.attachMovie( )*, `Object.__proto__`, объект *SharedObject* ; глава 14

Object.toString(), метод

Flash 5

возвращает значение объекта в виде строки

`someObject.toString()`

Возвращаемое значение

Строка, определенная в объекте, описывающая его или представляющая его еще каким-либо образом.

Описание

Метод *toString( )* возвращает описание `someObject` в виде строки. Согласно спецификации ECMA-262, *someObject.toString( )* должен по умолчанию возвращать выражение:

```
"[object " + class + "]"
```

где `class` — это внутренне определенное имя класса, к которому принадлежит `someObject`. В действительности некоторые классы ActionScript заменяют метод *toString( )* класса *Object*, действующий по умолчанию, чтобы предоставить более содержательную информацию о каждом экземпляре класса. Для тех классов, которые не имеют пользовательского метода *toString( )*, ActionScript не сообщает имя отдельного класса. Поэтому по умолчанию для экземпляров большинства классов метод *toString( )* возвращает:

```
[object Object]
```

Интерпретатор ActionScript автоматически вызывает метод *toString( )*, когда `someObject` используется в строковом контексте. Например:

```
x = new Object();
trace(x);       // Выводит: "[object Object]"
```

При создании собственных классов можно реализовать пользовательский метод *toString( )*. Например, метод *Date.toString( )* возвращает дату и время, а метод *Array.toString( )* возвращает разделяемый запятыми список элементов массива. Метод *Number.toString( )* возвращает число в виде строки.

Пример

В этом примере показано, как создать собственный метод *toString( )* для класса *Ball*, который был создан в главе 12:

```
// Создать конструктор Ball
function Ball (radius, color, xPosition, yPosition) {
```

```
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Добавить пользовательский метод toString()
Ball.prototype.toString = function () {
  return "A ball with the radius " + this.radius;
};

// Создать новый объект ball
myBall = new Ball(6, 0x00FF00, 145, 200);

// Теперь получить строковое значение myBall
trace(myBall);  // Выводит: "A ball with the radius 6"
```

См. также *Array.toString( ), Date.toString( ), Number.toString( ), Object.valueOf( )*

Object.unwatch(), метод

Flash 6

удаляет имеющуюся контрольную точку

`someObject.unwatch(prop)`

Аргументы

prop

Строка с именем свойства, контроль за которым следует прекратить. Имя должно соответствовать правилам создания идентификаторов, описанным в разделе «Идентификаторы» главы 15.

Возвращаемое значение

Булево значение, указывающее, было ли свойство `prop` под контролем (`true`, если `prop` существует и контролировалось, `false` в противном случае).

Описание

Метод *unwatch( )* удаляет контрольную точку, ранее установленную на `prop` с помощью метода *Object.watch( )*. После удаления контрольной точки присваивание `prop` новых значений более не будет происходить через посредство функции обратного вызова *watch( )*. Обратите внимание, что контрольную точку можно убрать только с помощью *Object.unwatch( )* – удаление контролируемого свойства не уничтожает контрольную точку. Подробное описание см. в *Object.watch( )*.

См. также *Object.watch( )*

Object.valueOf(), метод

Flash 5

возвращает элементарное значение объекта

`someObject.valueOf()`

Возвращаемое значение

Внутренне определенное элементарное значение `someObject`, если с `someObject` связано элементарное значение. Если элементарное значение с `someObject` не связано, то возвращается сам `someObject`.

Описание

Метод *valueOf( )* возвращает значение примитивного типа, связанное с объектом, если такая связь существует. Этот метод чаще всего применяется с экземплярами классов *Number*, *String* и *Boolean*, у которых есть связанные с ними примитивные данные. Необходимость в явном вызове *valueOf( )* возникает редко: он автоматически вызывается интерпретатором при использовании `someObject` там, где предполагается примитивный тип данных.

См. также

Boolean.valueOf( ), *MovieClip.valueOf( )*, *Number.valueOf( )*, *Object.toString( )*, *String.valueOf( )*

Object.watch(), метод

Flash 6

создает контрольную точку для перехвата присваивания свойству

`someObject.watch(prop, callback, userData)`

Аргументы

prop

Строка с именем контролируемого свойства `someObject`. Имя должно соответствовать правилам создания идентификаторов, описанным в разделе «Идентификаторы» главы 15.

callback

Функция обратного вызова, вызываемая при попытке присвоить `prop` значение. Значение, возвращаемое функцией, становится фактическим значением, присваиваемым свойству.

userData

Произвольное значение, передаваемое `callback`.

Возвращаемое значение

Булево значение `true`, если контрольная точка успешно создана, `false` в противном случае (например, если `callback` – недопустимая функция).

Описание

Метод *watch( )* регистрирует так называемую *контрольную точку* (*watchpoint*) для `someObject.prop`. Когда действует контрольная точка, `callback` выступает посредником для всех присваиваний значений `someObject.prop`.

Всякий раз, когда свойству `prop` присваивается значение, выполняется `callback`, которой передаются четыре аргумента:

prop

Имя модифицируемого свойства.

oldval

Прежнее значение свойства.

newval

Предлагаемое новое значение свойства, как оно задано в присваивании свойству, заставившем сработать контрольную точку.

userData

Необязательное значение данных, указанное в первоначальном вызове *watch( )*.

Таким образом, *callback* должна иметь следующий общий формат:

```
function someWatch (prop, oldval, newval, userData) {
  return theNewPropertyValue;
}
```

Функция *callback* должна возвратить фактическое значение, которое будет присвоено свойству. Например, следующий код размещает на rectangle.numSides контрольную точку, которая просто отклоняет все присваивания numSides благодаря тому, что возвращает *oldval* (фактически делая numSides свойством только для чтения). Это называют аннулированием (*nullifying*) присваивания.

```
// Создать объект со свойством numSides
var rectangle = new Object();
rectangle.numSides = 4;
// Добавить контрольную точку
rectangle.watch("numSides", rectWatch);
// Создать функцию обратного вызова - посредник присваиваний numSides
function rectWatch (prop, oldval, newval, userData) {
  // Возвратить прежнее значение, которое будет заново присвоено свойству
  return oldval;
}
// Применение: попытка изменить значение numSides
rectangle.numSides = 3;
// Проверим, изменилось ли...
trace(rectangle.numSides);  // Нет! Выводит: 4
```

Чтобы разрешить присваивание, как оно было изначально задано, *callback* может возвратить *newval*. Например:

```
function rectWatch (prop, oldval, newval, userData) {
  // Возвратить предлагаемое новое значение в неизмененном виде
  return newval;
}
```

Функция обратного вызова для контрольной точки, которая возвращает *newval* и потому не влияет, в конечном итоге, на присваивание свойству значения, все же может оказаться полезной. Поместив в функцию обратного вызова команду *trace( )* или точку останова отладчика, можно отлаживать, регистрировать в журнале или наблюдать любые изменения свойства.

В других случаях с помощью *callback* и *newval* можно вычислять или фильтровать окончательные значения свойства (например, вместо значений, выходящих за границы допустимого диапазона, задавать допустимые значения). Функция *callback* может выполнять и другие задачи, связанные с присваиванием свойству значений. Представим себе, например, игру, в которой кнопка Buy (купить) выключается, когда у игрока нет денег. Контролируя свойство cash (наличность) игрока, можно включать и выключать кнопку Buy в зависимости от текущего состояния средств игрока:

```
// Упрощенный класс GamePlayer
function GamePlayer (startingMoney) {
  // Добавим контрольную точку (контрольные точки можно добавлять до объявления свойств)
```

```
  this.watch("cash", this.assignCash);
  // Присвоим начальное значение свойству cash
  this.cash = startingMoney;
}

// Метод: assignCash()
GamePlayer.prototype.assignCash = function (prop, oldval, newval) {
  trace("Assigning cash: " + newval);
  // Если у игрока нет средств...
  if (newval == 0) {
    // ...выключить кнопку Buy
    trace("Player has no money. Disabling Buy button.");
    gameGUI.buyButton.enabled = false;
  } else if (newval > 0) {
    // ...в противном случае включить кнопку Buy
    trace("Player has some money. Enabling Buy button.");
    gameGUI.buyButton.enabled = true;
  }

  // Присвоить новое значение свойству cash
  return newval;
}

// Создать игрока и дать ему для начала 10 денежных единиц
p1 = new GamePlayer(10);

// Отобрать у игрока все деньги
p1.cash = 0;

// ВЫВОД:
Assigning cash: 10
Player has some money. Enabling Buy button.
Assigning cash: 0
Player has no money. Disabling Buy button.
```

Контрольную точку можно устанавливать, когда свойство еще не существует, и она не исчезнет, даже если удалить контролируемое свойство. Если свойство создается заново, срабатывает контрольная точка. Для прекращения контроля за свойством (то есть удаления контрольной точки) применяется *Object.unwatch( )*.

Не всякие свойства могут контролироваться, тут действуют следующие ограничения:

- Свойства, унаследованные через прототип объекта, не могут контролироваться. Контролируемое свойство должно быть назначено непосредственно в `someObject` (метод *Object.hasOwnProperty( )* позволяет определить, является свойство прямым или унаследованным).
- Нельзя контролировать свойства, доступные во Flash 4 (`_x`, `_y`, `_width`, `_height` и т. д.). См. раздел «Доступность» в статьях данного справочника, посвященных каждому свойству.
- Нельзя контролировать свойства доступа, созданные с помощью *Object.addProperty( )*. Свойства доступа, внутренне определенные во Flash 6 ActionScript, перечислены в табл. С.15. Обратите внимание, что объекты *Camera*, *Microphone*, *NetStream* и *Video* относятся к Macromedia Flash Communication Server MX (Comm Server) и не освещаются в данной книге (см. дополнительные ресурсы на *http://www.macromedia.com/software/ flashcom/* и в приложении А).

Таблица C.15. Свойства доступа (Flash 6), которые не могут контролироваться

Объект	Свойства доступа
Button	tabIndex
Camera	Все свойства
Microphone	Все свойства
MovieClip	tabIndex
NetStream	Все свойства
Sound	duration position
Stage	Все свойства
TextField	Все свойства
TextFormat	Все свойства
Video	Все свойства
XML	contentType docTypeDecl ignoreWhite loaded status xmlDecl
XMLnode	attributes childNodes firstChild lastChild nextSibling nodeName nodeType nodeValue parentNode previousSibling

См. также *Object.addProperty()*, `Object.__proto__`, *Object.unwatch()*

parseFloat(), глобальная функция

Flash 5

извлекает из строки число с плавающей точкой

`parseFloat(stringExpression)`

Аргументы

stringExpression

Строка, из которой должно быть извлечено число с плавающей точкой.

Возвращаемое значение

Извлеченное число с плавающей точкой. Или специальное числовое значение `NaN`, если извлечение оказалось неудачным.

Описание

Функция *parseFloat( )* преобразует строку в число с плавающей точкой (число с дробной частью). Она работает только со строками, содержащими допустимые строковые представления числа с плавающей точкой, а в противном случае возвращает NaN. Строка может иметь следующий формат:

- Допустимый начальный символ (знаки +, -, цифры от 0 до 9, буквы e, E и точка, допускается наличие необязательного ведущего непустого символа,[1] который игнорируется).
- Необязательный символ знака + или -.
- Хотя бы одна из цифр от 0 до 9 и необязательная десятичная точка.
- Необязательный показатель степени, начинающийся с e или E, за которым следует целочисленный показатель степени, который сам может содержать необязательный знак числа (+ или -), например «10e-3».

Замыкающие символы, которые не могут быть включены в описанный числовой формат, игнорируются, но не обязательно влекут возврат NaN. Число, извлекаемое из переданной строки, начинается с первого символа строки, отличного от пробела, и заканчивается символом, предшествующим первому недопустимому символу.

Применение

Поскольку данные, вводимые пользователем в текстовые поля, всегда принадлежат типу данных *string*, функция *parseFloat( )* часто применяется для извлечения числовых данных из текста, вводимого пользователем. Обратите внимание, что *parseFloat( )* может извлекать числа из строк, содержащих как числа, так и нечисловые символы, а функция *Number( )* на это не способна.

Примеры

```
parseFloat("14.5 apples");     // Выдает 14.5
parseFloat(".123");            // Выдает 0.123
var x = "15, 4, 23, 9";
parseFloat(x);                 // Выдает 15
```

См. также

isNaN( ), NaN, *Number( )*, *parseInt( )*; «Явное преобразование типа данных», глава 3 и «Целые числа и числа с плавающей точкой», глава 4

parseInt(), глобальная функция

Flash 5

извлекает из строки целое число или преобразует число к основанию 10

parseInt(*stringExpression*)
parseInt(*stringExpression*, *radix*)

Аргументы

stringExpression Строка, из которой должно быть извлечено целое число.

radix Необязательное целое число от 2 до 36, задающее основание системы счисления целого числа, содержащегося в *stringExpression*. Если этот аргумент не задан, основание, используемое по умолчанию, зависит от содержимого *stringExpression* (как описано ниже).

[1] Любой символ, кроме пробела и символа табуляции (\t). – *Примеч. ред.*

Возвращаемое значение

Извлеченное целое число в десятичной системе, независимо от исходного основания. Если извлечение было неудачным, возвращается специальное числовое значение `NaN`.

Описание

Функция *parseInt()* преобразует строковое выражение в целое число. Она работает только с допустимым строковым представлением целого числа в указанной системе счисления. Строка должна иметь следующий формат:

- Допустимый начальный символ (знаки +, - и цифры от 0 до 9, допускается наличие необязательного ведущего непустого символа, который игнорируется). Символы алфавита (буквы a–z и A–Z) могут присутствовать, если они представляют цифры по заданному основанию *radix*. Например, буквы a–f и A–F допустимы, если *radix* имеет значение 16 (шестнадцатеричная система), но не допустимы, если *radix* равно 10 (десятичная система).
- Ведущий 0 и последующий *X* или *x*, например «0x», что означает шестнадцатеричное число.
- Необязательный знак + или -.
- Одна или более цифр, допустимых в системе счисления с указанным основанием *radix*.
- Замыкающие символы, которые не могут быть включены в описанный числовой формат, игнорируются, но не обязательно приводят к тому, что возвращается `NaN`.

Число, полученное из переданной строки, начинается с первого символа строки, отличного от пробела, и оканчивается символом, предшествующим первому символу, не являющемуся допустимой цифрой в заданном *radix*. Например, в системе по основанию 10 буква *F* не является допустимой цифрой, поэтому следующее выражение возвращает число 2:

```
parseInt("2F", 10);
```

Но в системе по основанию 16 (шестнадцатеричной) символы *A*, *B*, *C*, *D*, *E* и *F* представляют собой допустимые цифры, поэтому следующее выражение возвращает число 47:

```
parseInt("2F", 16);  // шестнадцатеричное 2F дает 47 по основанию 10
```

Аргумент *radix* функции *parseInt()* указывает основание числа, имеющегося в строке. Иными словами, с помощью аргумента *radix* можно указать интерпретатору обработать строку как число с основанием 16 или обработать строку как число с основанием 2.

Функция *parseInt()* также интерпретирует префикс `0x`, указывающий на шестнадцатеричное число (как если бы был задан *radix* 16), и ведущий 0 (ноль), указывающий на восьмеричное число (как если бы был указан *radix* 8):

```
parseInt("0xFF");    // Анализируется как 16-ричное, дает 255
parseInt("FF", 16);  // Анализируется как 16-ричное, дает 255
parseInt("FF", 10);  // Анализируется как десятичное (неявное основание 10), дает NaN
parseInt("0377");    // Анализируется как 8-ричное, дает 255 = (3 * 64) + (7 * 8) + (7 * 1)
parseInt("377", 8);  // Анализируется как 8-ричное, дает 255
```

Явное указание основания имеет приоритет над неявным:

```
parseInt ("0xFF", 10)   // Анализируется как десятичное, дает 0
parseInt ("0x15", 10)   // Анализируется как десятичное, дает 0 (не 15 и не 21)
parseInt ("0377", 10)   // Анализируется как десятичное, дает 377
```

Обратите внимание, что функция *parseInt( )* извлекает только целые значения, в отличие от *parseFloat( )*, которая может извлекать и дробные значения, но используется только с десятичными числами.

Пример

В основном *parseInt( )* применяется для извлечения целых чисел из строки, в которой содержатся как числа, так и текст, или для удаления десятичной дроби в числе (аналогично *Math.floor( )*).

```
parseInt("Wow, 20 people were there");    // Дает NaN
parseInt("20 people were there");         // Дает 20
parseInt("1001", 2);                      // Дает 9 (двоичное 1001)
parseInt(1.5);                            // Дает 1 (число 1.5 преобразуется в строку
                                          // "1.5" перед выполнением операции parseInt)
```

См. также

Math.floor( ), NaN, *parseFloat( )*; «Явное преобразование типа данных», глава 3 и «Целые числа и числа с плавающей точкой», глава 4

play(), глобальная функция

Flash 2

начинает последовательное воспроизведение кадров фильма

```
play()
```

Описание

При вызове функции *play( )* начинается последовательное воспроизведение кадров текущей временной диаграммы. «Текущей» называется временная диаграмма главного фильма или клипа, из которой вызвана функция *play( )*. Кадры воспроизводятся с частотой, определяемой частотой смены кадров (FPS), установленной для фильма в целом в свойствах фильма Movie Properties (Modify → Movie → Frame Rate). Если фильм уже воспроизводится, то вызов *play( )* не имеет никаких последствий.

Начавшееся воспроизведение фильма или клипа продолжается, пока его специально не остановит вызов другой функции. Все клипы зацикливаются (снова начинают воспроизведение с кадра 1), когда головка воспроизведения достигает конца временной диаграммы. Однако в броузере главные фильмы зацикливаются только тогда, когда в коде, встраивающем фильм в страницу HTML, указана необходимость зацикливания с помощью атрибута LOOP. Если для встраивания фильма в страницу HTML применяется команда Publish, то атрибут LOOP устанавливается при выборе File → Publish Settings → HTML → Playback → Loop. Дополнительные сведения см. в приложении H.

Пример

Воспроизвести текущую временную диаграмму:

```
play()
```

Для того чтобы начать воспроизведение другого клипа, пользуйтесь функцией *MovieClip.play( )* вместо прежнего действия Tell Target.

Упражнение: Напишите рекурсивную функцию с именем *allPlay( )*, которая воспроизводит текущую временную диаграмму и рекурсивно воспроизводит все вложенные клипы, содержащиеся в текущей временной диаграмме.

См. также

gotoAndPlay( ), *MovieClip.play( )*, *stop( )*

prevFrame(), глобальная функция

Flash 2

перемещает головку воспроизведения фильма на один кадр назад и останавливает ее

```
prevFrame()
```

Описание

Функция *prevFrame( )* перемещает головку воспроизведения текущей временной диаграммы назад на один кадр и останавливает ее там. Под «текущей временной диаграммой» понимается временная диаграмма главного фильма или клипа, откуда вызвана функция *prevFrame( )*. Функция *prevFrame( )* – это синоним *gotoAndStop(_currentFrame – 1);*. При вызове из первого кадра временной диаграммы *prevFrame( )* просто останавливает в этом кадре головку воспроизведения, если только текущей сцене не предшествует другая, и тогда *prevFrame( )* перемещает головку воспроизведения в последний кадр предыдущей сцены.

См. также

gotoAndStop( ), *MovieClip.prevFrame( )*, *nextFrame( )*, *prevScene( )*, *stop( )*

prevScene(), глобальная функция

Flash 2

перемещает головку воспроизведения в кадр 1 предыдущей сцены

```
prevScene()
```

Описание

Функция *prevScene( )* перемещает главную головку воспроизведения фильма в кадр 1 сцены, предшествующей текущей, и останавливает главную временную диаграмму фильма. Под «текущей» следует понимать сцену, из которой вызвана функция *prevScene( )*. Чтобы оказать какое-либо воздействие, она должна вызываться на главной временной диаграмме сцены. Иными словами, *prevScene( )* не работает внутри клипа или обработчика события *onClipEvent( )*. При вызове из первой сцены фильма *prevScene( )* перемещает головку воспроизведения в кадр 1 этой сцены и останавливает воспроизведение фильма.

Не рекомендуется применять сцены при работе с фильмами, активно задействующими ActionScript. В отличие от клипов, сцены не представляются объектами, и ими нельзя непосредственно управлять с помощью большинства встроенных функций. Как правило, вместо сцен Flash лучше использовать метки и клипы в качестве псевдосцен на временной диаграмме.

См. также

nextScene( ), *prevFrame( )*

print(), глобальная функция

Flash 5

выводит на печать кадры фильма или клипа с использованием векторной графики

```
print(target, boundingBox)
```

Аргументы

target Строка или ссылка, указывающая путь к клипу или уровню документа, которые должны быть напечатаны (в строковом контексте ссылки преобразуются в пути).

boundingBox

Строка, указывающая способ кадрирования – определения границ кадров `target` при выводе на печать. Кадрирование определяется ограничивающей рамкой, которая представляет всю печатаемую страницу. Область `target`, включаемую в печатаемую страницу, можно задать с помощью одного из трех допустимых для `boundingBox` значений (которое должно быть задано как строковый литерал):

«bframe»

Рамка границ для каждого печатаемого кадра устанавливается индивидуально, в соответствии с размером содержимого каждого кадра. Поэтому содержимое каждого печатаемого кадра масштабируется, чтобы заполнить всю печатную страницу. При этом размер печатаемого изображения максимален.

«bmax»

Области, занимаемые содержимым всех печатаемых кадров, объединяются, чтобы образовать общую рамку границ. Содержимое каждого печатаемого кадра масштабируется и располагается на печатной странице относительно общей рамки границ. Таким образом, каждый кадр сохраняет свой размер относительно других кадров, но все кадры не обязательно заполняют печатную страницу.

«bmovie»

Рамка границ для всех печатаемых кадров устанавливается по размеру одного выделенного кадра в клипе `target`. Содержимое печатаемых кадров, из-за размеров выходящее из выделенного кадра, кадрируется по рамке границ этого выделенного кадра. Содержимое печатаемых кадров, по размерам меньшее, чем выделенный кадр, не масштабируется. Для того чтобы выделить кадр, определяющий ограничивающую рамку, ему надо присвоить метку `#b`.

Описание

Печать фильма Flash с помощью встроенной функции печати броузера File → Print может происходить по-разному, и часто в результате получаются отпечатки низкого качества, так как при этом не применяется векторная печать. В некоторых броузерах фильмы вообще не печатаются, и большинство броузеров не печатает одновременно больше одного кадра. С помощью функции *print( )* можно печатать содержимое одного или нескольких кадров фильма точно и с высоким качеством прямо из Flash. По умолчанию *print( )* отправляет на печать все кадры с временной диаграммы `target`, по одному кадру на страницу, обрезая или увеличивая их в зависимости от значения аргумента `boundingBox`. Для того чтобы напечатать только некоторые кадры, им надо присвоить метку `#P`.

Функция *print( )* посылает на PostScript-принтеры непосредственно векторную графику, а на принтеры без поддержки PostScript – графику, преобразованную в растровые изображения. Работа *print( )* основана на применении векторов, поэтому с ее помощью нельзя печатать фильмы, содержащие преобразования прозрачности или цвета. Для печати фильмов с цветовыми эффектами следует применять функцию *printAsBitmap( )*.

Применение

Во Flash Player 4 функции печати доступны через действие Get Url Action (не реализовано до версии 4.0.20.0). За подробностями обращайтесь к Macromedia Flash Printing SDK, который можно найти на странице:

http://www.macromedia.com/software/flash/open/webprinting/authoring.html

Пример

```
// Напечатать все кадры главной временной диаграммы,
// задав размер каждого из них так, чтобы он занимал всю страницу
print("_root", "bframe");

// Напечатать все кадры главной временной диаграммы,
// изменив размер каждого соответственно общему размеру всех кадров
print("_root", "bmax");
```

При щелчке по кнопке `print_btn` Flash выводит на печать все кадры на временной диаграмме родительского объекта кнопки, кадрированные по ограничивающей рамке кадра с меткой `#b`, размером во всю страницу:

```
print_btn.onRelease = function () {
  print(this._parent, "bmovie");
}
```

См. также *getURL( ), printAsBitmap( ), printAsBitmapNum( ), printNum( )*

printAsBitmap(), глобальная функция — Flash 5

печатает кадры фильма или клипа как растровые изображения

`printAsBitmap(target, boundingBox)`

Аргументы

target Строка или ссылка, указывающая путь к клипу или уровню документа, которые должны быть напечатаны (в строковом контексте ссылки преобразуются в пути).

boundingBox Строка, имеющая значение `"bframe"`, `"bmax"` или `"bmovie"` и указывающая способ изменения границ или масштаба кадров *target* при выводе на печать, как описано выше для *print( )*.

Описание

Функция *printAsBitmap( )* по своему действию идентична *print( )* за исключением того, что выводит на принтер не векторы, а растровое содержимое. В результате она может успешно печатать фильмы с преобразованием прозрачности и цвета, но дает результаты более низкого качества для векторной графики.

Применение

См. замечания по применению функции *print( )*.

См. также *print( ), printAsBitmapNum( ), printNum( )*

printAsBitmapNum(), глобальная функция — Flash 5

печатает кадры уровня документа как растровые изображения

`printAsBitmapNum(level, boundingBox)`

Аргументы

level Неотрицательное целое число или выражение с неотрицательным целым значением, задающее уровень документа, который должен быть напечатан.

boundingBox Строка, имеющая значение `"bframe"`, `"bmax"` или `"bmovie"` и указывающая способ изменения границ или масштаба кадров `target`.

Описание

Функция *printAsBitmapNum( )* почти идентична *printAsBitmap( )*, но требует задания целевого уровня (`level`) для операции печати в виде числа, а не строки. Это означает, что *printAsBitmapNum( )* может печатать только уровни документов, а не клипы. Обычно она применяется, когда требуется динамически назначить выводимый на печать уровень фильма, например:

```
var x = 3;
printAsBitmapNum(x, "bmax");
```

Того же результата можно достичь с помощью конкатенации строк и обычной функции *printAsBitmap( )*:

```
printAsBitmap("_level" + x, "bmax");
```

Применение

См. замечания по применению функции *print( )*.

См. также

print( ), *printAsBitmap( )*, *printNum( )*

printNum(), глобальная функция

Flash 5

печатает кадры уровня документа с использованием векторов

`printNum(level, boundingBox)`

Аргументы

level Неотрицательное целое число или выражение с неотрицательным целым значением, задающее уровень документа, который должен быть напечатан.

boundingBox Строка, имеющая значение `"bframe"`, `"bmax"` или `"bmovie"` и указывающая способ изменения границ или масштаба кадров `target`.

Описание

Функция *printNum( )* почти идентична *print( )*, но требует задания целевого `level` (уровня) для операции печати в виде числа, а не строки. Это означает, что *printNum( )* может печатать только уровни документов, а не клипы. Обычно она применяется, когда требуется динамически назначить выводимый на печать уровень фильма, например:

```
var x = 3;
printNum(x, "bmax");
```

Того же результата можно достичь с помощью конкатенации строк и обычной функции *print( )*:

```
print("_level" + x, "bmax");
```

Применение

См. замечания по применению функции *print( )*.

См. также

print( ), *printAsBitmap( )*, *printAsBitmapNum( )*

_quality, глобальное свойство

Flash 5

задает качество вывода в проигрывателе **чтение/запись**

`_quality`

Описание

Свойство `_quality` содержит строку, определяющую качество вывода Flash Player:

«LOW»

Низкое качество. Сглаживание (antialiasing) не выполняется ни для растровой, ни для векторной графики.

«AUTOLOW»

Начать с низкого качества, но перейти на высокое, если проигрыватель определит, что вычислительных ресурсов для этого достаточно.

«MEDIUM»

Среднее качество. Умеренное сглаживание векторной графики с помощью сетки 2×2 пиксела.

«AUTOHIGH»

Начать с высокого качества, но перейти на низкое, если проигрыватель определит, что вычислительных ресурсов недостаточно (исходя из частоты смены кадров).

«HIGH»

Высокое качество (по умолчанию). Векторная графика подвергается сильному сглаживанию с помощью сетки 4×4 пиксела. Растровая графика сглаживается, когда нет анимации.

«BEST»

Высшее качество. Векторная графика подвергается сильному сглаживанию с помощью сетки 4×4 пиксела. Растровая графика сглаживается всегда.

Чем ниже качество вывода, тем меньше требуется вычислений для воспроизведения кадра или фильма, и это приводит к повышению производительности. Обычно в большинстве фильмов `_quality` устанавливается в `"HIGH"`.

Пример

В данном примере устанавливается наилучшее возможное качество вывода фильма (то есть самое медленное воспроизведение):

```
_quality = "BEST";
```

См. также `_highQuality`, *toggleHighQuality( )*

random(), глобальная функция

Flash 4; устарела во Flash 5 с появлением Math.random()

генерирует случайное число

`random(`*number*`)`

Аргументы

number Положительное целое число.

Возвращаемое значение

Случайное целое число от 0 до *number* - 1 включительно.

Описание

Устаревшая сейчас функция *random( )* применялась во Flash 4 для генерирования случайных чисел. Она была сохранена только для обеспечения обратной совместимости. Во Flash 5 и выше следует применять метод *Math.random( )*. Обратите внимание, что *random( )* генерировала целые числа от 0 до number - 1, а *Math.random( )* генерирует числа с плавающей точкой от 0,0 до 0,999.

См. также *Math.random( )*

removeMovieClip(), глобальная функция

Flash 4

удаляет клип из Flash Player

removeMovieClip(*target*)

Аргументы

target Строка или ссылка, задающая путь к экземпляру клипа, подлежащему удалению из проигрывателя (в строковом контексте ссылки преобразуются в пути).

Описание

Функция *removeMovieClip( )* удаляет указанный клип из проигрывателя, не оставляя следов содержимого или оболочки клипа. Последующие ссылки на клип или его переменные и свойства возвращают undefined.

Функцию *removeMovieClip( )* можно применять только с экземплярами клипов, созданными первоначально посредством *duplicateMovieClip( )*, *attachMovie( )* или *createEmptyMovieClip( )*. На клипы с отрицательной глубиной (обычно созданные на этапе разработки) влияния не оказывает. Чтобы удалить клип с отрицательной глубиной, сначала поменяйте ее на положительную:

```
theClip_mc.swapDepths(1234);   // Задаем безопасное значение глубины.
theClip_mc.removeMovieClip();  // Удаляем клип.
```

Если на один и тот же клип ссылаются несколько переменных, то *removeMovieClip( )* удаляет клип со сцены, но другие ссылки на него сохраняются. Это означает, что клип сохраняется как объект данных, пока не будут удалены все ссылки на него. Например:

```
// Создать новый клип clipInstance_mc из символа библиотеки clipSymbol.
// Сохранить вспомогательную ссылку на clipInstance_mc в переменной c.
var c = this.attachMovie("clipSymbol", "clipInstance_mc", 1);
// Сейчас существуют две ссылки на клип: одна в c, а другая в clipInstance_mc.

// Удалить clipInstance_mc
clipInstance_mc.removeMovieClip();
// Проверить, существует ли clipInstance_mc по-прежнему...
trace(typeof clipInstance_mc);  // Нет. Выводит: undefined
// Проверить, существует ли ссылка на clipInstance_mc в c ....
trace(typeof c);                // Да! Выводит: movieclip (ссылка сохранилась!)
```

Для того чтобы полностью удалить все следы клипа, надо удалить все оставшиеся переменные, в которых есть ссылка на него. Продолжим предшествующий пример:

```
// Удалить ссылку в c.
delete c;
trace(typeof c);   // Выводит: undefined (теперь его действительно нет)
```

См. также

attachMovie( ), *duplicateMovieClip( )*, *MovieClip( ).removeMovieClip( )*, *MovieClip.swapDepths( )*, *unloadMovie( )*, «Удаление экземпляров клипов и главных фильмов», глава 13

_root, глобальное свойство

Flash 5 (то же, что «/» в фильмах Flash 4)

содержит ссылку на главную временную диаграмму фильма на текущем уровне — **только для чтения**

```
_root
```

Описание

Свойство `_root` хранит ссылку на главную временную диаграмму текущего уровня документа и применяется для вызова методов главного фильма или получения и задания свойств главного фильма, например:

```
_root.play();        // Воспроизводить главную временную диаграмму
_root._alpha = 40;   // Сделать полупрозрачным весь фильм
```

Свойство `_root` позволяет также ссылаться на вложенные клипы с помощью синтаксиса точки, например:

```
_root.myClip.play();
_root.shapes.square._visible = false;
```

Свойство `_root` предоставляет доступ к клипу в абсолютном выражении. Иными словами, ссылка, начинающаяся с `_root`, остается действующей (и неизменной) в любом месте фильма. Учтите, однако, что `_root` не всегда оказывается синонимом для `_level`*n* (главной временной диаграммы заданного уровня документа) или `_global` (пространства глобальных переменных). Создание глобальных переменных рассмотрено в описании `_global`.

При использовании в файле *.swf*, загруженном в клип, свойство `_root` изменяет значение. Подробнее об этом рассказано в разделе «Свойства _root и _parent» главы 2.

См. также

`_level`*n*; `_parent`; «Доступ к переменным других временных диаграмм», глава 2, «Ссылка на главный фильм с помощью `_root` и `_level`n», глава 13

scroll, свойство

Flash 4

текущая верхняя строка, выводимая в текстовом поле — **чтение/запись**

```
textFieldVar.scroll
```

Описание

Свойство `scroll` текстового поля можно как читать, так и записывать. Доступ к этому унаследованному свойству осуществляется как к свойству *textFieldVar* — свойству идентификатора, связанного с текстовым полем на экране. Идентификатор *text-*

FieldVar задается в окне Text Options (во Flash 5) или в диалоговом окне Text Field Properties (во Flash 4). Во Flash 6 вместо него следует применять объектно-ориентированное свойство *TextField*.scroll, где *TextField* – это экземпляр класса *TextField*. Дополнительные сведения см. в описании *TextField*.variable.

Свойство текстового поля scroll задает номер строки, выводимой в данный момент в качестве первой строки видимой области поля. При установке значения scroll происходит прокрутка текстового поля, благодаря чему переданное число становится номером верхней строки в видимой области поля.

Обычно свойство scroll применяется совместно с maxscroll для создания интерфейсов с прокруткой текста, как описывается в *TextField*.scroll.

Ошибки реализации

При работе с текстовыми полями и встроенными шрифтами во Flash Player 5.0.30.0 свойство scroll может стать причиной вывода части текста за пределами ограничивающего прямоугольника поля. Некоторые части изображения могут не удаляться при прокрутке текста в поле. Этот недуг «лечится» применением маски над слоем текстового поля. Ошибка была исправлена во Flash Player 5.0.41.0 и следующих.

Пример

```
// Записать в x индекс верхней строчки, отображаемой в myField
var x = myField.scroll;
// Прокрутка текста в myField вниз на одну строку
myField.scroll++;
```

См. также maxscroll; *TextField*.scroll, *TextField*.variable

Selection, объект

Flash 5; во Flash 6 добавлена поддержка кнопок и клипов

управляет выделением в текстовых полях и фокусом ввода фильма

Selection.*methodName*()

Методы

addListener( )

Регистрирует объект для получения уведомлений о событии *onSetFocus( )*.

getBeginIndex( )

Возвращает индекс первого выбранного символа в текстовом поле, имеющем фокус.

getCaretIndex( )

Возвращает индекс точки ввода в текстовом поле.

getEndIndex( )

Возвращает индекс символа в текстовом поле, следующего за выделением.

getFocus( )

Указывает текстовое поле, кнопку или клип, которые в данное время находятся в фокусе.

removeListener( )

Прекращает получение уведомлений о событии указанным приемником.

setFocus( )

Помещает фокус ввода в заданные текстовое поле, кнопку или клип.

setSelection( )

Выделяет символы в текстовом поле, имеющем в данный момент фокус ввода.

События приемника

onSetFocus( )

Происходит при изменении фокуса ввода.

Описание

Объект *Selection* применяется для управления взаимодействием пользователя с текстовыми полями, захвата участков текстовых полей и обнаружения фокуса ввода у кнопок, клипов и текстовых полей. *Selection* – это готовый объект, а не класс, поэтому к его методам можно обращаться прямо:

```
Selection.setFocus(input_txt);  // Поместить фокус ввода в текстовое поле input_txt
```

Методы объекта *Selection* для действий с символами всегда неявно относятся к текстовому полю, обладающему фокусом ввода с клавиатуры, поэтому нужен только один объект *Selection*. Поэтому, например, обращение к *Selection.getBeginIndex( )* происходит непосредственно через объект *Selection*, а не экземпляр класса *TextField*.

Методы объекта *Selection* могут устанавливать точку ввода и выделять или получать содержимое текстового поля. Эти возможности добавляют тонкое, но важное поведение в системы ввода данных пользователем. Например, в экране регистрации можно предложить пользователю ввести свое имя, переместив точку ввода в текстовое поле ввода имени. Можно высветить ошибку в форме, выделив неверный текст. Такой контроль важен также в приложениях типа обработки текста.

Позиции символов в текстовом поле нумеруются индексами, начинающимися с нуля: первый символ имеет индекс 0, второй – индекс 1 и т. д. Индексирование символов подробно описано в разделе «Индексы символов» главы 4.

Во Flash 6 моделировать операции «вырезать», «копировать» и «вставить» в текстовом поле можно с помощью метода *TextField.replaceSel( )*. Кроме того, во Flash Player 6 этим операциям назначены клавиатурные сокращения (Ctrl-X, Ctrl-C и Ctrl-V в Windows; Cmd-X, Cmd-C и Cmd-V на Макинтошах). Щелчок правой кнопкой мыши (в Windows) или щелчок кнопкой мыши при нажатой клавише Ctrl (на Макинтоше) обеспечивает доступ к командам «вырезать», «копировать» и «вставить» во Flash Player 5.

Объект *Selection* может также управлять клавиатурным фокусом (известным еще как *фокус ввода*) в фильме и получать информацию о фокусе ввода. Объект, обладающий фокусом ввода с клавиатуры, получает все нажатия клавиш. Например, когда в текстовом поле находится точка ввода текста, оно располагает фокусом ввода с клавиатуры, и все нажатия клавиш отображаются в нем в виде символов. Когда фокус ввода получает кнопка, ее можно активизировать нажатием клавиши Enter или пробела. А получивший фокус ввода клип может отвечать на нажатия клавиш через обработчики событий *onKeyDown( )*/*onKeyUp( )*. Кнопки и клипы получают фокус ввода с клавиатуры либо в результате нажатия пользователем клавиши Tab (не щелчка мышью!), либо программно с помощью *Selection.setFocus( )*. По умолчанию кнопка или клип, находящиеся в фокусе, заключаются в желтую прямоугольную рамку (см. глобальное свойство _focusrect). Изменение фокуса можно перехватывать глобально с помощью события *Selection.onSetFocus( )* или индивидуально с помощью обработчиков событий *onSetFocus( )*/ *onKillFocus( )* для *Button*, *MovieClip* и *TextField*. Узнать, какой элемент находится в данное время в фокусе, можно с помощью *Selection.getFocus( )*.

Применение

Щелчок по кнопке Submit автоматически снимает фокус с текстового поля, имевшего фокус перед этим. Для того чтобы перехватить выделение перед потерей фокуса, можно воспользоваться обработчиком события rollover кнопки Submit, поскольку это событие происходит раньше нажатия кнопки. Например:

```
submit_btn.onRollOver = function () {
  // Запомнить, какое поле было в фокусе перед нажатием кнопки.
  focusedField = Selection.getFocus();

  // Запомнить, какие символы были выделены.
  beginIdx = focusedField.getBeginIndex();
  endIdx =   focusedField.getEndIndex();
}
```

Запомнив в переменной поле, находившееся в фокусе, можно позднее выполнять действия над выделенными в нем символами или возвратить полю фокус после обнаружения ошибок в данных, введенных пользователем.

См. также Класс *Button*, `_focusrect`, класс *MovieClip*, класс *TextField*

Selection.addListener(), метод

Flash 6

регистрирует объект для получения уведомлений о событии onSetFocus()

`Selection.addListener(`*`listener`*`)`

Аргументы

listener Объект, в котором определен метод *onSetFocus( )*.

Описание

Метод *addListener( )* помещает *listener* в список объектов, оповещаемых, когда происходит событие *onSetFocus( )* объекта *Selection*. Аргумент *listener* – это объект любого класса, в котором определен метод с именем *onSetFocus*; метод вызывается, когда происходит событие *onSetFocus( )* для *Selection*. Прекратить получение событий объектом *listener* можно с помощью метода *Selection.removeListener( )*.

На каждое событие *onSetFocus( )* могут независимо реагировать несколько объектов. Если в *Selection* зарегистрировано несколько объектов, их методы вызываются в том порядке, в котором добавлялись приемники.

Пример кода *Selection.addListener( )* приведен в описании *Selection.onSetFocus( )*. Общие сведения о работе приемников и функций обратного вызова см. в главе 10.

См. также *Selection.onSetFocus( )*, *Selection.removeListener( )*; глава 10

Selection.getBeginIndex(), метод

Flash 5

возвращает индекс первого выделенного символа в текстовом поле, имеющем фокус

`Selection.getBeginIndex()`

Возвращаемое значение

Индекс первого символа в текущем выделении (подсвеченном блоке текста). Если ни у какого текстового поля нет фокуса ввода с клавиатуры, функция возвращает –1.

Если текстовое поле в фокусе, но никакие символы не выделены, то возвращается значение *Selection.getCaretIndex( )*.

Описание

Метод *getBeginIndex( )* определяет начало выделения. Чтобы определить диапазон символов, выделенных в данное время, следует вызвать две функции: *getBeginIndex( )* и *getEndIndex( )*.

Пример

В следующем примере создается строка, представляющая выделенные в данный момент символы и выводимая в окно Output:

```
var firstChar = Selection.getBeginIndex();
var lastChar  = Selection.getEndIndex();
// eval() требуется потому, что Selection.getFocus() возвращает строку
var currentSelection =
    eval(Selection.getFocus()).text.substring(firstChar, lastChar);
trace(currentSelection);
```

Следующий код расширяет текущее выделение на один символ влево:

```
Selection.setSelection(Selection.getBeginIndex() - 1, Selection.getEndIndex());
```

См. также

Selection.getCaretIndex( ), *Selection.getEndIndex( )*, *TextField.replaceSel( )*

Selection.getCaretIndex(), метод

Flash 5

возвращает индекс точки ввода в текстовом поле

```
Selection.getCaretIndex()
```

Возвращаемое значение

Индекс точки ввода в текстовом поле, находящемся в фокусе. Если фокуса ввода нет ни у одного текстового поля, возвращается –1. Если текстовое поле с фокусом пусто, возвращается 0.

Описание

Метод *getCaretIndex( )* указывает точку ввода (то есть местонахождение курсора) в текстовом поле. Когда текстовое поле получает фокус ввода с клавиатуры, курсор выглядит, как буква «I». Для задания места точки ввода следует применять *setSelection( )*.

Пример

Метод *getCaretIndex( )* возвращает –1, если ни у одного текстового поля нет фокуса, поэтому, проверив *getCaretIndex( )* на равенство –1, можно узнать, есть ли фокус у какого-либо поля.

В следующем примере проверяется, есть ли фокус ввода с клавиатуры у какого-нибудь текстового поля.

```
if (Selection.getCaretIndex() == -1) {
  trace("No field has focus");
}
```

Упражнение: Перепишите предыдущий пример в виде булевой функции *isTextFieldActive( )*, которая возвращает `true`, когда у текстового поля есть фокус, и `false` в противном случае. Модифицируйте ее, чтобы она принимала необязательный идентификатор текстового поля и проверяла, активно ли указанное текстовое поле. (Подсказка: см. *Selection.getFocus( )*.)

См. также *Selection.getFocus( )*, *Selection.setSelection( )*

Selection.getEndIndex(), метод

Flash 5

получить индекс символа, следующего за выделением в текстовом поле

```
Selection.getEndIndex()
```

Возвращаемое значение

Индекс символа, расположенного за последним символом в текущем выделении (подсвеченном блоке текста). Если фокуса нет ни у одного текстового поля, возвращается –1. Если текстовое поле находится в фокусе, но не выделены никакие символы, возвращается значение *Selection.getCaretIndex( )*.

Описание

Метод *getEndIndex( )* указывает на конец выделения. Чтобы определить диапазон символов, выбранных в данный момент, следует вызвать оба метода: *getEndIndex( )* и *getBeginIndex( )*.

Пример

Следующий код расширяет текущее выделение на один символ вправо:

```
Selection.setSelection(Selection.getBeginIndex(), Selection.getEndIndex()+ 1);
```

См. также *Selection.getBeginIndex( )*, *Selection.getCaretIndex( )*

Selection.getFocus(), метод

Flash 5; во Flash 6 добавлена поддержка кнопок и клипов

определяет текстовое поле, в котором в данный момент находится курсор

```
Selection.getFocus()
```

Возвращаемое значение

Строка, указывающая полный путь к объекту, имеющему фокус ввода с клавиатуры (например, `"_level1.theField_txt"` или `"_level0.listBox_mc"`). Если нет кнопки, клипа или текстового поля, обладающих фокусом ввода с клавиатуры, возвращается `null`.

Описание

Метод *getFocus( )* определяет кнопку, клип или текстовое поле, в которых в настоящее время находится фокус (если такой объект есть), возвращая строку пути к нему. Преобразовать эту строку в ссылку на переменную можно с помощью *eval( )*. В следующем примере с помощью *getFocus( )* определяется имя поля, находящегося в фокусе, и с помощью *eval( )* это имя преобразуется в ссылку. Затем подсчитывается количество символов в этом текстовом поле (не длина выделения):

```
var numChars = eval(Selection.getFocus()).length;
```

Пример

Метод *getFocus( )* возвращает `null`, если не выбран ни один объект, поэтому, проверив *getFocus( )* на равенство `null`, можно определить, есть ли фокус у какого-либо объекта, например:

```
if (Selection.getFocus() == null) {
  trace("No button, movie clip, or text field has focus");
}
```

См. также *eval( )*, *Selection.getCaretIndex( )*, *Selection.setFocus( )*

Selection.onSetFocus(), событие приемника

Flash 6

вызывается при изменении фокуса ввода приемника

`Selection.onSetFocus(`*oldFocus*, *newFocus*`)`

Аргументы

oldFocus

Объект *Button*, *MovieClip* или *TextField*, который ранее находился в фокусе, или `null`, если ни один объект ранее не был в фокусе.

newFocus

Объект *Button*, *MovieClip* или *TextField*, в котором теперь находится фокус, или `null`, если ни у одного объекта фокуса сейчас нет.

Описание

Событие *onSetFocus( )* генерируется при изменении фокуса ввода в фильме, например при нажатии пользователем клавиши Tab для перемещения с поля формы к кнопке Submit. С помощью *onSetFocus( )* можно централизованно управлять фокусом. Другой альтернативой является управление фокусом для индивидуальных объектов с помощью обработчиков *onSetFocus( )* и *onKillFocus( )* для *Button*, *MovieClip* или класса *TextField*, которые действуют, только когда конкретный экземпляр получает или теряет фокус.

Чтобы ответить на событие *onSetFocus( )*, надо сначала присвоить функцию обратного вызова свойству `onSetFocus` некоторого объекта, а затем зарегистрировать этот объект в качестве приемника события. Функция обратного вызова *onSetFocus( )* должна принимать параметры *oldFocus* и *newFocus*, например:

```
// Создать общий объект
selListener = new Object();

// Присвоить функцию обратного вызова свойству onSetFocus объекта
selListener.onSetFocus = function (oldFocus, newFocus) {
  trace("Focus changed from " + oldFocus + " to " + newFocus);
}

// Зарегистрировать приемник в объекте Selection
Selection.addListener(selListener);
```

На событие *onSetFocus( )* может реагировать любое количество объектов любого класса. Прекратить получение событий объектом приемника можно с помощью метода *Selection.removeListener( )*:

```
Selection.removeListener(selListener);
```

Любой объект *TextField*, *MovieClip* или *Button* может получать и терять фокус в следующих случаях:

- Пользователь перемещается на объект или уходит с него с помощью клавиши Tab.
- Метод *Selection.setFocus( )* устанавливает или снимает фокус с объекта программным образом.

Кроме того, объект текстового поля может получать и терять фокус при щелчке мыши внутри поля или за его пределами. Однако на фокус оказывают влияние свойство `tabEnabled` объектов *Button*, *MovieClip* и *TextField* и свойство `selectable` объекта *TextField*. Подробные сведения о том, как и когда кнопки, клипы и текстовые поля могут получать фокус, можно найти в описании *onSetFocus( )* для каждого из этих классов.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

См. также

Button.enabled, *Button.onSetFocus( )*, *Button*.tabEnabled, *MovieClip.onSetFocus( )*, *MovieClip*.selectable, *MovieClip*.tabEnabled, *Selection.addListener( )*, *Selection.removeListener( )*, *Selection. setFocus( )*, *TextField.onSetFocus( )*, *TextField*.tabEnabled

Selection.removeListener(), метод

Flash 6

отменяет отправку уведомлений о событии указанному приемнику

`Selection.removeListener(listener)`

Аргументы

listener Объект, ранее зарегистрированный в качестве приемника в *Selection*.

Возвращаемое значение

Булево значение `true`, если *listener* найден и удален; `false`, если *listener* не был зарегистрирован в качестве приемника в *Selection*. Возвращаемое значение полезно при отладке кода, который удаляет приемники.

Описание

Метод *removeListener( )* прекращает получение объектом *listener* уведомлений о событиях *Selection*. Пользоваться им следует только после того, как *listener* зарегистрирован в качестве приемника событий с помощью *addListener( )*. Объект приемника не уничтожается и может многократно удаляться и добавляться. Приемники событий следует удалять, только если выполняемая ими задача больше не нужна.

См. также

Selection.addListener( ), *Selection.onSetFocus( )*; глава 10

Selection.setFocus(), метод

Flash 5; во Flash 6 добавлена поддержка кнопок и клипов

устанавливает фокус клавиатуры на указанную кнопку, клип или текстовое поле

`Selection.setFocus(instanceName)`

Аргументы

instanceName

Строка, представляющая путь к кнопке, клипу или текстовому полю, которые должны получить фокус. Поскольку в строковом контексте ссылки на экземпляры преобразуются в пути, *instanceName* может быть объектной ссылкой, например "_level0.listBox_mc" вместо _level0.listBox_mc.

Возвращаемое значение

Булева величина, показывающая, была ли попытка установить фокус удачной (true) или безуспешной (false). Попытка установки фокуса может быть неудачной, если *instanceName* не обнаружен или это не кнопка, не клип и не текстовое поле. Метод *setFocus( )* возвращает false и тогда, когда у *instanceName* отключен фокус, о чем говорится ниже в разделе «Описание».

Описание

Метод *setFocus( )* устанавливает фокус ввода с клавиатуры на кнопку, клип или текстовое поле, заданные *instanceName*.

Если *instanceName* представляет собой кнопку, то при получении фокуса отображается ее состояние Over, заключенное в желтую прямоугольную рамку (если только _focusrect не равно false). Имеющая фокус кнопка может быть активизирована нажатием клавиши Enter или пробела.

Если *instanceName* – это клип, то при получении фокуса воспроизводится кадр с меткой _over, если он существует, заключенный в желтую прямоугольную рамку (если только _focusrect не равно false). Находящийся в фокусе клип получает уведомления о событиях *onKeyDown( )*, *onKeyUp( )*, *onMouseDown( )* и *onMouseUp( )*.

Особое внимание надо обратить на следующее:

- Во Flash 6 перемещение мыши над сценой снимает фокус с кнопок и клипов, но не с текстовых полей.
- Главные временные диаграммы (например, _root и _level1) не могут получать фокус ввода с клавиатуры.
- Нельзя передать фокус клипу посредством *Selection.setFocus( )*, если только его свойство focusEnabled не равно true.
- Для того чтобы клип без событий кнопок мог получать фокус ввода с клавиатуры, его свойство tabEnabled должно иметь значение true.

Если *instanceName* представляет собой текстовое поле ввода, то при получении им фокуса в поле появляется курсор, позволяя вводить в него символы. Динамические и статические текстовые поля не могут получать фокус. Однако текстовое поле для ввода данных пользователем все же может получать фокус с помощью *setFocus( )*, если его свойства selectable и tabEnabled имеют значение false.

Обычно текстовое поле помещается в фокус, чтобы привлечь внимание к полю или разрешить пользователю ввести данные. С помощью метода *setFocus( )* удобно также указывать пользователю на ошибочно введенные им данные. Например, пользователь может неправильно ввести адрес электронной почты в электронный бланк. Чтобы предупредить пользователя об ошибке, можно установить фокус в текстовое поле с адресом электронной почты и предложить внести исправления. Во Flash 5 с помощью *setFocus( )* можно также задавать пользовательский порядок перемещения по полям формы посредством клавиши табуляции, как показано в следующем примере.

Во Flash 6 порядок табуляции следует задавать непосредственно с помощью свойства *TextField*.tabIndex.

Применение

Обратите внимание, что при установке фокуса в текстовое поле его содержимое автоматически выделяется. Вот так можно установить фокус, не выделяя никаких символов:

```
// Сначала установить фокус в theField
Selection.setFocus("theField_txt");

// Теперь поместить точку ввода в начало theField
Selection.setSelection(0, 0);

// Или в конец theField
Selection.setSelection(theField.length, theField.length);
```

При просмотре фильма в броузере текстовое поле, имеющее фокус, принимает текст только тогда, когда сам фильм Flash находится в фокусе (то есть если пользователь щелкнул в какой-то точке фильма во время воспроизведения). Сделайте так, чтобы фильм получил фокус, прежде чем просить пользователя ввести что-либо в текстовые поля. Один из способов состоит в том, чтобы поместить в начало фильма кнопку с надписью «Щелкнуть здесь, чтобы начать».

Пример

В следующем примере показано, как назначить пользовательский порядок обхода полей при табуляции в заполняемой форме. Этот пример относится только к Flash 5, но показывает, какие мощные средства контроля над вводом данных пользователем дает объект *Selection*. Соответствующий файл *.fla* можно загрузить из сетевого хранилища кода. Дополнительные сведения о перехвате клавиши Tab можно найти в описании *TextField*.tabIndex.

```
// Назначение порядка перехода по клавише табуляции
// КОД В КЛИПЕ, СОДЕРЖАЩЕМ ПОЛЯ ФОРМЫ
onClipEvent (load) {
  // Сохранить путь к клипу в строке. Позднее он будет
  // использован при вызове Selection.setFocus().
  path = targetPath(this);
  // Создать массив с именами текстовых полей, указанными в требуемом
  // для табуляции порядке. Полем по умолчанию будет первое указанное.
  // Во Flash 6 это должны быть имена экземпляров, а не переменные в стиле Flash 5.

  tabOrder = ["field1", "field3", "field2", "field4"];
}

onClipEvent (keyUp) {
  // Если нажата клавиша Tab...
  if (Key.getCode() == Key.TAB) {
    // ...Если ни у одного поля нет фокуса...
    if (Selection.getFocus() == null) {
      // ...то установить фокус на первое поле, указанное в массиве
      Selection.setFocus(path + "." + tabOrder[0]);
    } else {
      // Иначе искать текущее поле с фокусом в массиве полей
      i = 0;
```

```
        focused = Selection.getFocus();
        while (i < tabOrder.length) {
          // Извлечь имя переменной поля из полного пути,
          // возвращенного Selection.getFocus()
          fieldName = focused.substring(focused.lastIndexOf(".") + 1);
          // Проверить каждый элемент tabOrder на совпадение с полем в фокусе.
          if (tabOrder[i] == fieldName) {
            // Остановиться при обнаружении совпадения
            currentFocus = i;
            break;
          }
          i++;
        }
        // Теперь, когда положение поля в фокусе в массиве tabOrder
        // известно, переместить фокус в следующее или предыдущее поле
        // в зависимости от состояния клавиши Shift.
        if (Key.isDown(Key.SHIFT)) {
          // Клавиша Shift нажата, поэтому перейти к предыдущему полю;
          // если текущее поле первое, перейти в последнее поле.
          nextFocus = currentFocus-1 == -1 ? tabOrder.length-1 : currentFocus-1;
        } else {
          // Клавиша Shift не нажата, поэтому перейти к следующему полю;
          // если текущее поле последнее, перейти в первое поле.
          nextFocus = currentFocus+1 == tabOrder.length ? 0 : currentFocus+1;
        }
        // Наконец, установить фокус в новое положение
        Selection.setFocus(path + "." + tabOrder[nextFocus]);
      }
    }
  }

  // КОД НА КНОПКЕ НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
  on (keyPress "<Tab>") {
    // Этот код-заместитель просто перехватывает клавишу Tab в Internet Explorer
    var tabCatcher = 0;
  }
```

См. также

MovieClip.focusEnabled, ***Selection.getFocus()***, ***Selection.setSelection()***, *TextField*.selectable, *TextField*.tabEnabled

Selection.setSelection(), метод

Flash 5

выделяет символы в текстовом поле, имеющем фокус, или устанавливает точку ввода

Selection.setSelection(*beginIndex*, *endIndex*)

Аргументы

beginIndex

Неотрицательное целое число, задающее индекс первого символа, который должен быть включен в новое выделение.

endIndex

Неотрицательное целое число, задающее индекс символа *за* последним символом, который должен быть включен в новое выделение.

Описание

Метод *setSelection( )* выделяет (высвечивает) символы от `beginIndex` до `endIndex - 1` в текстовом поле, имеющем фокус. Если ни у одного поля нет фокуса, *setSelection( )* не оказывает никакого воздействия. Обычно метод применяется для высвечивания введенных пользователем некорректных данных. Для замены текста в текущем выделении предназначен метод *TextField. replaceSel( )*.

Применение

У объекта *Selection* нет метода «*setCaretIndex*» (установить индекс возврата каретки), но с помощью метода *setSelection( )* можно установить точку ввода в определенное место текстового поля. Для этого надо задать одинаковые значения `beginIndex` и `endIndex`, например:

```
// Установить точку ввода после третьего символа
Selection.setSelection(3, 3);
```

Пример

```
// Выделить вторую и третью буквы текстового поля, имеющего фокус в данный момент
Selection.setSelection(1, 3);
```

См. также

Selection.getBeginIndex( ), *Selection.getCaretIndex( )*, *Selection.getEndIndex( )*, *TextField. replaceSel( )*

setInterval(), глобальная функция

Flash 6

устанавливает интервал выполнения функции или метода равным n миллисекундам

```
setInterval(function, interval, arg1, arg2,...argn)
setInterval(object, method, interval, arg1, arg2,...argn)
```

Аргументы

function	Ссылка на функцию, которую следует выполнять.
interval	Временной промежуток между выполнениями `function` или `method` в миллисекундах.
arg1,...argn	Список необязательных аргументов, передаваемых в `function` или `method`.
object	Объект, метод которого `method` будет выполняться каждые `interval` миллисекунд.
method	Строка с именем выполняемого метода (должен быть определен в `object`).

Возвращаемое значение

Идентификатор интервала, с помощью которого можно прекратить выполнение `function` или `method` по таймеру (что называют «очисткой интервала»).

Описание

Функция *setInterval( )* открывает *интервал*, который выполняет заданные функцию `function` или метод `method` каждые `interval` миллисекунд. Откроем очень простой интервал, который будет выполнять функцию *helloWorld( )* раз в секунду:

```
function helloWorld () {
  trace("Hello world!");
}
setInterval(helloWorld, 1000);  // 1000 миллисекунд составляет 1 секунду
```

Обратите внимание, что ссылка на функцию `helloWorld` указана без оператора вызова функции (). Следующая попытка установить интервал будет неудачной из-за ошибочного включения оператора вызова:

```
setInterval(helloWorld(), 1000);  // Неправильно! Требуется helloWorld, а не helloWorld().
```

В *setInterval( )* можно передавать литерал функции, например:

```
setInterval(function () {trace("Hello world!");}, 1000);
```

Указанной функции всед за параметром *interval* можно передать аргументы:

```
function hello (person) {
  trace("Hello, " + person);
}
setInterval(hello, 1000, "Mr. Thilgen");
```

Однако переданные аргументы вычисляются только один раз, при инициализации интервала. Например, следующий код ошибочно пытается перемещать клип под курсор мыши путем передачи координат курсора в функцию интервала:

```
function followMouse (clip, mousex, mousey) {
  clip._x = mousex;
  clip._y = mousey;
}
// Не будет работать, поскольку _root._xmouse и _root._ymouse выполняются только один раз
setInterval(followMouse, 100, trailer_mc, _root._xmouse, _root._ymouse);
```

В правильном варианте данные, изменяющиеся между вызовами функции, извлекаются в самой функции:

```
function followMouse (clip) {
  // Получить _root._xmouse и _root._ymouse внутри функции
  clip._x = _root._xmouse;
  clip._y = _root._ymouse;
}
setInterval(followMouse, 100, trailer_mc);
```

Другой вариант синтаксиса *setInterval( )* предназначен для периодического вызова метода объекта. Следующий код выполняет метод *nextFrame( )* главной временной диаграммы (`_root`) каждую четверть секунды, вызывая эффект слайд-шоу:

```
setInterval(_root, "nextFrame", 250);
```

Помните о необходимости задавать имя метода в виде строки, а не как ссылку на функцию!

Если в *setInterval( )* передается ссылка на функцию, то значением ключевого слова `this` в этой функции оказывается `undefined`:

```
function showThis () {
  trace(this);
}
setInterval(showThis, 100);  // 'this' будет иметь значение undefined
```

Однако при передаче *setInterval( )* объектной ссылки и имени функции значением ключевого слова `this` будет объект, переданный в качестве первого аргумента:

```
setInterval(_root, "showThis", 100); // 'this' окажется ссылкой на _root
```

При создании интервала ему присваивается ID (идентификатор интервала), который возвращается функцией *setInterval( )*. Этот ID следует сохранить в переменной, чтобы позднее интервал можно было остановить с помощью *clearInterval( )*, например:

```
// Открыть интервал
var intervalID = setInterval(someFunction, 50);
// Впоследствии закрыть интервал
clearInterval(intervalID);
```

Часто интервал очищается в функции, указанной в первоначальном обращении к *setInterval( )*:

```
var counter = 1;
function countToTen () {
  trace(counter);
  // Если сделали 10 раз...
  if (counter == 10) {
    // ...остановит интервал
    clearInterval(counterInterval);
  }
  counter++;
}

// Открыть интервал
var counterInterval = setInterval(countToTen, 200);
```

Если не сохранить ID интервала, то его выполнение нельзя прекратить. Во Flash 6 идентификаторы интервалов являются целыми числами, но не следует останавливать интервалы, пытаясь угадать численное значение их ID. Архитектура ID закрыта и может в будущем измениться. Чтобы остановить группу интервалов *вместе*, запишите их ID в массив и применяйте *clearInterval( )* к каждому элементу.

Учтите, что присваивание нового значения переменной, содержащей ID интервала, не очищает интервал – его выполнение будет продолжаться. Очистить интервал можно только с помощью *clearInterval( )*. Например, в следующем коде переменной `intervalID` присваиваются два ID интервалов, но обе функции, *functionA( )* и *functionB( )*, продолжают вызываться каждые 100 мс, а `intervalID` хранит ID только второго интервала:

```
var intervalID = setInterval(functionA, 100);
var intervalID = setInterval(functionB, 100);
// Переменная intervalID теперь содержит ID второго интервала,
// но первый интервал продолжает выполняться (и остановить его нельзя!)
```

Функция *setInterval( )* предоставляет основанную на таймере альтернативу обработчику *onEnterFrame( ) MovieClip*, в которой код выполняется на каждый такт частоты смены кадров. Однако *setInterval( )* оказывается все же зависимой от частоты смены кадров фильма. Интервал может выполняться не более 10 раз в каждом кадре. Поэтому даже в фильме с частотой 1 кадр в секунду интервал может выполняться не чаще 10 раз в секунду (каждые 100 мс), даже если задано меньшее значение `interval`. Теоретически при частоте 10 кадров в секунду интервал может выполняться 100 раз в секунду (10 раз на кадр × 10 кадров в секунду), или каждые 10 мс. Для того чтобы обновлять экран из функции интервала, выполняемой чаще, чем происходит смена

кадров, вызовите *updateAfterEvent( )* из этой функции или того метода, который выполняется интервалом. Например:

```
function moveClip (clip) {
  // Переместить клип
  clip._x += 5;
  // Обновить экран
  updateAfterEvent();
}

setInterval(moveClip, 1, ball_mc);
```

Если интервал задан так, что промежуток между его выполнениями больше, чем промежуток между тактами частоты смены кадров, то он будет выполняться в том кадре, который появится по истечении времени интервала. Так, если задан интервал длительностью 400 мс и частота смены кадров составляет 10 $с^{-1}$ (1 кадр каждые 100 мс), то по табл. С.16 можно узнать, в каких кадрах будет выполняться интервал.

Таблица С.16. Выполнение интервала относительно показа кадров

Номер кадра	Суммарное время (мс)	Интервал выполнен?
0	0	нет
1	100	нет
2	200	нет
3	300	нет
4	400	да
5	500	нет
6	600	нет
7	700	нет
8	800	да
9	900	нет
10	1000	нет

Но как бы ни выполнялся интервал по сравнению со сменой кадров фильма – чаще или реже, временные промежутки интервала не могут быть гарантированы. При фактическом тестировании фильма интервалы в табл. С.16 могут оказаться выполненными в кадрах 5 и 9 или даже 4 и 9. Выполняемый код, вывод проигрывателем на экран и доступные системные ресурсы – все это может повлиять на фактическое время выполнения интервалов. Если учету времени отводится в приложении важная роль (например, в игре), то функции следует писать так, чтобы они учитывали возможные отклонения времени выполнения интервалов.

Применение

В ActionScript нет функции *setTimeout( )* в стиле JavaScript. Для моделирования *setTimeout( )* создайте функцию интервала, очищающую себя при первом выполнении, как в следующем примере.

Пример

Здесь показан простой таймер, который выполняется с задержкой 5 секунд и больше никогда не выполняется снова:

```
function timesUp () {
  trace("Time's up!");
  // Остановить таймер
  clearInterval(timerInterval);
}
timerInterval = setInterval(timesUp, 5000);
```

См. также *clearInterval( ), MovieClip.onEnterFrame( )*

setProperty(), глобальная функция

Flash 4

присваивает значение свойству клипа

`setProperty(movieClip, property, value)`

Аргументы

movieClip Выражение, возвращающее строку, которая содержит путь к клипу. Во Flash 5 и выше строка может быть и ссылкой на клип, поскольку в строковом контексте ссылки на клипы преобразуются в пути.

property Имя встроенного свойства из Flash 4, которому должно быть присвоено значение `value`. Должно быть идентификатором, а не строкой (то есть `_alpha`, а не `"_alpha"`).

value Новое значение данных, которое должно быть присвоено указанному свойству `property` клипа `movieClip`.

Описание

Функция *setProperty( )* присваивает `value` одному из встроенных свойств `movieClip`, относящихся к Flash 4. Не может применяться для задания значений свойств, определенных пользователем, и свойств, добавленных к `MovieClip` во Flash 5 и более поздних. Во Flash 4 *setProperty( )* была единственным способом присваивания значений свойствам клипа; начиная с Flash 5 предпочтительными способами установки как встроенных, так и пользовательских свойств являются операторы `.` и `[]`.

Пример

```
// Синтаксис Flash 4. Повернуть главный фильм на 45 градусов:
setProperty("_root", _rotation, 45);

// Синтаксис Flash 5. Тоже поворачивает главный фильм на 45 градусов:
_root._rotation = 45;
```

См. также *getProperty( )*; раздел «'Объектность' клипов» главы 13; приложение C

SharedObject, объект

Flash 6

локальное хранение данных и передача удаленных данных

Конструктор

Отсутствует; экземпляры создаются с помощью *getLocal( )* или *getRemote( )*.

Свойства

data Контейнер данных для *SharedObject*.

Методы

close( ) Закрывает соединение между удаленным объектом *SharedObject* и Comm Server. См. документацию Comm Server.

connect( ) Создает соединение с удаленным объектом *SharedObject* на Comm Server. См. документацию Comm Server.

flush( ) Принудительно записывает данные на диск.

getLocal( ) Возвращает ссылку на указанный локальный экземпляр *SharedObject*.

getRemote( ) Возвращает ссылку на удаленный экземпляр *SharedObject*, которым управляет Comm Server. См. документацию Comm Server.

getSize( ) Возвращает размер *SharedObject* в байтах.

setFPS( ) Задает частоту, с которой изменения удаленного *SharedObject* посылаются на Comm Server. См. документацию Comm Server.

Обработчики событий

onStatus( ) Вызывается, когда *SharedObject* генерирует сообщение о своем состоянии (например, об ошибке, информационное или предупреждение).

Описание

Объект *SharedObject* предоставляет средства для записи данных ActionScript на компьютер конечного пользователя и передачи данных между клиентами Flash через Comm Server. Если данные хранятся локально, то *SharedObject* называют локальным совместно используемым объектом (*local SharedObject* – LSO). Если имеет место удаленная передача данных, то *SharedObject* называют удаленным совместно используемым объектом (*remote SharedObject* – RSO). Здесь мы опишем только возможности *SharedObject* по локальному хранению данных. Сведения о возможностях удаленной передачи данных можно найти в документации по Comm Server:

http://www.macromedia.com/software/flashcom/

Подобно cookies JavaScript в веб-броузере, локальный *SharedObject* постоянно хранит информацию в локальной файловой системе. Когда фильм закрывается, информация записывается в файл. Когда фильм открывается повторно, информация вновь становится доступной. Благодаря этому из фильма можно, скажем, записывать лучшие результаты игрока и делать их доступными в следующих нескольких сеансах игры. Локальные *SharedObject* обладают большей гибкостью, чем cookies JavaScript. Они дают возможность сохранять и получать типизированные данные прямо в виде свойств объекта, а не в виде неудобной строки cookie с парами имя/значение. Конкретно, *SharedObject* поддерживает следующие типы данных: *number*, *string*, *boolean*, *undefined*, *null*, *array* и *object* (пользовательские классы плюс *XML* и *Date*). Остальные встроенные классы и объекты ActionScript (например, *MovieClip* и *Function*) хранить в *SharedObject* нельзя.

Экземпляры *SharedObject* не создают с помощью функции конструктора. Для создания нового локального *SharedObject* применяется метод *getLocal( )*, принимающий в качестве параметра имя создаваемого объекта *SharedObject*:

```
player1Info_so = SharedObject.getLocal("player1");
```

Если *SharedObject* с указанным именем (`"player1"`) уже существует для данного фильма, то этот код извлечет его из локальной файловой системы, а если нет, то проигры-

ватель создаст его. Обратите внимание на суффикс _so; это официальный суффикс Macromedia для *SharedObject*, который будет включать подсказчик кода в будущих версиях Flash.

Создав или загрузив объект совместного доступа, мы можем помещать в него информацию путем задания значений пользовательских свойств для встроенного свойства data, например:

```
player1Info_so.data.highScore = 1200;
```

Когда закрывается фильм, вызывается метод *flush( )* или удаляется и попадает в сборку мусора *SharedObject*, Flash делает попытку сохранить свойства data в двоичном файле с расширением *.sol*. Попытка записи будет неудачной, если размер сохраняемых данных превосходит заданные пользователем границы, о чем рассказывается в описании *SharedObject.getLocal( )*. Поэтому следует всегда специально записывать данные с помощью *flush( )*, а не полагаться на автоматическое сохранение, которое не дает возможности обрабатывать превышение размеров.

Сохраненный *SharedObject* может впоследствии быть извлечен с помощью метода *getLocal( )* точно в том виде, в каком был создан:

```
// Загрузить ранее созданный SharedObject
player1Info_so = SharedObject.getLocal("player1");
```

С любыми свойствами, определенными в объекте, можно работать сразу после вызова *getLocal( )* и загрузки LSO. Например, запишем в текстовое поле hiScore_txt значение player1Info_so.data.highScore, созданное нами ранее:

```
hiScore_txt.text = player1Info_so.data.highScore;
```

По умолчанию фильмы, принадлежащие одному домену, могут в сумме записать на компьютер пользователя до 100 Кбайт данных. О том, как запросить дополнительное пространство и обрабатывать запросы записи, превышающие установленные пользователем пределы, рассказано в описании *SharedObject.flush( )*.

Развернутый пример, показывающий, как с помощью *SharedObject* сохранить записи адресной книги, есть в следующей моей статье:

http://www.macromedia.com/desdev/mx/flash/articles/addressbook.html

Пример

Следующий код создает *SharedObject* с именем userProfile, регистрирующий суммарное время просмотра фильма пользователем:

```
var userProfile_so = SharedObject.getLocal("userProfile");
trace(userProfile_so.data.totalVisitTime);
var then = 0;

this.onEnterFrame = function () {
  var now = getTimer();
  var frameTime = now - then;
  userProfile_so.data.totalVisitTime += frameTime;
  this.then = now;
}
```

См. также «Свойства объектов», глава 12

SharedObject.data, свойство

Flash 6

контейнер для данных, хранящихся локально или отправляемых на Comm Server

только для чтения

theSharedObj.data

Описание

Свойство data содержит общий объект, свойства которого записываются на диск при выгрузке фильма из проигрывателя или вызове *SharedObject.flush( )*. Если *theSharedObj* представляет собою локальный *SharedObject*, загруженный с диска, то в свойствах data содержатся те значения, которые были сохранены в последний раз. Это могут быть данные следующих типов: *number*, *string*, *boolean*, *undefined*, *null*, *array* и *object* (пользовательские классы плюс *XML* и *Date*). Остальные встроенные классы и объекты ActionScript (например, *MovieClip* и *Function*) хранить в *SharedObject* нельзя.

Для того чтобы сохранить значение, его надо записать в свойство data. Свойства, прикрепляемые непосредственно к *theSharedObj*, на диск не записываются, а присваивания самому свойству data вообще игнорируются. Таким образом, прикреплять свойства следует к *theSharedObj*.data, а не к самому *theSharedObj*, и не пытаться задать новое значение свойства data. Например:

```
timeTracker_so = SharedObject.getLocal("punchClock");

timeTracker_so.data.time = new Date();  // ПРАВИЛЬНО! Будет сохранено.
timeTracker_so.time = new Date();       // НЕПРАВИЛЬНО! Не будет сохранено.
timeTracker_so.data = new Date();       // НЕПРАВИЛЬНО! Игнорируется.
```

Свойства data можно удалять с помощью оператора *delete*. Например:

```
delete theSharedObj_so.data.score;
```

Присваивание свойству значения null или undefined удаляет значение, но само свойство сохраняется, будет обнаруживаться в цикле *for-in* и сохраняться на диске, хотя и с пустым значением.

По умолчанию максимальный объем данных, который можно сохранить на диске, составляет 100 Кбайт для каждого домена. Подробнее см. *SharedObject.flush( )*. Узнать объем свойств, присвоенных data, можно с помощью *SharedObject.getSize( )*.

Если свойством data оказывается объект пользовательского класса, процедура сохранения/извлечения разрывает связь между этим объектом и его классом. После извлечения объекта методы и свойства, получаемые им по цепочке прототипов, оказываются недоступными. Однако можно обратиться к методу *Object.registerClass( )* и связать объект с идентификатором, который будет использован для восстановления его класса при извлечении с диска. Подробнее см. в описании *Object.registerClass( )*.

Пример

Следующий код извлекает локальный *SharedObject* с именем «gameData». Затем присваивается значение свойству player1score, и объект сохраняется в результате вызова *flush( )*:

```
// Создать/извлечь SharedObject.
gameData_so = SharedObject.getLocal("gameData");
// Создать сохраняемое свойство.
gameData_so.data.player1score = 2600;
// Попытка сохранения SharedObject.
gameData_so.flush();
```

См. также

Object.registerClass( ), *SharedObject.flush( )*, *SharedObject.getLocal( )*, *SharedObject.getSize( )*

SharedObject.flush(), метод

Flash 6

принудительная запись данных на диск

```
theSharedObj.flush()
theSharedObj.flush(minimumDiskSpace);
```

Аргументы

minimumDiskSpace

Необязательное целое, указывающее (в байтах) объем локального дискового пространства, запрошенный для текущего домена. Позволяет запросить больше дискового пространства, чем требует `theSharedObj`. Если `minimumDiskSpace` опущен, *flush( )* запрашивает столько памяти, сколько возвращает `theSharedObj`.*getSize( )*.

Возвращаемое значение

Строка со значением `"pending"`, если запрошено дополнительное пространство для записи объекта, или булево значение – `true`, если запись успешна, и `false`, если неудачна.

Описание

Метод *flush( )* делает попытку сохранить свойства `theSharedObj.data` в файле *.sol*, о чем рассказывается в описании `SharedObject`.*getLocal( )*. Функционально это аналогично автоматическому сохранению, происходящему при выгрузке фильма из проигрывателя, но дает возможность запросить у пользователя дополнительное дисковое пространство, если это необходимо.

При выполнении *flush( )* Flash проверяет настройки локального хранения для домена, из которого загружен фильм. Если для данного домена в этих настройках указано «Never», то *flush( )* возвращает `false`, а *SharedObject* не сохраняется. Если разрешенный объем памяти для локального хранения превышает размер *SharedObject* и значение `minimumDiskSpace`, то *flush( )* возвращает `true` и *SharedObject* сохраняется. Если разрешенный объем памяти для локального хранения меньше, чем размер *SharedObject* или значение `minimumDiskSpace`, то *flush( )* возвращает строку `"pending"`, а Flash предлагает пользователю увеличить границу объема памяти, выделенной для локального хранения. После того как пользователь одобрит или отклонит запрос на увеличение объема памяти, будет вызван обработчик `theSharedObj.onStatus()` с параметром `infoObject`, содержащим информацию об успешности попытки сохранения.

Аргумент `minimumDiskSpace` задается для того, чтобы Flash запросил у пользователя больше памяти, чем фактически требует *SharedObject*. Это делается, чтобы избежать многократных запросов памяти при очередных увеличениях размера *SharedObject*.

Пример

В следующем коде показана типичная кнопка Save для игры, записывающей большой объем данных в *SharedObject*. При нажатии кнопки Save фильм сообщает о ходе сохранения и при необходимости просит пользователя подтвердить выделение памяти, необходимой для сохранения игры. Чтобы показать пользователю, какой объем памяти требуется, вызывается метод *getSize( )*. Обратите внимание, что в полном интерфейсе сохранения требуется реализовать *onStatus( )*:

```
// Создать SharedObject.
var gameData_so = SharedObject.getLocal("gameData");
// Присвоить большой объект карты как свойство (предполагается, что mapObj содержит данные игры).
gameData_so.data.customUserMap = mapObj;
// Вызвать saveGame() при нажатии save_btn.
save_btn.onRelease = saveGame;

// Функция saveGame() пытается сохранить объект совместного использования
function saveGame () {
  // Создать текстовое поле для сообщений пользователю о ходе сохранения.
  this._parent.createTextField("output_txt", 1, 0, 0, 300, 100);
  output_txt.border = true;
  output_txt.wordWrap = true;

  // Сообщить пользователю, что процесс сохранения начат.
  output_txt.text =  "Saving game...please wait...\n";
  // Запросить дисковую память, исходя из getSize(). Разделить на 1024, чтобы преобразовать
  // в килобайты, и запросить дополнительные 10 Кбайт, чтобы реже делать новые запросы.
  output_txt.text += "Space required: "
                  + (Math.floor(gameData_so.getSize() / 1024) + 10) + "KB.\n";

  // Сохранить SharedObject и записать в локальную переменную возвращенное значение.
  var saveAttempt = gameData_so.flush(gameData_so.getSize() + 10240);
  // Если flush() возвратил...
  if (saveAttempt == true) {
    // ...true, сохранение успешно, и работа завершена.
    output_txt.text += "Game saved.\n";
  } else if (saveAttempt == "pending") {
    // ..."pending", Flash запрашивает у пользователя дополнительное дисковое пространство.
    // Решение пользователя отражает infoObject, который будет передан onStatus(),
    // когда пользователь закроет диалоговое окно настройки.
    output_txt.text += "Please increase disk space to continue.\n";
  } else if (saveAttempt == false) {
    // ...false, пользователем запрещено локальное хранение для этого домена.
    // Попросить пользователя разрешить локальное хранение.
    output_txt.text += "Save failed. To enable saving, "
                    +  "uncheck 'Never' under Local Settings.\n";
    // Следуя этому, откроем диалоговое окно настроек локального хранения,
    // где пользователь сможет разрешить хранение, сбросив флаг Never.
    System.showSettings(1);
  }
}

// Функция, вызываемая, если flush() не хватило памяти и возникла ситуация "pending".
// Пользователю было предложено увеличить память для домена, и он согласился или отклонил запрос.
gameData_so.onStatus = function (infoObject) {
  // Свойство infoObject.code сообщит, что произошло.
  if (infoObject.code == "SharedObject.Flush.Success") {
    // Ура! Нам дали память!
    output_txt.text += "Required disk space added. Game saved.";
  } else if (infoObject.code == "SharedObject.Flush.Failed") {
    // Досадно. Памяти не дали. Игру сохранить нельзя.
    output_txt.text += "Insufficient disk space. Save failed.";
  }
}
```

См. также

SharedObject.getSize( ), SharedObject.onStatus( ), System.showSettings( )

SharedObject.getLocal(), метод

Flash 6

возвращает ссылку на заданный экземпляр SharedObject

```
theSharedObj.getLocal(objectName)
theSharedObj.getLocal(objectName, localPath);
```

Аргументы

objectName

Строка с именем локального объекта *SharedObject*. Может содержать прямой слэш (/). Не может содержать символы: ~ % & \ ; : " ' , < > ? #.

localPath

Необязательная строка, задающая часть локального пути к файлу *.sol*, сохраненному на компьютере клиента. Должна полностью или частично совпадать с фактическим путем к фильму, создающему `theSharedObj`.

Возвращаемое значение

Локальный *SharedObject* с именем `objectName` или `null`, если задан недопустимый `localPath`.

Описание

Посредством метода *getLocal( )* создается и извлекается локальный *SharedObject*. Этот метод ищет файл *.sol* с именем `objectName` по адресу, частично указанному в `localPath`. Файл *.sol* содержит данные, необходимые для воссоздания объекта ActionScript *SharedObject* в том виде, в каком он был последний раз сохранен. Если найден такой файл *.sol*, то *getLocal( )* возвращает локальный *SharedObject*, свойство `data` которого содержит значения, сохраненные в файле *.sol* в прошлый раз. Если соответствующий файл *.sol* не найден, *getLocal( )* создает и возвращает новый *SharedObject* с именем `objectName`. Если указанный путь `localPath` недопустим, *getLocal( )* возвращает `null`, что указывает на ошибку при выполнении операции.

Если `localPath` не задан, файл *.sol* извлекается по адресу, зависящему от операционной системы и отражающему путь к фильму, из котороого был вызван *getLocal( )*. Обычно этот путь включает имя самого фильма. Например, на моей машине Windows XP *SharedObject* с именем «player1», созданный фильмом *so.swf* на *http://www.moock.org/asdg/so.swf*, извлекается по следующему адресу:

```
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\asdg\so.swf\player1.sol
```

Обратите внимание, что *so.swf* фактически выступает в качестве имени каталога в пути к файлу *.sol*. Это предотвращает конфликты имен *.sol* среди *SharedObject*, создаваемых несколькими фильмами в одном и том же каталоге. Например, если в фильмах *asteroids.swf* и *pong.swf* (размещенных в *moock.org/games/)* создан *SharedObject* с одинаковым именем «player1», то их адресами на моем компьютере будут следующие:

```
// Asteroids
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\games\asteroids.swf\player1.sol

// Pong
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\games\pong.swf\player1.sol
```

Если *asteroids.swf* и *pong.swf* должны использовать один и тот же *SharedObject*, им надо указать общий `localPath` при вызове *getLocal( )*. Параметр `localPath` должен

содержать полный фактический путь (или его часть) к файлу *.swf* после имени домена. Например, в *pong.swf* аргумент *localPath* может иметь значение "/", "/games" или "/games/pong.swf", что дает следующие адреса (опять-таки, на моем компьютере):

```
// Для localPath "/" адресом будет:
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\player1.sol

// Для localPath "/games" адресом будет:
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\games\player1.sol

// Для localPath "/games/pong.swf" адресом будет:
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\moock.org\games\pong.swf\player1.sol
```

В результате любой фильм в каталоге */games* на *moock.org* может получить доступ к *SharedObject* с именем "player1" посредством такого обращения к *getLocal( )*:

```
getLocal("player1", "/games");
```

Любой фильм на всем сайте *moock.org* может получить доступ к *SharedObject* с именем "userInfo" посредством такого обращения к *getLocal( )*:

```
// Такое обозначение применяется для хранения глобально доступных SharedObjects
getLocal("userInfo", "/");
```

Однако фильм в каталоге */products* не может обратиться к *SharedObject*, созданному с *localPath*, равным "/games".

Локальный путь к файлу *.sol* не всегда имеет вид, пригодный для чтения человеком (как это было в наших примерах). Если путь очень длинный, файл *.sol* может получить имя типа *.../domain/343345345345345.sol*. По этой причине не следует полагаться на адрес или имя локального файла *.sol* и возлагать извлечение данных исключительно на API для *SharedObject*. Параметр *localPath* метода *getLocal( )* не зависит от внутреннего имени, используемого Flash для хранения фактического файла *.sol*.

Если фильм создает *SharedObject*, а затем перемещается в другой каталог в том же домене, он станет искать *SharedObject* по своему новому адресу и не найдет тех *SharedObject*, которые были созданы по его старому местонахождению, если не задать *localPath* правильным образом.

Фильмам, выполняющимся в локальной файловой системе (например, в автономном проигрывателе), назначается домен «localhost». Например, вот местонахождение *phoneNumbers.sol*, созданного фильмом, выполняющимся в корне любого жесткого диска на моем компьютере:

```
C:\Documents and Settings\fritz\Application Data\Macromedia\Flash
Player\localhost\phoneNumbers.sol
```

Обратите внимание, что два локальных фильма, выполняющихся из одинаковых путей на двух разных локальных дисках, вступят в конфликт между собой, поскольку файлы *.sol* по умолчанию всегда сохраняются на системном диске. Так, следующие два фильма могут читать и записывать локальные *SharedObject*, принадлежащие друг другу:

```
C:\circle\demo.swf
D:\circle\demo.swf
```

Чтобы удалить с диска файл *.sol*, удалите все, что содержится в объекте `data` соответствующего *SharedObject*. Например:

```
// Извлечь SharedObject player1
player1Info_so = SharedObject.getLocal("player1");

// Удалить все свойства в player1Info_so.data
for (var p in player1Info_so.data) {
  delete playerInfo_so.data[p];
}
```

Установка в проигрывателе низкой границы объема памяти для локального хранения не мешает созданию экземпляров *SharedObject* в ActionScript. Однако при закрытии фильма или вызове *flush( )* соответствующий файл *.sol* не будет создан, если имеющегося дискового пространства для этого недостаточно. См. *SharedObject.flush( )* и *System.showSettings( )*.

Ошибки реализации

Во Flash 6 до выпуска 29 (версия 6.0.29.0) при извлечении фильма с URL-строкой запроса последняя включается в имя каталога, содержащего локальный файл *.sol*. Например, файл *.sol*, который сохранен из фильма, загруженного с

http://www.moock.org/intro.swf?type=long

ошибочно записывается в локальный каталог

```
www.moock.org/intro.swf?type=long/
```

В Windows не допускаются вопросительные знаки в именах файлов, поэтому попытка записать локальный файл *.sol* в такой ситуации окажется безуспешной. В более поздних версиях Flash Player (6.0.30.0 или более поздних) файл *.sol* хранится в правильном месте:

```
www.moock.org/intro.swf/
```

При задании локального *SharedObject* в фильме, загруженном по строке запроса, следует задать параметр *localPath* метода *getLocal( )*, чтобы не допустить включение строки запроса в имя каталога для локального файла *.sol*.

В документации Flash Comm Server методы *getRemote( )*, *connect( )* и *close( )* объекта *SharedObject*, применяемые для управления удаленными объектами совместного использования, рассмотрены подробно.

См. также *SharedObject.getSize( )*, *System.showSettings( )*

SharedObject.getSize(), метод

Flash 6

возвращает размер SharedObject в байтах

`theSharedObj.getSize()`

Возвращаемое значение

Общий размер `theSharedObj` в байтах.

Описание

Метод *getSize( )* возвращает суммарный размер в байтах всех свойств `theSharedObj.data`. При запросе на увеличение объема локальной памяти для домена можно посредством метода *getSize( )* сообщить пользователю, сколько именно памяти требует

фильм (см. пример для *SharedObject.flush( )*). Чтобы преобразовать значение в килобайты, надо разделить его на 1024:

```
soKB = Math.floor(theSharedObj.getSize() / 1024);
```

Чем больше свойств в *theSharedObj*.data, тем дольше выполняется *getSize( )*.

См. также

SharedObject.flush( )

SharedObject.onStatus(), обработчик события

Flash 6

вызывается, когда SharedObject генерирует сообщение статуса (ошибка, информация, предупреждение)

theSharedObj.onStatus(*infoObject*)

Аргументы

infoObject

Объект, в свойствах code и level которого содержится сообщение о состоянии.

Описание

Для локальных *SharedObject* событие *onStatus( )* происходит, только если при вызове *flush( )* предельного объема локальной памяти недостаточно для сохранения *SharedObject* на диске. В этом случае Flash показывает окно диалога Settings и просит пользователя увеличить дисковое пространство для данного домена. После того как пользователь примет решение, вызывается *onStatus( )*, в параметре *infoObject* которого указывается, было ли добавлено запрашиваемое дисковое пространство. Объект *infoObject* определяет два свойства – code и level, возможные значения которых, относящиеся к локальным объектам *SharedObject*, перечислены в табл. C.17. Обратите внимание, что с *onStatus( )* не связывается статус "pending", однако слово «pending» может возвращаться методом *flush( )*.

Таблица C.17. Возможные значения свойств infoObject в SharedObject.onStatus( ) для локальных SharedObject

Значения свойства code	Значения свойства level	Смысл
«SharedObject.Flush.Failed»	«error»	Пользователь не выделил памяти для хранения *SharedObject*
«SharedObject.Flush.Success»	«status»	Пользователь выделил память для хранения *SharedObject*

Для локальных *SharedObject* достаточно проверить значение свойства code, как показано в следующем примере.

Пример

Следующий обработчик *onStatus( )* выполняется после того, как пользователь решил, выделить ли дисковое пространство, достаточное для хранения theSharedObj_so. Если не выделено достаточно памяти, этот код выключает приложение:

```
theSharedObj_so.onStatus = function (infoObject) {
  // Свойство code объекта infoObject сообщает о том, выделил ли пользователь
  // достаточный объем памяти для сохранения SharedObject.
  if (infoObject.code == "SharedObject.Flush.Success") {
```

```
        // Память выделена.
        output_txt.text = "Required disk space added. Application saved.";
      } else if (infoObject.code == "SharedObject.Flush.Failed") {
        // Память не выделена.
        output_txt.text += "You must increase disk space to use this application.";
        // Вывести кадр "application disabled".
        gotoAndStop("disabled");
      }
    }
```

См. также *SharedObject.flush( )*, *SharedObject.getSize( )*

Sound, класс

Flash 5; усилен во Flash 6

управление звуком в фильме и средства загрузки внешних звуков

Конструктор

new Sound( )
new Sound(*target*)

Аргументы

target Строка, задающая путь к клипу или уровню документа, звуком которого требуется управлять. Может также являться ссылкой на клип или уровень документа (в строковом контексте ссылки преобразуются в пути).

Свойства

duration Суммарная длительность звука в миллисекундах.

id3 Доступ к тегам ID3 в файле MP3.

position Длительность воспроизведенного звука в миллисекундах.

Методы

attachSound( )

Связать звук из библиотеки с экземпляром *Sound*.

getBytesLoaded( )

Количество загруженных байтов внешнего звукового файла.

getBytesTotal( )

Физический размер на диске внешнего звукового файла в байтах.

getPan( ) Извлечь текущую установку панорамирования.

getTransform( )

Определить текущее распределение каналов по левой и правой колонкам (то есть баланс).

getVolume( ) Извлечь текущую установку громкости.

loadSound( ) Загрузить в проигрыватель внешний звуковой файл.

setPan( ) Задать панорамирование звука по левому и правому каналам.

setTransform( )

Задать распределение каналов по левой и правой колонкам (то есть баланс).

setVolume( ) Задать громкость.

start( ) Начать воспроизведение прикрепленного или загруженного звука.

stop( ) Заглушить все звуки либо указанный прикрепленный или загруженный звук.

Обработчики событий

onLoad( ) Происходит, когда заканчивается загрузка внешнего звука.

onSoundComplete( )

Происходит, когда заканчивается воспроизведение звука.

Описание

Объекты класса *Sound* применяются для загрузки внешних звуковых файлов и управления имеющимися или загруженными звуками фильма. Объекты *Sound* могут управлять:

- Всеми звуками Flash Player.
- Всеми звуками конкретного экземпляра клипа или главного фильма (в том числе всеми звуками во вложенных клипах).
- Отдельным звуком, прикрепленным программно или загруженным.

Для создания объекта *Sound*, управляющего всеми звуками в Player (в том числе звуками в файлах *.swf* на уровнях документов), следует вызвать конструктор *Sound* без параметров:

```
myGlobalSound = new Sound();
```

Для того чтобы создать объект *Sound*, управляющий звуками конкретного клипа или главного фильма, задайте параметр `target`, указывающий клип или фильм, которыми требуется управлять. Обратите внимание, что при этом также контролируются звуки в клипах, находящихся внутри `target`. Например:

```
spaceshipSound = new Sound("spaceship");  // Управление звуками в клипе spaceship
mainSound      = new Sound("_root");      // Управление звуками
                                          // на главной временной диаграмме
```

Если требуется создать отдельный звук, который можно запускать, останавливать и зацикливать независимым образом, создайте какой-нибудь объект *Sound* и прикрепите к нему звук с помощью метода *attachSound( )* или *loadSound( )*.

А загрузить в объект *Sound* внешний звуковой файл можно посредством метода *loadSound( )*. Процесс загрузки звукового файла позволяют контролировать методы *getBytesLoaded( )*, *getBytesTotal( )* и *onLoad( )*.

Flash не предоставляет доступа или контроля частоты или высоты звука, но позволяет узнать общую длительность звучания (`duration`) и истекшее время (`position`).

Flash позволяет воспроизводить одновременно до восьми отдельных звуков. При загрузке с помощью *attachSound( )* для каждого одновременно воспроизводимого звука необходим свой экземпляр *Sound*.

Воспроизведение звука не прекращается при удалении объекта, который им управляет. Прекратить воспроизведение загруженного или прикрепленного звука позволяет метод *Sound.stop( )*.

См. также `capabilities.hasAudio`, *stopAllSounds( )*; `_soundbuftime`

Sound.attachSound(), метод

Flash 5

связывает звук из библиотеки с объектом Sound

`soundObject``.attachSound(`*`linkageIdentifier`*`)`

Аргументы

linkageIdentifier

Имя прикрепляемого звука, заданноне в библиотеке в пункте Options → Linkage.

Описание

Метод *attachSound( )* помещает экспортированный из библиотеки звук под управление `soundObject`. Прикрепленный звук можно отдельно запускать и останавливать, вызывая методы *start( )* и *stop( )* над `soundObject`. О том, как приостановить воспроизведение звука, рассказано в заметке

http://www.moock.org/asdg/technotes/pauseSound/

Для того чтобы прикрепить звук к `soundObject`, его надо экспортировать из библиотеки фильма, выполнив следующие шаги:

1. Выбрать в библиотеке экспортируемый звук.
2. Выбрать Options → Linkage. Появляется диалоговое окно Linkage Properties.
3. В Linkage выбрать Export For ActionScript.
4. Во Flash 6 можно сбросить флажок «Export in first frame».
5. В окне Identifier ввести уникальное имя, идентифицирующее звук.

Во Flash 5 *все* экспортированные звуки загружаются в первый кадр содержащего их фильма (не тогда, когда они действительно прикрепляются или воспроизводятся с помощью ActionScript), что может привести к задержке загрузки, если звуки велики по размеру.

Большей степени контроля над загрузкой звуков во Flash 5 можно достичь, если поместить их во внешние файлы *.swf* и импортировать с помощью *loadMovie( )* или *MovieClip.loadMovie( )* по мере необходимости.

Во Flash 6 прикрепленные звуки можно экспортировать в любой кадр. В окне диалога Linkage Properties сбросьте флажок «Export in first frame», после чего поместите звук в нужный кадр на главной временной диаграмме. Метод *attachSound( )* будет работать только после этого кадра. Другую возможность предоставляет метод *loadSound( )*, с помощью которого можно загрузить внешний звуковой файл, когда это потребуется.

Одновременно к объекту *Sound* может быть прикреплен только один звук, но можно создать до восьми объектов *Sound*, действующих одновременно. Прикрепление нового звука к объекту *Sound* заменяет звук, прикрепленный к этому объекту ранее. Чтобы синхронизировать несколько прикрепленных звуков, прикрепите их к отдель-

ным объектам *Sound* и запустите все в одном кадре, затем с помощью *Sound.setVolume( )* отрегулируйте громкость каждого звука.

Когда фильм с экспортированным звуком загружен на целевой уровень или в клип, экспортированный звук можно прикрепить только в том случае, когда заданное для объекта *Sound* свойство *target* совпадает с этим уровнем или клипом. Допустим, например, что мы сохранили экспортированные звуки в *linkedSounds.swf*, а затем загружаем *linkedSounds.swf* в клип с именем host. После завершения загрузки файла *.swf* можно успешно прикреплять и воспроизводить его экспортированные звуки, например:

```
// Создать новый sound с target равным host
host.bang = new Sound(host);
host.bang.attachSound("loudBang");
host.bang.start();
```

Распространенной ошибкой является пропуск аргумента *target* при создании объектов *Sound* в загружаемых фильмах. Если *target* не указан, то по умолчанию в качестве него принимается _level0, где нельзя найти экспортированный звук. Например, следующая попытка воспроизвести звук в host будет безуспешной, потому что звук loudBang недоступен в библиотеке для _level0:

```
// Создать новый sound, который будет управлять звуком _level0.
host.bang = new Sound();

// Операция будет неудачной..."loudBang" нет в библиотеке _level0.
host.bang.attachSound("loudBang");

// Воспроизведение не начнется, поскольку звук не был прикреплен.
// Мы забыли задать target при создании host.bang.
host.bang.start();
```

Внутри фильма, который будет загружаться в другие фильмы, можно обеспечить возможность прикрепления экспортированных звуков, используя ключевое слово this в качестве параметра *target* для всех объектов *Sound*. Например:

```
// Код внутри sounds.swf, который будет загружаться в другие фильмы
music = new Sound(this);      // Задать в качестве target текущий клип
music.attachSound("song");
music.start();
```

Пример

В следующем примере звук с идентификатором phaser добавляется к объекту *Sound* с именем *phaserSound*. Затем запускаем и останавливаем воспроизведение звука phaser. В конструкторе не требуется задавать *target*, поскольку это главный фильм, а не фильм, загружаемый в клип:

```
phaserSound = new Sound();
phaserSound.attachSound("phaser");

// Запустить звук phaser
phaserSound.start();

// Остановить только звук phaser
phaserSound.stop("phaser");
```

См. также *Sound.loadSound( )*, *Sound.start( )*, *Sound.stop( )*

Sound.duration, свойство

Flash 6

общая длительность звука в миллисекундах **только для чтения**

soundObject.duration

Описание

Свойство duration возвращает длительность в миллисекундах прикрепленного или загруженного звука, связанного с *soundObject*. Оно не применимо к звукам временной диаграммы, размещенным на этапе разработки. Если в один *soundObject* загружено несколько звуков, то свойство duration относится к звуку, загруженному последним. Если звук зациклен, то duration возвращает длину одного цикла. Если прикрепленных звуков нет, duration возвращает undefined. Во время загрузки внешнего потокового звука duration отражает длительность звука, загруженного к настоящему времени. Во время загрузки внешнего звука события duration содержит 0, пока звук не будет загружен полностью, и тогда в нем будет находиться фактическая длительность звука. Теоретически свойство duration должно быть доступно в *Sound.onLoad( )*, но это не так из-за ошибки во Flash 6 (см. ниже ошибки реализации).

Свойство duration совместно с position можно применять для определения процентного отношения уже воспроизведенного звука. См. в Sound.position пример кода, который увеличивает громкость пропорционально воспроизведенной части звука.

Ошибки реализации

Во Flash 6, когда звук экспортирован из библиотеки и вызван метод *loadSound( )* объекта *soundObject*, свойство duration совпадает с длиной библиотечного звука, пока часть внешнего звука не будет загружена в проигрыватель. Чтобы значение duration было точным, проверяйте его только после того, как *getBytesLoaded( )* возвратит положительное число.

Во Flash 6 свойство duration вначале недоступно в *onLoad( )*. Верное значение duration можно получить посредством опроса при помощи функции *setInterval( )*, например так:

```
song = new Sound();
song.onLoad = function (success) {
  // Из-за ошибки реализации duration не отображается
  trace("Initial Duration: " + this.duration);

  // Сохранить ссылку на текущий звук
  var theSound = this;

  // Функция, которая проверяет существование duration
  function pollForDuration () {
    if (theSound.duration >= 0) {
      // Показать duration, ставшее доступным
      trace("Actual duration: " + theSound.duration);
      // Воспроизвести звук
      theSound.start();
      // Прекратить опрос правильного duration
      clearInterval(intID);
    }
  }

  // Если звук загрузился нормально...
  if (success == true) {
```

```
      // Периодически проверять duration
      var intID = setInterval(pollForDuration, 100);
    }
  }

  // Загрузить файл MP3
  song.loadSound("track1.mp3", false);
```

Пример

Следующий код прикрепляет экспортированный звук «JingleBells» к объекту *Sound*, а затем получает длительность звука:

```
music = new Sound();
music.attachSound("JingleBells");
trace(music.duration);  // Выводит длительность JingleBells в миллисекундах
```

См. также *Sound*.position

Sound.getBytesLoaded(), метод

Flash 6

возвращает количество загруженных байт внешнего звукового файла

soundObject.getBytesLoaded()

Возвращаемое значение

Целое число, представляющее количество байт, загрузка которых в проигрыватель завершена. Разделить на 1024 для преобразования в килобайты.

Описание

Метод *getBytesLoaded( )* сообщает о количестве байт звукового файла, загруженных во Flash Player в результате вызова *loadSound( )*. Для нового объекта *Sound* метод *getBytesLoaded( )* возвращает undefined. Сразу после каждого вызова *loadSound( )* *getBytesLoaded( )* также возвращает undefined. По мере поступления данных *getBytesLoaded( )* возвращает количество байт, загруженных к данному моменту (от 0 до *getBytesTotal( )*), пока не будет загружен весь звук. С этого момента он начинает возвращать суммарное количество байт, полученных при последней передаче, пока не поступит новый вызов *loadSound( )*.

Для объектов *Sound* метод *getBytesLoaded( )* применяется совместно с *getBytesTotal( )*, чтобы определить (и, как правило, показать), как идет загрузка звукового файла. Однако для определения того, завершилась ли операция загрузки, метод *getBytesLoaded( )* не нужен: по окончании загрузки звука автоматически выполняется обработчик события *onLoad( )*.

Пример

Следующий код добавляет новый метод *preloadSound( )* во все объекты *Sound*. Он действует так же, как *loadSound( )*, за исключением того, что с помощью *getBytesLoaded( )* сообщает о ходе загрузки звука. Обработчик *onLoad( )* выполняется по завершении операции загрузки, как обычно. Учтите, что в данном виде этот код не может одновременно загружать несколько звуков, поскольку не создает очереди запросов загрузки. Загрузчик с очередью читатель может создать в качестве упражнения.

```
/*
 *    Метод: Sound.checkLoadProgress()
 * Описание: Проверяет ход выполнения load(). Вызывается только preload().  Сообщает о ходе
```

```
 *           загрузки, вызывая пользовательский обработчик Sound.onBytesLoaded().
 */
Sound.prototype.checkLoadProgress = function () {
  // Узнать, сколько килобайт загружено.
  var kbLoaded = Math.floor(this.getBytesLoaded()/1024);
  // Узнать размер загружающегося файла в килобайтах.
  var kbTotal = Math.floor(this.getBytesTotal()/1024);
  // Вычислить процент выполнения.
  var percentDone = isNaN(Math.floor(kbLoaded/kbTotal * 100)) ?
                    0 : Math.floor(kbLoaded/kbTotal * 100);
  // Выполнить onBytesLoaded(), который обычно показывает ход загрузки.
  this.onBytesLoaded(this.getBytesLoaded(), this.getBytesTotal(),
                     kbLoaded, kbTotal, percentDone);
  // Если объем загруженных данных равен размеру файла, загрузка файла завершена.
  // Однако следует убедиться, что getBytesLoaded() возвращает число больше 0, потому что
  // getBytesLoaded() и getBytesTotal() возвращают 0, пока не поступит первый байт данных.
  if (this.getBytesLoaded() > 5
      && this.getBytesLoaded() == this.getBytesTotal()) {
    // ...прекратить вызов checkLoadProgress().
    this.clearLoadCheck();
  }
}

/*
 *    Метод: Sound.clearLoadCheck()
 * Описание: Удаляет интервал, который вызывает checkLoadProgress().
 */
Sound.prototype.clearLoadCheck = function () {
  // Если существует интервал загрузки для этого объекта Sound ...
  if (this.loaderID) {
    // ...прекратить его вызов.
    clearInterval(this.loaderID);
  }
}

/*
 *    Метод: Sound.preloadSound()
 * Описание: Начинает загрузку звукового файла и выполняет
 *           checkLoadProgress() каждые 200 мс, пока файл не будет загружен.
 */
Sound.prototype.preloadSound = function (url, isStreaming) {
  // Прекратить выполнение другого предзагрузчика, если оно еще продолжается.
  this.clearLoadCheck();

  // Загрузить звуковой файл.
  this.loadSound(url, isStreaming);
  // Вызывать checkLoadProgress() каждые 200 мс.
  this.loaderID = setInterval(this, "checkLoadProgress", 200);
}

// Применение:
// Создать объект Sound.
music = new Sound();
// Присвоить функцию обратного вызова onLoad().
music.onLoad = function (success) {
  // Если загрузка успешна...
  if (success) {
    // ...использовать новый звук.
```

```
      trace("done loading!");
      this.start();
    } else {
      // ...иначе обработка ошибок и удаление интервала предзагрузчика (ОБЯЗАТЕЛЬНО!).
      trace("load failed");
      this.clearLoadCheck();
    }
  }
  // Присвоить функцию обратного вызова onBytesLoaded(). В ней каждый отдельный
  // объект Sound может решить, что делать с информацией предзагрузки.
  music.onBytesLoaded = function (bytesLoaded, bytesTotal, kbLoaded, kbTotal, percentLoaded) {
    trace("Loading: " + kbLoaded +  " of " + kbTotal);
    trace(percentLoaded + " percent complete.");
  }

  // Загрузить потоковый звук.
  music.preloadSound("http://www.yoursite.com/sounds/song.mp3", true);
```

См. также *setInterval( )*, *Sound.getBytesTotal( )*, *Sound.onLoad( )*

Sound.getBytesTotal(), метод

Flash 6

возвращает физический размер внешнего звукового файла в байтах

soundObject.getBytesTotal()

Возвращаемое значение

Целое число, представляющее размер звукового файла. Для преобразования в килобайты его следует разделить на 1024.

Описание

Метод *getBytesTotal( )* сообщает размер в байтах внешнего файла, загружаемого в *soundObject* посредством *loadSound( )*. Для нового объекта *Sound* метод *getBytesTotal( )* возвращает undefined. Сразу после каждого вызова *loadSound( ) getBytesTotal( )* также возвращает undefined, пока не будет установлен размер загружаемого файла. С этого момента он начинает возвращать общий размер загружаемого файла, пока не поступит новый вызов *loadSound( )*.

Для объектов *Sound* метод *getBytesTotal( )* используется вместе с *getBytesLoaded( )*, чтобы определить (и, как правило, показать), как идет загрузка звукового файла. См. подробности и пример кода в описании *getBytesLoaded( )*.

См. также *Sound.getBytesLoaded( )*, *Sound.onLoad( )*

Sound.getPan(), метод

Flash 5

возвращает последнее заданное значение панорамы

soundObject.getPan()

Возвращаемое значение

Число, указывающее последнее значение, заданное методом *setPan( )*. Обычно находится в диапазоне от –100 (левый канал включен, правый выключен) до 100 (правый канал включен, левый выключен). По умолчанию устанавливается 0 (левый и правый каналы в равной пропорции).

Описание

Изменяя панораму звука, можно создавать иллюзию движения его источника. Метод *getPan( )* применяется для того, чтобы выяснить текущее распределение звука между левым и правым каналами, которым управляет `soundObject`. Обычно *getPan( )* используется вместе с *setPan( )* для настройки текущей панорамы звука.

Пример

Следующий код изменяет панораму звука на 20 в сторону левого канала:

```
mySound = new Sound();
mySound.setPan(mySound.getPan() - 20);
```

См. также *Sound.getTransform( )*, *Sound.setPan( )*

Sound.getTransform(), метод[1] Flash 5

возвращает текущее распределение каналов звука между левой и правой колонками

`soundObject.getTransform()`

Возвращаемое значение

Безымянный объект, в свойствах которого содержатся процентные величины каналов звука, управляемые `soundObject`.

Описание

Метод *getTransform( )* возвращает объект со свойствами, говорящими о том, как каналы звуков, управляемых `soundObject`, распределены между левой и правой колонками. Возвращаемый объект имеет свойства `ll`, `lr`, `rl` и `rr`, описываемые в методе *Sound.setTransform( )*.

См. также *Sound.getPan( )*, *Sound.setTransform( )*

Sound.getVolume(), метод Flash 5

возвращает текущее значение громкости

`soundObject.getVolume()`

Возвращаемое значение

Число, указывающее текущую громкость, установленную *setVolume( )*. Обычно находится в диапазоне от 0 (нулевая громкость) до 100 (значение громкости по умолчанию), но может быть и больше.

Описание

Метод *getVolume( )* применяется для определения текущей громкости звуков, управляемых `soundObject`. Обычно *getVolume( )* используется вместе с *setVolume( )* для настройки текущей громкости звука.

---

[1] Здесь есть ошибка реализации, которая может встретиться при разработке объектно-ориентированных приложений. Объект, возвращаемый функцией *getTransform( )*, не наследует никаких свойств и методов класса *Object* и вообще не имеет таких свойств, как `__proto__` и `constructor`. Аналогичная ошибка встречается в объектах, возвращаемых функциями *TextField.getTextExtent* и *Color.getTransform*. – *Примеч. науч. ред.*

Пример

Уменьшим громкость звука на 20:

```
mySound = new Sound();
mySound.setVolume(mySound.getVolume() - 20);
```

См. также *Sound.setVolume( )*

Sound.id3, свойство

Flash 6.0.40.0

доступ к тегам ID3 файла MP3 **только для чтения**

soundObject.id3

Описание

Свойство id3 — это объект, в свойствах которого представлены ID3-теги звукового файла в формате MP3. Файлы MP3 могут загружаться в *soundObject* с некоторого URL с помощью *loadSound( )* или из библиотеки фильма с помощью *attachSound( )*. Свойства этого объекта, соответствующие спецификации id3, перечислены в табл. C.18. Обратите внимание, что Flash Player 6 поддерживает только теги ID3v1 и ID3v1.1.

Таблица C.18. Свойства объекта Sound.id3

Свойство	Описание
album	Название альбома, из которого взята песня
artist	Исполнитель
comment	Любые замечания относительно песни
genre	Целое число, соответствующее описанию музыкального стиля песни. Подробности см. в подразделе «Применение»
songname	Название песни
track	Номер песни в ее альбоме (только ID3v1.1)
year	Год создания песни

Теги ID3 становятся доступны только после полной загрузки файла MP3 (теги ID3v1 и ID3v1.1 хранятся в конце файла MP3). Если в файле MP3 нет тегов ID3 той версии, которую поддерживает Flash Player, свойства объекта id3 получают значение undefined.

Применение

«Стандарт» ID3 не является официальной спецификацией, поддерживаемой какой-либо организацией стандартизации. Однако это распространенная система встраивания метаданных в файл MP3. Ее создал один программист, Эрик Кемп (Eric Kemp), который также составил первый список из 80 названий жанров. Названия жанров сверх этих 80 не формализованы, да и вся система жанров признается неудовлетворительной большинством профессиональных музыкантов. Один из списков жанров можно получить по адресу:

http://www.mp3dev.org/mp3/doc/html/id3.html

Дополнительную информацию о формате ID3 можно найти на сайте *http://www.id3.org/*.

Ошибки реализации

Свойства id3 не сразу становятся доступными в *Sound.onLoad( )*. Вызовите *setInterval( )*, чтобы выяснить допустимое значение duration, прежде чем обращаться к объекту id3, как показано в следующем примере. (Сами свойства id3 имеют значение undefined, если теги ID3 не обнаружены, поэтому их опрос не даст надежных результатов.)

Пример

Следующий код загружает файл MP3 и выводит теги ID3 по окончании загрузки:

```
// ТРЕБУЕТ FLASH PLAYER 6.0.40.0 ИЛИ ВЫШЕ
// Создать текстовое поле для вывода информации ID3.
this.createTextField("output_txt", 1, 100, 100, 400, 300);
output_txt.border = true;

// Создать объект Sound.
song = new Sound();
// Когда загрузится звук, вывести данные ID3.
song.onLoad = function (success) {
  // Сохранить ссылку на текущий звук.
  var theSound = this;

  // Функция, которая выводит данные ID3, когда duration становится действительным.
  function pollForDuration () {
    if (theSound.duration > 0) {
      // Вывести информацию ID3.
      output_txt.text += "Title:  " + theSound.id3.songname + "\n";
      output_txt.text += "Artist: " + theSound.id3.artist + "\n";
      output_txt.text += "Album:  " + theSound.id3.album + "\n";
      output_txt.text += "Year:   " + theSound.id3.year + "\n";
      output_txt.text += "Track:  " + theSound.id3.track + "\n";
      output_txt.text += "Genre:  " + theSound.id3.genre + "\n";
      output_txt.text += "Notes:  " + theSound.id3.comment + "\n";

      // Воспроизводить звук.
      theSound.start();

      // Прекратить опрос действительного duration. Это необходимо из-за
      // ошибки: свойство id3 не сразу становится доступно в Sound.onLoad().
      clearInterval(intID);
    }
  }

  // Если звук загружен нормально...
  if (success == true) {
    // ...периодически проверять допустимость duration.
    var intID = setInterval(pollForDuration, 100);
  }
}

// Загрузить файл MP3
song.loadSound("track1.mp3", false);
```

См. также capabilities.hasMP3, *Sound.loadSound( )*, *Sound.onLoad( )*

Sound.loadSound(), метод

Flash 6

загрузка внешнего звукового файла в проигрыватель

soundObject.loadSound(*url*, *isStreaming*)

Аргументы

url Строка, содержащая сетевой или локальный путь к загружаемому файлу.

isStreaming Булево значение, определяющее тип загружаемого звука: потоковый (`true`) или события (`false`).

Описание

Метод *loadSound( )* загружает внешний звуковой файл с заданного *url*. Во Flash 6 *loadSound( )* позволяет загружать только файлы MP3. Если *isStreaming* имеет значение `true`, загружаемый звук ведет себя как потоковый, и его воспроизведение начинается автоматически, как только заполняется необходимый буфер (см. глобальное свойство `_soundbuftime`). Если *isStreaming* имеет значение `false`, загружаемый звук будет вести себя как звук события и не начнет воспроизводиться автоматически. Для проигрывания звука события следует применять *Sound.start( )* после того, как звук полностью загрузится (см. пример ниже). Остановить воспроизведение загруженного звука (потокового или события) можно с помощью *Sound.stop( )*. В версиях Flash Player до 6.0.40.0 потоковый звук нельзя воспроизводить через *Sound.start( )*. См. дополнительные сведения в описании этого метода.

За ходом загрузки звукового файла можно наблюдать с помощью *getBytesLoaded( )* (см. в описании этого метода пример кода предварительной загрузки). По окончании загрузки звука выполняется *Sound.onLoad( )*. Если по указанному *url* нет звукового файла, *loadSound( )* завершается, не генерируя сообщения об ошибке, и *Sound.onLoad( )* выполняется с параметром *success*, равным `false`.

Для загрузки звука из библиотеки фильма, а не из файла, применяется *attachSound( )*. В отличие от прикрепленных звуков, в один объект *Sound* можно одновременно загрузить несколько звуков. Однако свойства `position` и `duration`, а также метод *start( )* относятся только к звуку, загруженному последним. Поэтому, если надо управлять несколькими звуками, каждый звук следует загружать в собственный экземпляр *Sound*. Если в одном объекте *Sound* есть как прикрепленные, так и загруженные звуки, *start( )* применяется к загруженному звуку, даже если прикрепленный звук добавлен позднее, чем загруженный. Например:

```
// Загрузить track1.mp3
bgMusic.loadSound("track1.mp3");
// После того как загружен track1.mp3, прикрепите track2 с помощью attachSound()
bgMusic.attachSound("track2");
// Проиграть звук...
bgMusic.start();  // Играет track1.mp3 (загруженный звук)
```

Flash может одновременно проигрывать до восьми разных звуков. При загрузке посредством *attachSound( )* каждому звуку требуется свой экземпляр *Sound*.

Узнать, поддерживает ли Flash Player воспроизведение файлов MP3, можно с помощью `System.capabilities.hasMP3`.

Применение

Сжатые файлы MP3 часто содержат незначительные изъяны (например, задержку в начале), влияющие на циклическое воспроизведение во Flash. Для того чтобы компенсировать их, можно задать небольшое смещение загруженного звука события, например:

```
loopSound.start(0.05, 3);
```

Однако при этом не учитывается, что в конце звука может быть участок тишины, поэтому результаты могут быть не очень хорошими. Наилучшее циклическое воспроизведение достигается при использовании сжатия ADPCM или MP3 непосредственно во Flash и прикреплении циклического звука из библиотеки с помощью *attachSound( )*.

Пример

Чтобы загрузить звук, сначала создадим объект *Sound*, затем вызовем метод *loadSound( )*. Часто хорошим решением является определение функции обратного вызова для *onLoad( )*, чтобы ответить на завершение загрузки звука, как это делается в следующем примере. Сначала мы загружаем звук события. Обратите внимание, что обработчик *onLoad( )* должен быть определен раньше, чем произойдет обращение к *loadSound( )*:

```
// Создать объект Sound
music = new Sound();
// Определить функцию обратного вызова onLoad() для нашего объекта
music.onLoad = function () {
  // Здесь размещается код, реагирующий на завершение загрузки звука
  trace("Sound has fully loaded");
  // Это звук события и он полностью загружен, поэтому теперь можно
  // беспрепятственно воспроизвести его
  this.start();
}
// Загрузить звук события
music.loadSound("http://www.somesite.com/song.mp3", false);
```

Теперь загрузим тот же самый звук, но как потоковый. Для потокового звука определение функции обратного вызова *onLoad( )* не так важно, ведь воспроизведение звука начинается автоматически, как только достаточная его часть будет загружена в буфер:

```
// Создать объект Sound
music = new Sound();
// Загрузить потоковый звук
music.loadSound("http://www.somesite.com/song.mp3", true);
```

См. также

`capabilities.hasAudio`, *Sound.getBytesLoaded( )*, *Sound.getBytesTotal( )*, *Sound.onLoad( )*, `_soundbuftime`

Sound.onLoad(), обработчик события

Flash 6

выполняется по окончании загрузки внешнего звука

soundObject.onLoad(*success*)

Аргументы

success Булево значение `true`, если загрузка была успешной, и `false` в противном случае. Если при обращении к *loadSound( )* звуковой файл не был найден, *onLoad( )* выполняется с параметром *success*, равным `false`.

Описание

Обработчик события *onLoad( )* выполняется, когда звуковой файл полностью загружен в *soundObject* в результате предшествовавшего вызова *loadSound( )*. Для того чтобы задействовать *onLoad( )*, мы присваиваем ему пользовательскую функцию обрат-

ного вызова, выполняющую действия, необходимые при нормальной загрузке звука и/или неудаче загрузки. Например:

```
// Создать объект Sound
carSound = new Sound();
// Присвоить функцию обратного вызова
carSound.onLoad = function (success) {
  if (success) {
    // Действия в случае успешной загрузки звука
  } else {
    // Выполнить код, обрабатывающий неудачу загрузки
  }
}
```

Если загружен звук события, то можно начать его воспроизведение с помощью *onLoad( )*:

```
carSound = new Sound();
carSound.onLoad = function (success) {
  if (success) {
    this.start();
  } else {
    displayLoadError();
  }
}
carSound.loadSound("http://www.somesite.com/sounds/driving.mp3", false);
```

Функция обратного вызова – обработчик *onLoad( )* – должна быть определена раньше, чем будет осуществлен вызов *loadSound( )*. Обратите внимание, что в теле обработчика *onLoad( )* можно ссылаться на объект *Sound* с помощью ключевого слова `this`.

Пока звук не загрузился полностью, ход загрузки можно контролировать с помощью *getBytesLoaded( )*.

Ошибки реализации

Свойства `duration`, `position` и `id3` объекта *Sound* вначале оказываются недоступными в обработчике *onLoad( )*. Подробности и возможные решения см. в описаниях этих свойств.

См. также

Sound.getBytesLoaded( ), *Sound.getBytesTotal( )*, *Sound.loadSound( )*, *Sound.onSoundComplete( )*

Sound.onSoundComplete(), обработчик события

Flash 6

выполняется по окончании воспроизведения звука

soundObject.onSoundComplete()

Описание

Обработчик события *onSoundComplete( )* выполняется, когда заканчивается воспроизведение прикрепленного или загруженного звука (то есть когда *soundObject.position == soundObject.duration*). С его помощью можно создать список файлов, воспроизводимых последовательно, или вызвать некоторую анимацию, начинающуюся точно в тот момент, когда заканчивается воспроизведение звука. Чтобы воспользоваться *onSoundComplete( )*, мы присваиваем ему функцию обратного вызова, выпол-

няющую действия, которые должны быть выполнены при завершении воспроизведения звука. Например, следующий код загружает трек «вступительного» голоса за кадром. По окончании трека появляется меню в кадре chapter1:

```
narrative = new Sound();
narrative.attachSound("introduction");
narrative.start();

narrative.onSoundComplete = function () {
  // Введение закончено. Показать меню chapter1.
  _level0.gotoAndStop("chapter1");
}
```

Flash проверяет десять раз на каждый такт частоты смены кадров, не надо ли сгенерировать событие *onSoundComplete( )*. Поэтому, теоретически, фильмы с более высокой частотой смены кадров точнее генерируют события *onSoundComplete( )*.

Пример

Следующий код отображает ход воспроизведения звука в окне Output:

```
// Создать объект Sound
s = new Sound();
// Прикрепить и воспроизвести звук библиотеки
s.attachSound("groove");
s.start();

// Когда звук кончится, вывести "done!" в окно Output
s.onSoundComplete = function () {
  trace("done!");
}

// В каждом кадре проверить, какая часть звука воспроизведена
this.onEnterFrame = function () {
  trace("Played: " + s.position + " of " + s.duration + " milliseconds");
}
```

См. также *Sound*.duration, ***Sound.onLoad( )***, *Sound*.position

Sound.position, свойство

Flash 6

время, в течение которого уже воспроизводится звук, в миллисекундах **только для чтения**

soundObject.position

Описание

Свойство position указывает, как долго воспроизводится прикрепленный или загруженный звук, и измеряется в миллисекундах. Если в один объект *soundObject* загружено несколько звуков, то свойство position отражает первый загруженный звук. Подробности см. далее, в подразделе «Ошибки реализации». Свойство position можно применять для запуска анимаций, синхронизируемых непосредственно с воспроизведением звука. Например, следующий код воспроизводит анимацию взрыва после того, как звук воспроизводился в течение пяти секунд:

```
// Создать звук
introMusic = new Sound();
// Прикрепить библиотечный звук, экспортированный как "song"
introMusic.attachSound("song");
```

```
// В каждом кадре проверить, прошло ли 5 секунд воспроизведения звука
this.onEnterFrame = function () {
  // Если звук проигрывается 5 секунд...
  if (introMusic.position > 5000) {
    // Показать взрыв
    this.gotoAndPlay("explosion_animation");
    // Прекратить проверку position
    delete this.onEnterFrame;
  }
}

// Начать воспроизведение звука
introMusic.start();
```

В случае совместного применения с duration свойство position может также сообщить, какой процент звука воспроизведен к данному моменту (см. следующий пример). Однако в зацикленных звуках position сбрасывается в 0 в каждом очередном цикле. Поэтому чтобы определить процент воспроизведения циклического трека, надо сложить position каждого цикла и сравнить с duration звука, умноженной на количество циклов.

Перед воспроизведением звука свойство position возвращает 0, но после воспроизведения звука position не сбрасывается в 0, а остается равным duration.

Свойство position доступно только для чтения, и им нельзя воспользоваться для воспроизведения звука с заданного места. Для того чтобы воспроизведение звука начиналось в заданный момент времени, следует применять *Sound.start( )*, передавая ему смещение *offset*.

Ошибки реализации

Во Flash 6 свойство position не сбрасывается с каждым звуком, загруженным в один экземпляр Sound. Например, если экземпляр theMusic загружает и воспроизводит полностью файл *song1.mp3*, а затем загружает *song2.mp3*, его свойство position будет равно общей длительности *song1.mp3*. Это препятствие можно обойти, загружая каждый звук в отдельный экземпляр Sound. Там, где это возможно, удаляйте отработавшие экземпляры, чтобы сэкономить память.

Во Flash 6 во время загрузки потокового звука значение свойства position может оказаться неточным (в пределах миллисекунд). Если важна точность position (например, при синхронизации нескольких звуков), используйте непотоковые звуки.

Во Flash 6 свойство position вначале недоступно в *onLoad( )*. Допустимое значение position можно выяснить при помощи функции *setInterval( )*, например:

```
song = new Sound();
song.onLoad = function (success) {
  // Из-за ошибки в реализации position не отображается
  trace("Initial Position: " + this.position);

  // Сохранить ссылку на текущий sound
  var theSound = this;

  // Функция, которая проверяет, что position существует
  function pollForPosition () {
    if (theSound.position >= 0) {
      // Теперь, когда position доступно, показать его
      trace("Actual position: " + theSound.position);
      // Прекратить опрос допустимого position
      clearInterval(intID);
```

```
    }
  }

  // Если звук загружен нормально...
  if (success == true) {
    // Воспроизвести звук
    theSound.start();
    // Периодически проверять допустимость position
    var intID = setInterval(pollForPosition, 10);
  }
}

// Загрузить файл MP3
song.loadSound("track1.mp3", false);
```

Пример

Следующий код добавляет во все объекты *Sound* метод *startFadeIn()*, увеличивающий громкость пропорционально продолжительности воспроизведенной части звука. В представленном виде метод работает лишь с прикрепленными звуками, поскольку в нем нет кода предзагрузчика, необходимого для применения такого эффекта к загруженным звукам:

```
/*
 *     Метод: Sound.startFadeIn()
 *  Описание: Постепенно увеличивает громкость от 0 до 100%.
 * Параметры: fadeDuration. Длительность наращивания громкости в миллисекундах.
 */
Sound.prototype.startFadeIn = function (fadeDuration) {
  this.setVolume(0);
  this.fadeDuration = (fadeDuration == undefined) ? this.duration : fadeDuration;
  this.volInterval = setInterval(this, "fadeIn", 10);
}

/*
 *    Метод: Sound.fadeIn()
 * Описание: Внутренний метод, осуществляющий изменение громкости.
 * Вызывается startFadeIn().
 */
Sound.prototype.fadeIn = function () {
  this.vol = Math.floor(this.position / this.fadeDuration * 100);
  this.setVolume(this.vol);
  // Прекратить вызов fadeIn(), если проигрывание звука закончилось
  if (this.fadeDuration < this.position) {
    clearInterval(this.volInterval);
  }
  trace("volume: " + this.vol);
}

// Применение:
// Создать новый объект Sound.
var music = new Sound();

// Прикрепить звук из библиотеки и воспроизвести его
music.attachSound("song");
music.start();

// Вызвать метод startFadeIn(), задав 2 секунды наращивания громкости (2000 мс)
music.startFadeIn(2000);
```

См. также *Sound.attachSound( )*, *Sound*.duration, *Sound.loadSound( )*

Sound.setPan(), метод

Flash 5

устанавливает баланс звука между левым и правым каналами

soundObject.setPan(*pan*)

Аргументы

pan Число от –100 (левый) до 100 (правый), указывающее распределение между левой и правой колонками для звуков, управляемых *soundObject*. Если передаваемое значение *pan* больше 100, то фактически присваиваемым значением будет 200-*pan*. Если передаваемое значение *pan* меньше –100, то фактически присваиваемым значением будет -200 - *pan*.

Описание

Метод *setPan( )* устанавливает баланс между левым и правым каналами для звуков, управляемых *soundObject*. Изменяя баланс во времени, можно вызвать перемещение звука из одной колонки в другую (это называют *панорамированием – panning*).

Для того чтобы звуки, управляемые *soundObject*, воспроизводились только в левой колонке, надо задать для *pan* значение, равное –100. Если же значение *pan* будет равно 100, то звуки, управляемые *soundObject*, будут воспроизводиться только в правой колонке. Звук распределяется между каналами поровну, если *pan* равно 0.

Обратите внимание, что *setPan( )* действует на все звуки, управляемые *soundObject*. Если *soundObject* представляет собой глобальный звук, то *setPan( )* действует на все звуки фильма. Если *soundObject* привязан к клипу или главной временной диаграмме, то *setPan( )* действует на все звуки в этом клипе или временной диаграмме и во всех содержащихся в них клипах.

Действие *setPan( )* можно изменить только с помощью нового вызова *setPan( )*. Значение, присвоенное *setPan( )*, распространяется на все будущие звуки, управляемые *soundObject*, даже если удалить *soundObject*.

Следующий код добавляет во все клипы новый метод *doPanEffect( )*, вызывающий беспрерывное изменение панорамы звуков клипа между левой и правой колонками:

```
// Метод: MovieClip.doPanEffect()
MovieClip.prototype.doPanEffect = function () {
  var panEffect = new Sound(this);
  var panDirection = "right";
  var panIncrement = 10;

  this.onEnterFrame = function () {
    if (panDirection == "right") {
      newPan = panEffect.getPan() + panIncrement;
      if (newPan > 100) {
        panDirection = "left";
        panEffect.setPan(panEffect.getPan() - panIncrement);
      } else {
        panEffect.setPan(newPan);
      }
    } else {
      newPan = panEffect.getPan() - panIncrement;
      if (newPan < -100) {
        panDirection = "right";
```

```
          panEffect.setPan(panEffect.getPan() + panIncrement);
        } else {
          panEffect.setPan(newPan);
        }
      }
    }
  }

  // Применение:
  theSoundClip_mc.doPanEffect();
```

Следующий пример демонстрирует изменение звука в soundClip_mc в зависимости от положения курсора мыши. Предполагается, что ширина и высота сцены равны 550 и 400 пикселам соответственно. Панорама звука смещается влево или вправо при горизонтальном перемещении мыши, а при вертикальном перемещении мыши увеличивается или уменьшается громкость. Чтобы опробовать этот эффект, создайте на главной временной диаграмме новый клип с именем soundClip_mc и прикрепите код примера к кадру 1 главной временной диаграммы. Затем поместите звуки внутрь soundClip_mc или экспортируйте звук из библиотеки с идентификатором "bgMusic".

```
  // Создать новый объект Sound и прикрепить к нему звук bgMusic
soundClip_mc.volControl = new Sound(soundClip_mc);
soundClip_mc.volControl.attachSound("bgMusic");
soundClip_mc.volControl.start(0, 999);  // Воспроизводить звук в цикле

// В каждом кадре задать громкость и панораму
soundClip_mc.onEnterFrame = function () {
  // Получить горизонтальное положение мыши и установить соответствующий баланс
  var mouseX = (_root._xmouse / 550) * 200;
  this.volControl.setPan(mouseX - 100);
  // Получить вертикальное положение мыши и установить соответствующую громкость
  var mouseY = (_root._ymouse / 400) * 300;
  this.volControl.setVolume(300 - mouseY);
}
```

См. также *Sound.getPan( ), Sound.setTransform( ), Sound.setVolume( )*

Sound.setTransform(), метод

Flash 5

распределяет левый и правый каналы между левой и правой колонками

soundObject.setTransform(*transformObject*)

Аргументы

transformObject

Определяемый пользователем объект, задающий новые настройки каналов в виде ряда свойств.

Описание

Метод *setTransform( )* обеспечивает возможность точного управления выводом звуковых каналов на левую и правую колонки. В принципе, *setTransform( )* имеет сходство с *setPan( )*, но обеспечивает более тонкое управление стереозвучанием.

Стереозвук представляет собой комбинацию двух отдельных звуков – *левого* и *правого каналов*, которые обычно отдельно посылаются на левую и правую колонки. Метод *setTransform( )* позволяет указать, какая часть каждого канала воспроизводится через каждую колонку. Например, можно потребовать, чтобы Flash проигрывал по-

ловину левого канала в левой колонке, не проигрывал никакой части левого канала в правой колонке, а весь правый канал проигрывался в обеих колонках. Или чтобы весь звук левого и правого каналов воспроизводился только в левой колонке.

Для использования *setTransform( )* необходимо сначала создать объект с рядом предопределенных свойств. Свойства описывают распределение левого и правого каналов стереозвука между левой и правой колонками, как описано в табл. С.19.

Таблица С.19. Свойства transformObject

Имя свойства	Значение свойства	Описание свойства
`ll`	от 0 до 100	Процент левого канала, воспроизводимый в левой колонке
`lr`	от 0 до 100	Процент правого канала, воспроизводимый в левой колонке
`rl`	от 0 до 100	Процент левого канала, воспроизводимый в правой колонке
`rr`	от 0 до 100	Процент правого канала, воспроизводимый в правой колонке

После того как объект со свойствами, описанными в табл. С.19, создан, он передается методу *setTransform( )* объекта *Sound*. Значения свойств `transformObject` становятся новыми процентными отношениями выходных мощностей каналов для звуков, которые управляются через `soundObject`.

Текущие значения процентных отношений для некоторого объекта *Sound* можно получить посредством метода *Sound.getTransform( )*.

Пример

```
// Создать новый объект Sound
mySound = new Sound();

// Создать новый родовой объект для использования с setTransform()
transformer = new Object();

// Задать свойства объекта преобразования
transformer.ll = 0;      // Отсутствие левого канала в левой колонке
transformer.lr = 0;      // Отсутствие правого канала в левой колонке
transformer.rl = 0;      // Отсутствие левого канала в правой колонке
transformer.rr = 100;    // Весь правый канал в правой колонке

// Применить новое распределение каналов путем передачи объекта
// преобразования методу setTransform()
mySound.setTransform(transformer);
```

См. также *Sound.getTransform( )*, *Sound.setPan( )*, *Sound.setVolume( )*

Sound.setVolume(), метод

Flash 5

устанавливает громкость звуков, управляемых объектом Sound

`soundObject.setVolume(volume)`

Аргументы

volume Число, характеризующее громкость звука, управляемого `soundObject`, причем 0 означает отсутствие звука (тишину). Чем больше абсолютное значение `volume` (независимо от того, положительное это число или отрицательное), тем громче будут звуки, управляемые

объектом *soundObject*. Например, –50 дает такую же громкость, как 50. Значение громкости по умолчанию равно 100.

Описание

Метод *setVolume( )* изменяет громкость звуков, управляемых *soundObject*. Чтобы вообще убрать звук, надо задать громкость равной 0. Чтобы сделать звук громче, надо увеличить абсолютное значение *volume*. Значения в диапазоне 100–200 обычно дают большую громкость, но максимальное значение не установлено.

Обратите внимание, что *setVolume( )* действует на все звуки, управляемые *soundObject*. Если *soundObject* представляет собой глобальный звук, то *setVolume( )* действует на все звуки фильма. Если же *soundObject* привязан к клипу или главной временной диаграмме, то *setVolume( )* действует на все звуки в этом клипе или временной диаграмме.

Величина громкости, установленная *setVolume( )*, сохраняется, пока установки не будут изменены другим вызовом *setVolume( )*. Установленные *setVolume( )* значения распространяются на все будущие звуки, управляемые *soundObject*, даже если удалить *soundObject*.

Пример

В первом примере просто устанавливается громкость клипа фильма с именем theClip_mc:

```
var mySound = new Sound(theClip_mc);
mySound.setVolume (65);
```

А здесь показано, как создавать кнопки для настройки уровня громкости фильма:

```
var globalSound = new Sound();
var maxVolume = 200;
var minVolume = 0;
var volumeIncrement = 20;

volumeUp_btn.onRelease = function () {
  globalSound.setVolume(Math.min(globalSound.getVolume() + volumeIncrement, maxVolume));
}

volumeDown_btn.onRelease = function () {
  globalSound.setVolume(Math.max(globalSound.getVolume() - volumeIncrement, minVolume));
}
```

См. также *Sound.getVolume( )*, *Sound.stop( )*, *Sound.stopAllSounds( )*

Sound.start(), метод

Flash 5

начинает воспроизведение прикрепленного звука

soundObject.start(*secondOffset*, *loops*)

Аргументы

secondOffset

Число с плавающей точкой, задающее (в секундах) момент, с которого должно начаться воспроизведение звука, прикрепленного к *soundObject* (часто называемое *точкой входа – entry point*). Например, *secondOffset* со значением 1 начинает воспроизведение через одну секунду после фактического начала звука, определенного в библиотеке. Значение по умолчанию равно 0. Точку выхода (то есть время

прекращения воспроизведения звука) задать нельзя. Звук воспроизводится до конца или до остановки вручную. В описании `position` приведен код, прекращающий воспроизведение через *n* миллисекунд.

loops

Положительное целое число, определяющее количество воспроизведений звука, прикрепленного к `soundObject`. Звук воспроизводится один раз, если задано значение 1 (значение по умолчанию), два раза подряд, если 2, и т. д. Часть звука, начиная с `secondOffset`, повторяется до конца столько раз, сколько указано в `loops`. Применимо только к не потоковым звукам.

Описание

Метод *start( )* применяется для воспроизведения программно определяемых звуков, добавленных к `soundObject` посредством *attachSound( )* или *loadSound( )*. Метод *start( )* не воспроизводит все звуки клипа или фильма: он воспроизводит только звук, последним добавленный в `soundObject` с помощью *attachSound( )* или *loadSound( )*, и всегда отдает предпочтение загруженному звуку перед прикрепленным, если в объекте *Sound* имеются и тот и другой.

Для воспроизведения только части звука, добавленного в `soundObject`, задается аргумент `secondOffset`. Для многократного воспроизведения звука, прикрепленного к `soundObject`, следует задать аргумент `loops`. Обычно при многократном вызове *soundObject.start( )* воспроизводится несколько экземпляров одного и того же звука, а не начинается повторное проигрывание звука сначала, но если звук – загруженный потоковый, то его воспроизведение начинается с момента, заданного параметром `secondOffset`. Как правило, останавливать воспроизведение, прежде чем начинать его заново, следует с помощью *stop( )*.

Метод *start( )* работает с загруженными потоковыми звуками только во Flash Player 6.0.40.0 и более поздних версиях. Параметр `loops` не действует для потоковых звуков во всех версиях Player.

Пример

```
// Создать новый объект Sound
boink = new Sound();

// Прикрепить к объекту Sound звук, экспортированный как boink
boink.attachSound("boink");

// Воспроизвести boink полностью; soundOffset по умолчанию равен 0
boink.start();

// Воспроизвести только часть boink, пропустив полсекунды с начала; по умолчанию loops равно 1
boink.start(.5);

// Воспроизвести boink три раза от начала и до конца
boink.start(0, 3);
```

См. также *Sound.loadSound( )*, `Sound.position`, *Sound.stop( )*

Sound.stop(), метод

Flash 5

прекращает воспроизведение всех звуков или заданного прикрепленного звука

```
soundObject.stop()
soundObject.stop(linkageIdentifier)
```

Аргументы

linkageIdentifier

Имя любого звука, прикрепленного к любому объекту *Sound* с таким же *target*, как *soundObject*. Идентификаторы связи задаются в библиотеке через Options → Linkage.

Описание

При вызове без *linkageIdentifier* метод *stop( )* прекращает все звуки, управляемые *soundObject*. Если *soundObject* представляет собой глобальный звук, то *stop( )* прекращает воспроизведение всех звуков в фильме, а если *soundObject* был создан с параметром *target*, то прекращает воспроизведение всех звуков в *target*.

При вызове с аргументом *linkageIdentifier* метод *stop( )* прекращает воспроизведение только конкретного звука, заданного этим аргументом. В этом случае *linkageIdentifier* должен быть звуком, прикрепленным к объекту *Sound* с помощью *attachSound( )*. Однако останавливаемый звук не обязательно должен быть прикреплен к самому *soundObject*. Он может быть прикреплен к *любому* объекту *Sound*, у которого такой же *target*, как у *soundObject*. Или, если *soundObject* создавался без аргумента *target* (то есть как глобальный объект *Sound*), останавливаемый звук может быть прикреплен к любому другому глобальному объекту *Sound*.

Пример

```
// Создать глобальный объект Sound
mySound = new Sound();

// Прикрепить к объекту звук doorbell
mySound.attachSound("doorbell");

// Прекратить все звуки в фильме
mySound.stop();

// Воспроизвести doorbell
mySound.start();

// Остановить только doorbell
mySound.stop("doorbell");

// Создать другой глобальный объект Sound
myOtherSound = new Sound();

// Прикрепить звук doorknock к этому объекту
myOtherSound.attachSound("doorknock");

// Воспроизвести doorknock
myOtherSound.start();

// Теперь остановить doorknock через mySound, а не myOtherSound.
// Это выполнится, поскольку у обоих объектов Sound одинаковый target.
mySound.stop("doorknock");
```

См. также *Sound.attachSound( )*, *Sound.start( )*, *stopAllSounds( )*

_soundbuftime, глобальное свойство

Flash 4

длительность потокового звука для предварительной загрузки в секундах — **чтение/запись**

```
_soundbuftime
```

Описание

Свойство `_soundbuftime` представляет собой целое число, задающее длительность (в секундах) потокового звука, который должен быть загружен до начала воспроизведения (по умолчанию 5 секунд). Применимо как к звукам временной диаграммы, так и к звукам, загружаемым на этапе выполнения через *Sound.loadSound( )*.

Flash синхронизирует воспроизведение фильма с потоковым звуком, чтобы гарантировать, скажем, согласованность движений губ персонажа мультипликации с сопровождающей звуковой дорожкой. Анимация останавливается, пока не будет загружено `_soundbuftime` секунд потокового звука; поэтому установка большого значения может приводить к чрезмерной задержке начала воспроизведения при низкоскоростном соединении. Поскольку поток в сети может оказаться медленным или кратковременно прерываться, задание короткого `_soundbuftime` может приводить к пропускам в звуке (если в буфер загружено недостаточно звуковых данных, звук может временно прерываться). Идеальная настройка может отличаться от фильма к фильму и зависеть от сложности графики, настройки качества звука и пропускной способности соединения конечного пользователя с Интернетом. Значение по умолчанию (5 секунд) обычно вполне удовлетворительно, но в отдельных случаях для нахождения наилучшей установки могут потребоваться эксперименты. Объем буфера потока может меняться во время воспроизведения, но это свойство является глобальным и не может устанавливаться отдельно для разных звуков.

Пример

```
_soundbuftime = 10;  // Буферизовать 10 секунд аудиоданных
```

См. также Объект *Sound*, *stopAllSounds( )*

Stage, объект

Flash 6

доступ к размеру, масштабу и выравниванию фильма

Свойства

align Расположение фильма во Flash Player.

height Высота доступной области отображения Flash Player в пикселах либо высота документа этапа разработки, что зависит от значения `scaleMode`.

scaleMode Определяет способ, которым размеры фильма устанавливаются относительно проигрывателя.

showMenu Булево значение, определяющее, какие элементы содержит контекстное меню Flash Player.

width Ширина доступной области отображения Flash Player в пикселах либо ширина документа этапа разработки, что зависит от значения `scaleMode`.

Методы

addListener( ) Регистрирует объект в качестве приемника событий *onResize( )*.

removeListener( )

Отменяет передачу уведомлений о событиях указанному приемнику.

События приемника

onResize( ) Происходит при изменении размеров фильма.

Описание

Объект *Stage* предоставляет характеристики области отображения во время выполнения Flash Player. Он сообщает о текущих размерах фильма и об их изменении и предоставляет возможность управления глобальными настройками фильма, например содержимым контекстного меню проигрывателя (в Windows активизируемого щелчком правой кнопки мыши или щелчком при нажатой клавише Ctrl на Макинтоше). В основном объект *Stage* используется для настройки зрительного представления фильма при изменении размеров броузера или автономного проигрывателя.

Stage представляет собой готовый объект, а не класс. Доступ к его свойствам и методам осуществляется непосредственно, например:

```
trace(Stage.width);  // Выводит ширину фильма
```

См. также *MovieClip.*`_height`, *MovieClip.*`_width`

Stage.addListener(), метод

Flash 6

регистрирует объект в качестве приемника событий onResize()

`Stage.addListener(`*`listener`*`)`

Аргументы

listener Объект, в котором определен метод *onResize( )*.

Описание

Метод *addListener( )* помещает `listener` в список объектов, оповещаемых о событии *onResize( )* объекта *Stage*. Аргумент `listener` – это объект любого класса, в котором определен метод с именем *onResize*; метод вызывается, когда происходит соответствующее событие для *Stage*. Экземпляры встроенных классов, например *TextField*, *Button* или *MovieClip*, могут реагировать на *Stage.onResize( )*. Прекратить получение событий объектом `listener` можно с помощью метода *Stage.removeListener( )*.

На каждое событие *onResize( )* могут независимо реагировать несколько объектов. Если в *Stage* зарегистрировано несколько объектов, их методы вызываются в том порядке, в котором добавлялись приемники.

Общие сведения о работе приемников см. в главе 10.

Пример

Следующий код присваивает метод *onResize( )* клипу `ball_mc` и регистрирует этот клип в качестве приемника события *onResize( )*. Обратите внимание, что свойство `Stage.scaleMode` должно иметь значение `"noScale"`, чтобы генерировалось событие *onResize( )*:

```
Stage.align = "LT";
Stage.scaleMode = "noScale";
ball_mc.onResize = function () {
  // Поместить ball_mc в центр Stage (предполагается, что
  // центр содержимого ball_mc находится в его точке регистрации).
  // Внутри метода 'this' указывает на ball_mc.
  this._x = Stage.width/2;
```

```
    this._y = Stage.height/2;
  }
  Stage.addListener(ball_mc);
```

См. также *Stage.onResize( )*, *Stage.removeListener( )*, `Stage.scaleMode`; глава 10

Stage.align, свойство

Flash 6

выравнивает фильм внутри Flash Player

`Stage.align`

Описание

Если фильм встроен в веб-страницу, то теги HTML `OBJECT` и `EMBED` могут указывать размеры, не совпадающие с размерами документа Flash. При воспроизведении фильма в автономном проигрывателе размеры последнего могут быть изменены пользователем. В обоих случаях строка `align` определяет расположение фильма относительно имеющегося во Flash Player пространства. По умолчанию `align` – пустая строка, что указывает на необходимость центрирования фильма в проигрывателе по горизонтали и вертикали. В табл. С.20 перечислены возможные значения параметра выравнивания.

Таблица С.20. Значения свойства Stage.align

Свойство align	Описание	Выравнивание по горизонтали	Выравнивание по вертикали
"" (пустая строка)	по центру	по центру	по центру
"B"	внизу по центру	по центру	внизу
"T"	вверху по центру	по центру	вверху
"L"	с левого края по центру	с левого края	по центру
"LB"	левый нижний угол	с левого края	внизу
"LT"	левый верхний угол	с левого края	вверху
"R"	с правого края по центру	с правого края	по центру
"RB"	правый нижний угол	с правого края	внизу
"RT"	правый верхний угол	с правого края	вверху

Эффект `align` наиболее очевиден (и полезен), когда `scaleMode` имеет значение «noScale»; в этом случае фильм не меняет размеры в проигрывателе, а меняет только местоположение.

При изменении размеров проигрывателя документ *.swf*, который в нем содержится, автоматически меняет свое расположение в соответствии со значением `align`. Чтобы задать иное расположение элементов в фильме при изменении его размеров, применяется событие приемника *onResize( )*.

Если фильм встроен в веб-страницу, теги `OBJECT` и `EMBED` тоже могут задавать значение `align`; любые присваивания в ActionScript имеют приоритет над значениями HTML.

Применение

ActionScript исправляет неточно заданное значение `align`, чтобы оно совпало с наиболее близким корректным значением. Например, `"tl"` превращается в `"LT"`, а `"jjjt"`

превращается в "T". Однако чтобы сделать код понятным, следует всегда использовать документированные значения.

Пример

Следующий код прикрепляет фильм к левому верхнему углу проигрывателя:

```
Stage.scaleMode = "noScale";
Stage.align = "LT";
```

См. также *Stage.onResize( )*, Stage.scaleMode

Stage.height, свойство

Flash 6

высота доступной области показа Flash Player в пикселах либо высота документа этапа разработки **только для чтения**

Stage.height

Описание

Целочисленное свойство height содержит высоту фильма в пикселах в зависимости от значения Stage.scaleMode. Если scaleMode имеет значение "noScale", height возвращает размер доступной области отображения Flash Player, то есть размеры окна автономного проигрывателя или текущий размер области OBJECT/EMBED в броузере. Когда значение scaleMode отлично от "noScale", height сообщает размер фильма Flash, заданный на этапе разработки в Modify → Document → Dimensions.

В режиме Test Movie scaleMode по умолчанию равно "noScale", поэтому height может не соответствовать размерам, заданным на этапе разработки. В броузере или Standalone Player scaleMode по умолчанию равно "showAll", поэтому height соответствует размерам, заданным на этапе разработки, если явно не установить scaleMode равным "noScale".

Для того чтобы получать уведомление при изменении Stage.height, необходимо добавить приемники события *Stage.onResize( )*.

Применение

Не смешивайте Stage.height с _root._height. Первое представляет собой высоту доступного пространства проигрывателя или размеры заданного документа, а второе – высоту всех объектов на главной временной диаграмме фильма.

Ошибки реализации

Если scaleMode равно "noScale" и обращение к height происходит из нескольких первых кадров фильма, это свойство может давать неверное значение. Для надежности пропустите несколько кадров, прежде чем обращаться к height, или, что еще лучше, обращайтесь к нему только внутри приемника *onResize( )*.

В режиме Test Movie значение height оказывается на четыре пиксела меньше, чем в действительности.

Пример

Следующий код создает текстовое поле и располагает его в центре сцены:

```
Stage.scaleMode = "noScale";
Stage.align = "LT";

this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "Derek got a haircut.";
```

```
theField_txt._x = Stage.width/2 - theField_txt._width/2;
theField_txt._y = Stage.height/2 - theField_txt._height/2;
```

См. также

MovieClip.createTextField( ), *MovieClip*._height, *Stage.addListener( )*, *Stage.onResize( )*, Stage. width, класс *TextField*

Stage.onResize(), событие приемника

Flash 6

вызывается при изменении размеров фильма

listenerObject.onResize()

Описание

Событие *onResize( )* генерируется, когда Stage.scaleMode имеет значение "noScale" и изменяются размеры проигрывателя. Размеры проигрывателя изменяются в следующих случаях:

- Пользователь изменяет размер автономного проигрывателя или вызывается *fscommand ("fullscreen", "true")*.
- Пользователь изменяет размер окна броузера с находящимся в нем фильмом с помощью процентных величин в атрибутах HEIGHT или WIDTH тегов OBJECT/EMBED.

Событие *onResize( )* позволяет динамически изменять расположение элементов фильма относительно пространства, доступного в данный момент. Благодаря этому можно создавать «растягивающиеся» структуры, ведущие себя подобно таблицам на страницах HTML, ширина колонок которых задана в процентных отношениях. Фильм в окне броузера генерирует события изменения размера, только если задать для него в HTML значения HEIGHT или WIDTH в виде процентов. Например:

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
 codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab"
 WIDTH="100%"
 HEIGHT="75">
 <PARAM NAME=movie VALUE="theMovie.swf">

 <EMBED src="theMovie.swf"
  WIDTH="100%"
  HEIGHT="75"
  TYPE="application/x-shockwave-flash"
  PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
 </EMBED>
</OBJECT>
```

Обратите внимание, что можно задать один размер фиксированным в пикселах (HEIGHT="75"), а другой – в процентах (WIDTH="100%"). При первоначальной загрузке фильма в броузер *onResize( )* не генерируется, даже если задать размер в процентах. Запустить код для первоначального размещения элементов в фильме надо вручную.

В следующем примере регистрируется приемник события, реагирующий на *onResize( )*. Обратите внимание, что scaleMode надо обязательно задать равным "noScale":

```
Stage.align = "T";             // Поместить фильм вверху в центре
Stage.scaleMode = "noScale";   // Предотвратить изменение масштаба (обязательно!)

// Создать объект приемника
resizeListener = new Object();
resizeListener.onResize = function () {
```

```
  trace("New width: " + Stage.width);
};
Stage.addListener(resizeListener);
```

На событие *onResize( )* может реагировать любое количество объектов приемников. Например, при динамическом расположении элементов каждый визуальный объект (текстовое поле, кнопка или клип) может быть зарегистрирован в качестве приемника *onResize( )*, чтобы каждый элемент можно было расположить самостоятельно при изменении размеров фильма, как описано в главе 10. См. в примере ниже код, который перемещает клип при изменении размеров фильма.

Получение события *onResize( )* объектом приемника можно прекратить посредством метода *Selection.removeListener( )*:

```
Stage.removeListener(resizeListener);
```

Изменения представления, осуществляемые внутри приемника *onResize( )*, отображаются немедленно независимо от частоты смены кадров. Вызывать для этого *updateAfterEvent( )* не требуется.

Пример

Следующий код прикрепляет `logo_mc` к левому верхнему углу области воспроизведения проигрывателя, даже если изменяется размер сцены:

```
Stage.align = "LT";
Stage.scaleMode = "noScale";
stageListener = new Object();
stageListener.onResize = function () {
  logo_mc._x = Stage.width - logo_mc._width;
  logo_mc._y = 0;
}
Stage.addListener(stageListener);
```

Более пространный пример поля, изменяющего размеры, можно загрузить с:

http://www.moock.org/webdesign/flash/actionscript/resizableMovie/resizableMovie.zip

Ошибки реализации

Когда вызов *fscommand("fullscreen", "true")* генерирует *onResize( )*, значения `Stage.width` и `Stage.height` внутри функции обратного вызова оказываются неверными. Чтобы определить фактический размер автономного проигрывателя в полноэкранном режиме, узнайте при помощи *setInterval( )* значения `width` и `height` или выясните размеры экрана непосредственно с помощью `System.capabilities.screenResolutionX` и `System. capabilities.screenResolutionY`.

См. также

Stage.addListener( ), `Stage.height`, *Stage.removeListener( )*, `Stage.scaleMode`, `Stage.width`; глава 10.

Stage.removeListener(), метод

Flash 5

отменяет отправку уведомлений о событии указанному приемнику

`Stage.removeListener(`*`listener`*`)`

Аргументы

listener Объект, ранее зарегистрированный в качестве приемника в *Selection*.

Возвращаемое значение

Булево значение `true`, если *`listener`* найден и удален, и `false`, если *`listener`* не был зарегистрирован в качестве приемника в `Stage`. Возвращаемое значение полезно при отладке кода, удаляющего приемники.

Описание

Метод *removeListener( )* прекращает получение объектом *`listener`* уведомлений о событиях `Stage`. Вызывать его следует только после того, как *`listener`* зарегистрирован в качестве приемника событий с помощью *addListener( )*. Объект приемника не уничтожается и может многократно удаляться и добавляться. Оператор *delete* позволяет уничтожить приемник, который надо предварительно удалить с помощью *removeListener( )*.

Пример

Приемники событий следует удалять лишь тогда, когда выполняемая ими задача больше не нужна. Например, у текстового поля могут быть два режима показа в приложении – один с «процентной шириной» (растянуть поле до ширины фильма), а другой с «фиксированной шириной» (сохранять ширину постоянной). В зависимости от текущего режима можно добавить в поле приемник события или удалить его:

```
// Задать scaleMode для работы onResize().
Stage.align = "LT";
Stage.scaleMode = "noScale";

// Создать текстовое поле.
this.createTextField("body_txt", 1, 5, 0, 200, 20);
body_txt.border = true;
body_txt.type = "input";
body_txt.onResize = function () {
  this._width = Stage.width - 10;
};

// Функция переключения режима. Динамически добавляет и удаляет приемник.
function setEditMode (mode) {
  if (mode == "fixed") {
    Stage.removeListener(body_txt);
  } else if (mode == "percent") {
    Stage.addListener(body_txt);
  }
}

// Задать режим.
setEditMode("percent");
```

См. также *Stage.addListener( )*, *Stage.onResize( )*; глава 10

Stage.scaleMode, свойство

Flash 6; начиная с Flash 3 частично доступно через fscommand()

определяет режим изменения размеров фильма относительно проигрывателя — **чтение/запись**

`Stage.scaleMode`

Описание

Свойство `scaleMode` имеет строковый тип и управляет режимом изменения масштаба фильма (изменения размеров) при несоответствии доступного пространства во Flash

Player и размеров сцены фильма, заданных на этапе разработки через Modify → Document → Dimensions. Допустимы следующие значения scaleMode:

«exactFit» Прикрепляет края фильма к краям проигрывателя, изменяя при необходимости отношение геометрических размеров фильма, чтобы заполнить все свободное пространство.

«showAll» Подгоняет ширину или высоту фильма под меньшее из значений ширины или высоты проигрывателя. Фильм, подгоняемый под размеры проигрывателя, не кадрируется и не искажается, но вокруг него появляются границы (в соответствии с текущим значением Stage.align). По умолчанию границы будут вверху и внизу или слева и справа.

«noBorder» Подгоняет ширину или высоту фильма под большее из значений ширины или высоты проигрывателя. Если соотношение геометрических размеров фильма не совпадает с соотношением геометрических размеров проигрывателя, фильм кадрируется в соответствии с текущим значением Stage.align. По умолчанию за пределами кадра остаются либо верх и низ, либо оба края.

«noScale» Сохраняет размер фильма, установленный на этапе разработки. Если проигрыватель меньше фильма, фильм обрезается. Если проигрыватель больше фильма, на свободном пространстве отображается фон фильма. Размещением фильма управляет свойство Stage.align.

По умолчанию свойство scaleMode для автономного проигрывателя и подключаемого модуля броузера равно "showAll". В режиме Test Movie его значение по умолчанию равно "noScale".

Обычно свойство scaleMode применяется совместно со свойством align и событием *onResize( )* в фильмах с динамически располагаемым содержимым.

Если фильм встроен в веб-страницу, то теги OBJECT или EMBED также могут устанавливать значение scaleMode; значения, присваиваемые в ActionScript, имеют приоритет над значениями HTML.

Применение

Во Flash Player версий с 3 по 5 эквивалент Stage.scaleMode = "noScale" можно получить командой *fscommand("allowscale", "false")*.

Пример

```
// Исказить фильм, чтобы он заполнил всю область показа проигрывателя
Stage.scaleMode = "exactFit";

// Запретить всякое масштабирование независимо от размеров проигрывателя
Stage.scaleMode = "noScale";
```

См. также *Stage.addListener( )*, Stage.align, *Stage.onResize( )*

Stage.showMenu, свойство

Flash 6; начиная с Flash 3 частично доступно через fscommand()

булево значение, определяющее, какие элементы содержатся в контекстном меню Flash Player **чтение/запись**

Stage.showMenu

Описание

Булево свойство showMenu определяет, будет ли контекстное меню Flash Player содержать полный список элементов (true) или минимальный список элементов (false). По умолчанию имеет значение true, но ему часто присваивают значение false, чтобы пользователь не мог управлять фильмом нежелательным образом, например останавливать, перематывать назад или увеличивать масштаб. Доступ к контекстному меню осуществляется с помощью щелчка правой кнопкой в Windows или Ctrl-щелчка на Макинтоше.

Если showMenu равно true, то во Flash 6 появляются следующие пункты меню:

Zoom In
Zoom Out
100%
Show All
Quality
Play
Loop
Rewind
Forward
Back
Settings...
Print...
About Macromedia Flash Player 6

Если showMenu имеет значение false, то доступны следующие пункты меню:

Settings...
About Macromedia Flash Player 6

В обоих случаях при выборе пункта Settings появляется одноименное диалоговое окно, в котором пользователь может задать объем дискового пространства для *SharedObject* и настроить камеру и микрофон.

Встроенных средств, позволяющих полностью отключить контекстное меню Flash Player, нет. Но есть ряд средств сторонних разработчиков, с помощью которых можно убрать меню из автономного проигрывателя:

http://www.flashjester.com/software/jugglor/ и http://www.northcode.com/studio/

Если фильм встроен в веб-страницу, то теги OBJECT и EMBED тоже могут задавать значение showMenu; любые присваивания в ActionScript имеют приоритет над значением. заданным в коде HTML.

Применение

Во Flash версий с 3 по 5 showMenu можно задать с помощью *fscommand( )*, например *fscommand ("showMenu", "false")*.

Пример

```
// Сокращает контекстное меню до минимума элементов
Stage.showMenu = false;
```

Stage.width, свойство

Flash 5

ширина доступной области воспроизведения Flash Player в пикселах или ширина документа этапа разработки

только для чтения

```
Stage.width
```

Описание

Целочисленное свойство `width` содержит ширину фильма в пикселах в зависимости от значения `Stage.scaleMode`. Если `scaleMode` равно `"noScale"`, то `width` возвращает размер доступной области воспроизведения Flash Player – окна автономного проигрывателя или текущий размер области `OBJECT/EMBED` в броузере. Когда значение `scaleMode` отлично от `"noScale"`, `width` сообщает размер фильма Flash, который был задан на этапе разработки в Modify → Document → Dimensions.

В режиме Test Movie `scaleMode` по умолчанию равно `"noScale"`, поэтому `width` может не соответствовать размерам, заданным на этапе разработки. В броузере или Standalone Player `scaleMode` по умолчанию равно `"showAll"`, поэтому `width` соответствует размерам, заданным на этапе разработки, если явно не установить `scaleMode` равным `"noScale"`.

Чтобы получать уведомление при изменении `Stage.width`, надо добавить приемники события *Stage.onResize( )*.

Применение

Не спутайте `Stage.width` с `_root._width`. Первое представляет собой ширину доступного пространства проигрывателя или размеры заданного документа, а второе – ширину всех объектов на главной временной диаграмме фильма.

Ошибки реализации

Если `scaleMode` равно `"noScale"` и обращение к свойству `width` происходит из нескольких первых кадров фильма, то возвращаемое им значение может быть неверным. Для надежности пропустите несколько кадров, прежде чем обращаться к `width`, или, что еще лучше, обращайтесь к нему только внутри приемника *onResize( )*.

В режиме Test Movie значение `width` на четыре пиксела меньше действительного.

Пример

Следующий код с помощью методов черчения *MovieClip* проводит линию по всей ширине сцены на 30 пикселов ниже его верхнего края:

```
Stage.scaleMode = "noScale";
Stage.align = "LT";

this.lineStyle(1);
this.moveTo(0, 30);
this.lineTo(Stage.width, 30);
```

См. также *MovieClip.*`_width`, *Stage.addListener()*, `Stage.height`, *Stage.onResize()*

startDrag(), глобальная функция

Flash 4

заставляет фильм или клип следовать за курсором мыши

```
startDrag(target)
startDrag(target, lockCenter)
startDrag(target, lockCenter, left, top, right, bottom)
```

Аргументы

target Строка или ссылка, указывающие путь к фильму или экземпляру клипа, который должен следовать за курсором мыши (в строковом контексте ссылки на клипы преобразуются в пути).

lockCenter Булева величина, определяющая, должна ли точка регистрации *target* помещаться под указателем мыши (`true`) или перемещаться относительно исходного положения (`false`).

left Число, задающее минимальную координату x, левее которой не может перемещаться точка регистрации *target*.

top Число, задающее минимальную координату y, выше которой не может перемещаться точка регистрации *target*.

right Число, задающее максимальную координату x, правее которой не может перемещаться точка регистрации *target*.

bottom Число, задающее максимальную координату y, ниже которой не может перемещаться точка регистрации *target*.

Описание

Функция *startDrag( )* вызывает видимое перемещение *target* за курсором мыши по проигрывателю (называемое также *буксировкой* (*dragging*) клипа). Перемещение буксируемого клипа можно ограничить рамкой, координаты которой передаются как аргументы функции *startDrag( )*. Координаты ограничивающей рамки задаются относительно холста, на котором расположен *target*. Если холстом является сцена главного фильма, то левый верхний угол этой сцены находится в точке (0,0). Если холстом является клип, то (0,0) обозначает точку регистрации холста клипа. Обратите внимание, что координатная система Flash переворачивает ось Y декартовой системы кординат: значения y *возрастают* в направлении к нижней части экрана и *уменьшаются* в направлении верхней части экрана. Отрицательные значения y расположены *выше* начала координат (то есть над осью X).

Буксировку можно прекратить в любой момент с помощью функции *stopDrag( )*. Одновременно можно буксировать только один клип или фильм, поэтому вызов функции *startDrag( )* с новым *target* автоматически прекращает текущую операцию буксировки. При этом во время буксировки фильма или клипа вместе с ним перемещаются все содержащиеся в нем клипы.

Пример

```
// Перетащить ball, ограничив его перемещение левым верхним углом сцены
startDrag("ball", true, 0, 0, 225, 200);
```

См. также *MovieClip.*`_dropTarget`, *Movieclip.startDrag( )*, *stopDrag( )*

stop(), глобальная функция

Flash 2

останавливает головку воспроизведения в текущем кадре

```
stop()
```

Описание

Функция *stop( )* – это простая, но базовая функция, останавливающая воспроизведение фильма или клипа. Это глобальный аналог метода *MovieClip.stop( )* и заклятый

враг *play( )*. Она часто применяется для того, чтобы дать пользователю возможность сделать выбор в графическом меню. Даже когда фильм остановлен, проигрыватель продолжает воспроизводить звуки, реагировать на действия пользователя и обрабатывать события. Flash по-прежнему отображает на экране изменения в других клипах или кнопках, даже если головка воспроизведения остановлена. Аналогично изменения, выполненные в обработчиках событий или через *setInterval( )*, отображаются на экране, даже если головка воспроизведения остановлена. Функция *stop( )* просто не позволяет головке воспроизведения перемещаться вдоль текущей временной диаграммы.

См. также *MovieClip.stop( )*, *play( )*

stopAllSounds(), глобальная функция

Flash 3

выключает звук в фильме

```
stopAllSounds()
```

Описание

Функция *stopAllSounds( )* заглушает все звуки, воспроизводящиеся в данное время в фильме, независимо от глубины их вложенности в клипы. Это относится ко всем звукам фильма, в том числе к программно генерируемым объектам *Sound*. Более точное управление остановкой, запуском и громкостью звука обеспечивает класс *Sound*.

Обратите внимание, что *stopAllSounds( )* оказывает только временное воздействие. Любой звук, который начинается после вызова *stopAllSounds( )*, будет воспроизводиться нормально. Поскольку звуки кнопок не управляются классом *Sound*, нет возможности навсегда прекратить воспроизведение звуков в фильме. Однако если задать для громкости значение 0 в глобальном для проигрывателя объекте *Sound*, то воспроизведение всех звуков, не связанных с кнопками, прекратится.

См. также *Sound.setVolume( )*, *Sound.stop( )*

stopDrag(), глобальная функция

Flash 4

завершает текущую операцию буксировки

```
stopDrag()
```

Описание

Функция *startDrag( )* вызывает перемещение клипа вслед за курсором мыши по сцене. Операция *stopDrag( )* прекращает перемещение клипа за курсором мыши. Одновременно можно буксировать лишь один клип или фильм, поэтому для *stopDrag( )* не требуется аргумент `target`: прекращается любая текущая операция буксировки.

Функция *stopDrag( )* применяется совместно со *startDrag( )* для создания во Flash простых интерфейсов «drag-and-drop», которые демонстрируются в разделе «Interface Widgets» (элементы интерфейса) сетевого хранилища кода. Если буксируемый клип сбрасывается на другой клип, то узнать, на какой клип он был сброшен, позволяет свойство `MovieClip._droptarget`.

Пример

Следующий код начинает буксировку клипа `window_mc` при нажатии над ним кнопки и сбрасывает его, когда кнопка отпускается. Обратите внимание, что мы останавли-

ваем буксировку клипа при обнаружении событий *onRelease( )* или *onReleaseOutside( )*, что гарантирует сброс клипа независимо от того, находится ли курсор мыши над клипом, когда он освобождается:

```
window_mc.onPress = function () {
  this.startDrag();
}

window_mc.onRelease = window_mc.onReleaseOutside = function () {
  stopDrag();
}
```

См. также `MovieClip._dropTarget`, *MovieClip.stopDrag( )*, *startDrag( )*

#strict, прагма

Flash 5 (не документирована)

включает более строгое соответствие стандарту ECMA-262

`#strict`

Описание

Недокументированная и непостоянная функция. Прагма `#strict` вынуждает Flash Player более строго следовать спецификации ECMA-262, на которой основан ActionScript. Оказывает воздействие на преобразование типов данных *undefined* и *string* и включает полный учет регистра символов.

Известно, что прагма `#strict` приводит к аварийному завершению Internet Explorer под Windows при работе Flash Player 6, и потому применять ее не следует. Во время написания этой книги Macromedia официально аннулировала поддержку `#strict` во Flash Player 6.0.61.0 после того, как открытое бета-тестирование Flash Player 6.0.60.48 выявило существующие проблемы. Подробная информация о прагме `#strict` приведена на *http://www.moock.org/asdg/technotes/strict*.

См. также Приложение D

String(), глобальная функция

Flash 5

преобразовует значение к типу данных String

`String(value)`

Аргументы

value

Выражение, содержащее значение, которое должно быть преобразовано в строку.

Возвращаемое значение

Результат преобразования *value* в элементарную строку.

Описание

Функция *String( )* преобразует аргумент в элементарное строковое значение и возвращает это преобразованное значение. Результаты преобразования различных типов данных в элементарную строку описываются в табл. 3.2 главы 3. Обычно необхо-

димости в применении функции *String( )* нет: если требуется, ActionScript автоматически преобразует значения к типу *string*.

Не следует путать глобальную функцию *String( )* с конструктором класса *String*. Первая преобразует выражение в строку, тогда как последний является классом-оболочкой для элементарных строковых данных, позволяя применять к ним свойства и методы.

Применение

Обратите внимание на возможное появление функции *String( )* в файлах *.fla* Flash 4, преобразованных в формат более поздних версий Flash. Сведения об обращении с типами данных при преобразовании файлов Flash 4 в файлы Flash 5 и более поздних версий можно найти в разделе «Преобразование типов данных Flash 4 во Flash 5» приложения С.

См. также Класс *String*; «Явное преобразование типов», глава 3

String, класс

Flash 5; во Flash 6 повышена эффективность

класс-оболочка для примитивного строкового типа

Конструктор

`new String(value)`

Аргументы

value Выражение, которое должно быть разрешено и при необходимости преобразовано в строку, а затем помещено в объект *String*.

Свойства

length Количество символов в строке.

Методы класса

Следующий метод вызывается через сам класс *String*, а не через объект класса *String*:

fromCharCode( )

Генерирует строку из одной или нескольких кодовых позиций Unicode.

Методы

Следующие методы объекта вызываются через экземпляр класса *String*:

charAt( ) Возвращает символ в заданной позиции строки.

charCodeAt( ) Возвращает кодовую позицию символа в заданной позиции строки.

concat( ) Объединяет один или более элементов в строку.

indexOf( ) Находит первое вхождение указанной подстроки в строку.

lastIndexOf( ) Находит последнее вхождение указанной подстроки в строку.

slice( ) Извлекает из строки подстроку, исходя из положительных или отрицательных позиций символов.

split( ) Преобразует строку в массив.

substr( ) Извлекает из строки подстроку, исходя из начальной позиции и длины.

substring( ) Извлекает из строки подстроку, исходя только из положительных позиций символов.

toLowerCase( ) Возвращает версию строки, записанную буквами нижнего регистра.

toUpperCase( ) Возвращает версию строки, записанную буквами верхнего регистра.

Описание

Класс *String* решает несколько задач:

- Позволяет обращаться к свойству `length` для строк и выполнять операции над строками, такие как *indexOf( )* и *slice( )*. Объекты *String* автоматически создаются (и в конечном счете уничтожаются) интерпретатором, когда с элементарным строковым значением вызывается метод.
- Может применяться для преобразования в строку элемента данных любого типа.
- Применяется для доступа к методу класса *fromCharCode( )* при создании новой строки, исходя из заданных кодовых позиций Unicode.
- Применяется для создания объекта *String*, содержащего элементарное строковое значение в безымянном внутреннем свойстве, однако особого смысла в этом нет.

Во Flash 6 производительность (скорость выполнения) методов класса *String* значительно повышена по сравнению с Flash 5. Скорость выполнения большинства методов *String* во Flash Player 6 в 50–1000 раз выше, чем во Flash Player 5.

Применение

На практике конструктор класса *String* применяется в основном для преобразования в строки других типов данных. Подробнее см. в описании глобальной функции *String( )*.

См. также «Строковый тип данных», глава 4

String.charAt(), метод

Flash 5

возвращает символ в указанной позиции строки

`string.charAt(index)`

Аргументы

index Целочисленная позиция извлекаемого символа, которая должна находиться в диапазоне от 0 (первый символ) до `string.length - 1` (последний символ).

Возвращаемое значение

Символ в позиции `index` строки `string`.

Описание

Метод *charAt( )* определяет символ, находящийся в определенной позиции (`index`) строки.

Пример

```
trace("It is 10:34 pm".charAt(1));  // Выводит: "t" (вторая буква)
var country = "Canada";
trace(country.charAt(0));           // Выводит: "C" (первая буква)
```

```
// Эта функция удаляет из строки все пробелы и возвращает результат
function stripSpaces (inString) {
  var outString = "";
  for (i = 0; i < inString.length; i++) {
    if (inString.charAt(i) != " ") {
      outString += inString.charAt(i);
    }
  }
  return outString;
}
```

См. также

String.charCodeAt( ), *String.indexOf( )*, *String.slice( )*; «Функция charAt()», глава 4

String.charCodeAt(), метод

Flash 5

возвращает кодовую позицию символа в заданном месте строки

string.charCodeAt(*index*)

Аргументы

index Целочисленная позиция символа в строке, которая должна иметь значение от 0 (первый символ) до *string*.length-1 (последний символ).

Возвращаемое значение

Целое число, представляющее кодовую позицию в Unicode символа в позиции *index* строки *string*. Кодовые позиции первых 256 символов приведены в приложении B.

Пример

```
var msg = "A is the first letter of the Latin alphabet.";
trace(msg.charCodeAt(0));  // Выводит: 65 (код символа "A")
trace(msg.charCodeAt(1));  // Выводит: 32 (код символа "пробел")
```

См. также *String.charAt( )*, *String.fromCharCode( )*; «Функция charCodeAt()», глава 4; приложение B

String.concat(), метод

Flash 5

конкатенирует одно или несколько значений в строку

string.concat(*value1*, *value2*, ...*valuen*)

Аргументы

value1,...*valuen*

Значения, которые должны быть преобразованы в строки (если это необходимо) и конкатенированы с *string*.

Возвращаемое значение

Результат конкатенации *string* с *value1*, *value2*, ...*valuen*.

Описание

Метод *concat( )* создает строку из ряда значений. Он эквивалентен оператору конкатенации строк (+), но иногда его имеет смысл предпочесть из соображений ясности кода,

так как оператор + может применяться и для сложения чисел. Подробнее о *concat( )* см. главу 4.

Применение

Заметьте, что *concat( )* не модифицирует *string*: он возвращает совершенно новую строку.

Пример

```
var greeting = "Hello";
excitedGreeting = greeting.concat("!");
trace(greeting);                    // Выводит: "Hello"
trace(excitedGreeting);             // Выводит: "Hello!"

var x = 4;                          // Инициализировать x как целое
trace(x + 5);                       // Выводит: 9
trace(x.concat(5));                 // Ошибка, так как x не строка.
trace(String(x).concat(5));         // Выводит: "45"

var x = "4";                        // Инициализировать x как строку
trace(x.concat(5));                 // Выводит: "45"
trace(concat("foo", "fee"));        // Ошибка, так как concat() должна
                                    // вызываться как метод строки.
```

См. также

«Соединение строк» и «Функция concat()», глава 4; «Сложение», глава 5

String.fromCharCode(), метод класса

Flash 5

генерирует строку из одной или нескольких кодовых позиций

`String.fromCharCode(code_point1, code_point2,...code_pointn)`

Аргументы

code_point1,...code_pointn

Последовательность из одного или нескольких целых чисел, соответствующих кодовым позициям символов в Unicode.

Возвращаемое значение

Строка, образованная конкатенацией символов, представленных указанными кодовыми позициями.

Описание

Метод класса *fromCharCode( )* создает символ или последовательность символов из кодовых позиций, как описано в главе 4. Если кодовые позиции меньше 128, то строка строится фактически из последовательности кодов ASCII.

Пример

```
// Создает символ ©, за которым следует год (2003)
copyNotice = String.fromCharCode(169) + " 2003";
```

См. также

String.charCodeAt( ); «Функция fromCharCode()», глава 4; приложение B

String.indexOf(), метод

Flash 5

находит первое вхождение подстроки в строку

```
string.indexOf(substring)
string.indexOf(substring, startIndex)
```

Аргументы

substring Строка, содержащая символ или символы, которые требуется найти.

startIndex Необязательное целое, задающее позицию в строке `string`, с которой должен быть начат поиск `substring`. Должно находиться в диапазоне от 0 (первый символ) до `string.length - 1` (последний символ). По умолчанию – 0.

Возвращаемое значение

Позиция первого вхождения `substring` в `string` (начиная со `startIndex`). Возвращает –1, если `substring` не найдена в `string` в позиции `startIndex` или далее.

Описание

Метод *indexOf( )* применяется для поиска в строке символа или проверки того, что строка содержит определенную подстроку. Поиск осуществляется с учетом регистра. Для поиска без учета регистра применяется *toLowerCase( )*, как в третьем примере.

Пример

```
// Проверить, содержит ли адрес электронной почты символ @
var email = "derekaol.com";
if (email.indexOf("@") == -1) {
  trace ("This isn't a valid email address");
}

// Проверить, содержит ли адрес электронной почты символ @ и относится ли он к домену aol.com
var email = "derek@aol.com";
var atPos = email.indexOf("@");
if (atPos != -1 && email.indexOf("aol.com") == atPos + 1) {
  gotoAndStop("AOLuserOffer");
}
```

Ниже приведена общая функция, которая ищет в строке ключевое слово без учета регистра и может понадобиться при оценке ответов в вопроснике с заполнением полей:

```
// Общая функция поиска вхождения в origStr подстроки
// searchStr с использованием нечувствительного к регистру сравнения
function search (origStr, searchStr) {
  var origStr = origStr.toLowerCase();
  var searchStr = searchStr.toLowerCase();
  return origStr.indexOf(searchStr) != -1;
}

var answer = "einstein";
var guess = "Dr. Albert Einstein";

// Увеличить score, если в guess содержится "einstein"
if (search(guess, answer)) {
  score++;
}
```

См. также

String.charAt( ), *String.lastIndexOf( )*, *String.toLowerCase( )*; «Функция indexOf()», глава 4

String.lastIndexOf(), метод

Flash 5

находит последнее вхождение подстроки в строку

```
string.lastIndexOf(substring)
string.lastIndexOf(substring, startIndex)
```

Аргументы

substring Строка, содержащая символ или символы, которые нужно найти.

startIndex Необязательное целое, задающее в `string` позицию, с которой должен быть начат поиск `substring`. Поиск в `string` идет в обратном направлении от `startIndex`, который должен находиться в диапазоне от 0 (первый символ) до `string.length - 1` (последний символ). По умолчанию `string.length - 1`.

Возвращаемое значение

Позиция последнего вхождения `substring` в `string`, предшествующего `startIndex`. Возвращает –1, если `substring` не найдена в `string` в позиции до `startIndex`.

Описание

Метод *lastIndexOf( )* применяется для поиска последнего вхождения подстроки в строку или проверки строки на предмет содержания определенной подстроки. Поиск осуществляется с учетом регистра. Для поиска без учета регистра следует применять *toLowerCase( )*, как в примере для *String.indexOf( )*.

Пример

```
URL = "http://www.moock.org/webdesign/flash/fillthewindow.html";
// Находит последний слэш
lastSlash = URL.lastIndexOf("/");
// Извлекает из URL имя файла
file = URL.substring(lastSlash + 1);
trace(file);  // Выводит: fillthewindow.html
```

См. также

String.charAt( ), *String.indexOf( )*, *String.toLowerCase( )*; «Функция lastIndexOf()», глава 4

String.length, свойство

Flash 5

количество символов в строке **только для чтения**

```
string.length
```

Описание

Свойство `length` возвращает количество символов в `string`. Обратите внимание, что нулевой символ (ASCII 0) не обозначает конец строки, как в некоторых языках (например, C), но и не учитывается в значении `length`. Например:

```
// Создать строку "A" + null + "B"
var myString = String.fromCharCode(65,0,66);
trace(myString.length);  // Выводит: 2 (нулевой символ игнорируется)
```

Пример

```
var myString = "hello";
trace (myString.length);  // Выводит: 5
trace ("hello".length);   // Выводит: 5

// Преобразуем число 1000 в строку и проверим ее длину
var age = 1000;
// Вывести сообщение об ошибке, если в числе неверное количество цифр.
if (String(age).length != 2) {
  trace ("Введите двузначное число");
}
```

См. также *Array.length( )*; «Свойство length», глава 4

String.slice(), метод

Flash 5

извлекает из строки подстроку, определяемую положительными или отрицательными позициями символов

string.slice(*startIndex*, *endIndex*)

Аргументы

startIndex Целое число, задающее позицию первого символа, извлекаемого из *string*. Если *startIndex* отрицателен, позиция отсчитывается от конца строки, причем последний символ занимает позицию –1, предпоследний – позицию –2 и т. д. (Таким образом, отрицательный *startIndex* указывает на символ в позиции *string*.length + *startIndex*.)

endIndex Целое число, задающее позицию символа *за* последним символом, извлекаемым из *string*. Если *endIndex* отрицателен, позиция отсчитывается от конца строки, причем последний символ занимает позицию –1, предпоследний – позицию –2 и т. д. (Таким образом, отрицательный *endIndex* указывает на символ в позиции *string*.length + *endIndex*.) Значением по умолчанию является *string*.length.

Возвращаемое значение

Подстрока *string*, начинающаяся с *startIndex* и заканчивающаяся *endIndex* - 1, где *startIndex* и *endIndex* отсчитываются с нуля.

Описание

Метод *slice( )* – это один из трех методов, применяемых для извлечения подстроки из строки (*substring( )* и *substr( )* – два других). Он позволяет указывать отрицательные значения начального и конечного индексов, что дает возможность извлекать подстроку, отсчитывая от конца строки.

Применение

Заметьте, что *slice( )* не модифицирует *string*, а возвращает совершенно новую строку.

Пример

```
var fullName = "Steven Sid Mumby";
```

```
middleName = fullName.slice(7, 10);   // Присваивает "Sid" переменной middleName
middleName = fullName.slice(-9, -6);  // Тоже присваивает "Sid" переменной middleName
```

См. также

String.substr( ), *String.substring( )*; «Функция slice()» и «Сочетание исследования строки с извлечением подстроки», глава 4

String.split(), метод

Flash 5

преобразует строку в последовательность элементов массива

string.split(*delimiter*)

Аргументы

delimiter Символ или последовательность символов, по которым должна разбиваться строка *string* при образовании элементов нового массива.

Возвращаемое значение

Массив, элементы которого содержат подстроки, образуемые разбиением *string* на участки, ограниченные *delimiter*.

Описание

Метод *split( )* разбивает строку на подстроки, присваивает эти подстроки элементам массива и возвращает этот массив. Смежные положения *delimiter* без промежуточных символов между ними приводят к созданию пустых элементов. Например, следующий код:

```
owners = "terry,doug,,,jon";
ownersArray = owners.split(",");
```

присваивает элементам ownersArray следующие значения (элементы 2 и 3 содержат значения undefined):

```
0: terry
1: doug
2:
3:
4: jon
```

Метод *split( )* обычно применяется для преобразования строки, полученной из сценария CGI или текстового файла, в массив, подлежащий дальнейшей обработке. Он также полезен для разбора параметров в вызове *asfunction* из текстового поля тега HMTL <A>, когда функции можно передать только один строковый аргумент. См. раздел «Вызов функций ActionScript из ссылок HTML» приложения E. Обычно в качестве разделителя выступает запятая или символ табуляции. Применяя в качестве разделителя пробел, можно разбить строку в последовательность слов.

Пример

Допустим, что в текстовом файле *names.txt* хранится список имен. Каждое имя отделяется от других символом табуляции, который предполагается на месте показанных пробелов:

```
owners=terry    doug    jon
```

В кадре 1 фильма загружаем в фильм файл *names.txt*:

```
this.loadVariables("names.txt");
```

Убедившись, что файл *names.txt* полностью загрузился (см. *loadVariables( )*), расщепляем загруженную переменную `owners` в массив:

```
splitString = String.fromCharCode(9);  // Присвоить splitString символ Tab
ownersArray = owners.split(splitString);

trace(ownersArray[1]);  // Выводит: "doug"
```

Учтите, что во Flash 5 при обработке больших участков текста выполнение *split( )* может занять много времени, и что во Flash 6 этот метод выполняется заметно быстрее. Если скорость работы вас не устраивает, попробуйте разбить данные на меньшие фрагменты или воспользоваться XML. См. класс *XML*.

Ошибки реализации

Во Flash 5, если в качестве разделителя выступает пустая строка, вся строка *string* попадает в первый элемент генерируемого массива. Согласно ECMA-262, разделитель в виде пустой строки должен вызывать разбиение *string* на отдельные символы. Аналогично Flash 5 не распознает многосимвольные разделители и присваивает строку *string* целиком первому элементу возвращаемого массива. Во Flash Player 6 эти ошибки исправлены.

См. также *Array.join( )*; «Функция split()», глава 4

String.substr(), метод

Flash 5

извлекает из строки подстроку, заданную начальной позицией и длиной

string.substr(*startIndex*, *length*)

Аргументы

startIndex Целое число, задающее позицию первого символа, извлекаемого из *string*. Если *startIndex* меньше нуля, то позиция отсчитывается от конца строки, причем последний символ занимает позицию –1, предпоследний – позицию –2 и т. д. (Таким образом, отрицательный *startIndex* указывает на символ в позиции *string*.length+ *startIndex*.)

length Количество символов, извлекаемых из *string*, начиная с *startIndex* включительно. Если не указано, извлекаются все символы, начиная с *startIndex* и до конца строки.

Возвращаемое значение

Подстрока *string*, начинающаяся в позиции *startIndex* и содержащая *length* символов. Если *length* опущена, результат содержит символы от *startIndex* и до конца *string*.

Описание

Метод *substr( )* – это один из трех методов, применяемых для извлечения подстроки из строки (*slice( )* и *substring( )* – два других). Метод *substr( )* извлекает строку, исходя из количества символов, заданного аргументом *length*, а не индекса последнего символа.

Применение

Заметьте, что *substr( )* не модифицирует *string*: он возвращает совершенно новую строку.

Пример

```
var fullName = "Steven Sid Mumby";

middleName = fullName.substr(7, 3);       // Присваивает "Sid" переменной middleName
firstName  = fullName.substr(0, 6);       // Присваивает "Steven" переменной firstName
lastName   = fullName.substr(11);         // Присваивает "Mumby" переменной lastName

// Обратите внимание на отрицательные индексы начала...
middleName = fullName.substr(-9, 3);      // Присваивает "Sid" переменной middleName
firstName  = fullName.substr(-16, 6);     // Присваивает "Steven" переменной firstName
lastName   = fullName.substr(-5);         // Присваивает "Mumby" переменной lastName
```

См. также

String.slice( ), *String.substring( )*; «Сочетание исследования строки с извлечением подстроки» и «Функция substr()», глава 4

String.substring(), метод

Flash 5

извлекает из строки подстроку, определяемую положительными позициями символов

string.substring(*startIndex*, *endIndex*)

Аргументы

startIndex Положительное целое число, задающее позицию первого символа, извлекаемого из *string*. Если отрицательно, принимается равным 0.

endIndex Положительное целое число, задающее позицию символа *за* последним символом, извлекаемым из *string*. По умолчанию имеет значение *string*.length. Если отрицательно, принимается равным 0.

Возвращаемое значение

Подстрока *string*, начинающаяся с *startIndex* и заканчивающаяся в *endIndex*-1, где *startIndex* и *endIndex* отсчитываются с нуля.

Описание

Метод *substring( )* – это один из трех методов, применяемых для извлечения подстроки из строки (*slice( )* и *substr( )* – два других). Метод *substring( )* идентичен *slice( )* за исключением того, что не допускает отрицательные значения *startIndex* и *endIndex* и автоматически переупорядочивает эти два индекса, если *endIndex* меньше *startIndex*.

Применение

Заметьте, что *substring( )* не модифицирует *string*, а возвращает совершенно новую строку.

Пример

```
// Извлечь имена из строки
var fullName = "Steven Sid Mumby";
middleName = fullName.substring(7, 10);   // Присваивает "Sid" переменной middleName
middleName = fullName.substring(10, 7);   // Присваивает "Sid" переменной middleName
                                          // (индексы автоматически меняются местами)
firstName  = fullName.substring(0, 6);    // Присваивает "Steven" переменной firstName
lastName   = fullName.substring(11);      // Присваивает "Mumby" переменной lastName
```

В примере C.4 приведена функция, посредством которой можно реализовать поиск и замену всех вхождений подстроки в строку:

Пример С.4. Функция для поиска и замены

```
function replace (origStr, searchStr, replaceStr) {
  var tempStr = "";
  var startIndex = 0;
  if (searchStr == "") {
    return origStr;
  }

  if (origStr.indexOf(searchStr) != -1) {
    while ((searchIndex = origStr.indexOf(searchStr, startIndex)) != -1) {
      tempStr += origStr.substring(startIndex, searchIndex);
      tempStr += replaceStr;
      startIndex = searchIndex + searchStr.length;
    }
    return tempStr + origStr.substring(startIndex);
  } else {
    return origStr;
  }
}

msg = "three times three is four";
trace(replace(msg, "three", "two"));  // Выводит: "two times two is four"
```

См. также

String.slice( ), *String.substr( )*; «Сочетание исследования строки с извлечением подстроки» и «Функция substr()», глава 4

String.toLowerCase(), метод

Flash 5

создает версию строки, записанную буквами нижнего регистра

string.toLowerCase()

Возвращаемое значение

Новая строка как эквивалент *string* в нижнем регистре. Символы, не имеющие такого эквивалента, остаются без изменения.

Описание

Метод *toLowerCase( )* создает новую версию *string* в нижнем регистре, его можно использовать при форматировании или нечувствительном к регистру сравнении символов.

Применение

Заметьте, что *toLowerCase( )* не модифицирует *string*, а возвращает совершенно новую строку.

Пример

```
// Присвоить msg значение "this sentence has mixed caps!"
msg = "ThiS SenTencE Has MixED CaPs!".toLowerCase();

// Выполнить нечувствительное к регистру сравнение двух строк
function caseInsensitiveCompare (stringA, stringB) {
  return (stringA.toLowerCase() == stringB.toLowerCase());
}

trace(caseInsensitiveCompare("Colin", "colin"));  // Выводит: true
```

См. также *String.toUpperCase( )*; «Функция toLowerCase()», глава 4

String.toUpperCase(), метод

Flash 5

создает версию строки, записанную буквами верхнего регистра

```
string.toUpperCase()
```

Возвращаемое значение

Новая строка как эквивалент *string* на верхнем регистре (называемый также ALL CAPS). Символы, не имеющие эквивалента в верхнем регистре, остаются без изменения.

Описание

Метод *toUpperCase( )* создает новую версию *string* на верхнем регистре, его можно использовать при форматировании или нечувствительном к регистру сравнении символов.

Применение

Заметьте, что *toUpperCase( )* не модифицирует *string*, а возвращает совершенно новую строку.

Пример

```
"listen to me".toUpperCase();   // Дает строку "LISTEN TO ME"
var msg1 = "Your Final Score: 234";
var msg2 = msg1.toUpperCase();  // Присвоить msg2 значение "YOUR FINAL SCORE: 234"
```

См. также *String.toLowerCase( )*; «Функция toUpperCase()», глава 4

super, «оператор»

Flash 6

вызывает конструктор надкласса объекта или реализованный в надклассе замещенный метод

```
super(arg1, arg2,...argn);
super.methodName(arg1, arg2,...argn);
```

Операнды

arg1,...argn

Список аргументов, передаваемых функции-конструктору надкласса.

methodName

Имя реализованного в надклассе замещенного метода, который должен быть вызван.

Описание

«Оператор» *super*, появившийся во Flash 6, выполняет две задачи объектно-ориентированного программирования:

- Вызывает конструктор надкласса объекта.
- Вызывает реализацию в надклассе замещенного метода.

Значительно больше информации о нем содержится в разделе «'Оператор' super» главы 5.

Применение

Формально *super* не является настоящим оператором ActionScript, поэтому мы заключаем слово «оператор» в кавычки.

См. также

instanceof, *new*, *Object.hasOwnProperty( )*; «„Оператор“ super», глава 5; «Создание надкласса», «Доступ к замещенным методам» и «Пользовательская функция setSuperClass()», глава 12.

System, объект

Flash 6

доступ к настройкам и характеристикам проигрывателя и системы

Свойства

capabilities Ссылка на объект *Capabilities*.

security Объект для задания разрешений для фильмов из других доменов.

useCodepage Заставить Flash Player использовать кодовую страницу операционной системы, а не Unicode.

Методы

showSettings( ) Показать диалоговое окно настройки Flash Player.

Описание

Объект *System* предоставляет информацию о характеристиках Flash Player и системы, в которой он работает, и возможность управления ими. За исключением *showSettings( )* (метода, вызывающего диалоговое окно настроек Flash Player) и `useCodepage` (свойства, используемого для управления режимом кодировки символов), доступ к характеристикам Player и системы осуществляется через объекты *Capabilities* и *Security*, которые сами хранятся в виде свойств *System*. Объект *Capabilities* в настоящем «Справочнике по языку» документирован отдельно как *Capabilities*, *объект*. Единственный метод объекта *Security*, *allowDomain( )*, который может разрешать фильмам, загруженным из разных доменов, получать доступ друг к другу, документирован как *System.security.allowDomain( )*.

Обратите внимание, что *System* – это готовый объект, а не класс. Доступ к его свойствам и методам осуществляется непосредственно:

```
System.showSettings(0);  // Выводит вкладку Privacy окна диалога Settings
```

См. также

Объект *Capabilities*, объект *Stage*

System.capabilities, свойство

Flash 6

ссылка на объект Capabilities

```
System.capabilities
```

Описание

Свойство `capabilities` представляет собой объектную ссылку на встроенный объект *Capabilities*. Полное описание можно найти в статье «Capabilities» настоящего «Справочника по языку». Запись в `System.capabilities` технически возможна, но изменять его в своих фильмах не следует.

См. также Объект *Capabilities*

System.security.allowDomain(), метод Flash 6

устанавливает права фильмам, загруженным из разных доменов

```
System.security.allowDomain(domain1, domain2,...domainn)
```

Аргументы

domain1,...domainn

Список из одного или нескольких доменов, которым должен быть предоставлен доступ к фильму.

Описание

Свойство `security` представляет собой ссылку на объект *Security*, в котором определен один важный метод, *allowDomain( )*. С помощью *allowDomain( )* мы предоставляем фильмам из других доменов доступ к текущему фильму. Хотя ограничения, связанные с защитой, введены во Flash Player начиная с версии 4, Flash 6 формализует понятие «песочницы» (*security sandbox*). Согласно правилам песочницы, фильму, находящемуся в некотором домене, не разрешается:

- Вызывать методы и функции в фильме, загруженном из другого домена.
- Обращаться к данным (переменным и свойствам, определенным пользователем, и вложенным клипам) в фильме, загруженном из другого домена.
- Загружать XML или переменные из другого домена с помощью класса *LoadVars*, *loadVariables( )*, *XML.load( )* или *XML.sendAndLoad( )*.
- Соединяться с сервером *XMLSocket* другого домена.

По умолчанию песочница активна, и все попытки выполнения перечисленных операций игнорируются Flash Player 6. Эти ограничения отменяются посредством метода *allowDomain( )*.

Этот метод вызывается в фильме, доступ к которому надо предоставить конкретному домену или группе доменов, заданных в *domain1*, *domain2*, ...*domainn*. Допустим, например, что Graham размещает свое резюме *grayResume.swf* на *graydient.com* и хочет разрешить доступ к нему для *jobBoard.swf*, размещенного на *moock.org*. В первом кадре *grayResume.swf* он должен вызвать *allowDomain( )*, чтобы предоставить права фильмам на *moock.org*:

```
System.security.allowDomain("moock.org");
```

После добавления *moock.org* к списку доверенных доменов для *grayResume.swf*, *jobBoard.swf* (на *moock.org*) может загрузить *grayResume.swf* с *graydient.com* и присваивать значения его переменным, останавливать головку воспроизведения и выполнять прочие действия без каких-либо ограничений со стороны системы защиты. Например:

```
// Создать контейнер для загрузки grayResume.swf
this.createEmptyMovieClip("holder", 1);
// Загрузить grayResume.swf
holder.loadMovie("http://www.graydient.com/grayResume.swf");

// Создать загрузчик, обнаруживающий окончание загрузки фильма
var preloadInterval = setInterval(setText, 1000);
```

```
// Функция, вызываемая раз в секунду. После загрузки эта функция
// записывает в фильм некоторый текст и останавливает воспроизведение.
function setText () {
  if (holder._framesloaded > 0 && holder._framesloaded == holder._totalframes) {
    // Остановить загруженный фильм
    holder.stop();
    // Записать некоторый текст
    holder.subtitle_txt.text = "Gray's resume brought to you by moock.org.";
    // Остановить проверку конца загрузки
    clearInterval(preloadInterval);
  }
}
```

На фильмы, располагающиеся в одном и том же домене, ограничения песочницы не действуют. В табл. С.21 показано, в каких случаях действует песочница защиты. Чтобы узнать домен текущего фильма, следует применять *LocalConnection.domain( )*.

Таблица С.21. Ограничения защиты, основанные на доменах

Домен – источник фильма	Хост, к которому надо подключиться	Разрешено?	Разрешено с помощью allowDomain()?
www.somewhere.com	*www.somewhere. com*	Да	Да
www.somewhere.com	*other.somewhere. com*	Да	Да
www.somewhere.com	*somewhere.com*	Да	Да
somewhere.com	*www.somewhere. com*	Да	Да
www.somewhere.com	*www.elsewhere.com*	Нет	Да

Для обеспечения обратной совместимости фильмы Flash 4 и Flash 5, загруженные во Flash Player 6 из разных доменов, могут все же обращаться к данным друг друга (как допускалось в старых версиях Player). Однако фильм Flash 5 не может получить доступ к данным фильма Flash 6, загруженного из другого домена, и наоборот.

Метод *allowDomain( )* не может непосредственно изменить ограничения защиты при загрузке XML, загрузке переменных или связи через *XMLSocket*. Однако можно загружать XML или переменные из другого домена, используя фильм этого домена в качестве агента связи (proxy). Допустим, например, что *main.swf* на *moock.org* желает загрузить *newsFeed.xml* с *graydient.com*. Файл *main.swf* должен загрузить proxy-фильм *xmlloader.swf* с *graydient.com*. Затем *xmlloader.swf* должен загрузить файл *newsFeed.xml* и предоставить доступ к нему *moock.org* с помощью *allowDomain( )*. После этого фильм *main.swf* может обращаться к загруженным данным через *xmlloader.swf*. К сожалению, на момент написания книги существовала ошибка, не позволявшая этой системе proxy-фильмов функционировать в текущей версии Flash Player 6 (6.0.61.0). Macromedia известно об этой проблеме, и она работает над ней.

Ограничения для XML и загрузки переменных можно также обойти с помощью псевдонима DNS или сценария на сервере, выступающего посредником между Flash и внешним сайтом. Дополнительная информация имеется по адресу:

http://www.macromedia.com/support/flash/ts/documents/loadvars_security.htm
http://www.macromedia.com/support/flash/ts/documents/load_xdomain.htm

Клиент Flash в одном домене может соединиться с сервером *XMLSocket* в другом домене с помощью фильма в домене сервера, действующего как посредник при связи (от-

меченная выше ошибка не относится к связи через *XMLSocket*). Допустим, что сервер сокетов чата выполняется на *moock.org*. Владелец *graydient.com* желает разместить на *graydient.com* фильм чата (*chat.swf*), который использует сервер чата *moock.org*.

Владелец *moock.org* должен создать фильм-посредник для *graydient.com*, выполнив следующие действия:

1. Создать фильм с именем *socketObject.swf*.
2. В кадр 1 фильма *socketObject.swf* поместить такой код:

```
// Предоставить доступ к graydient.com
System.security.allowDomain("graydient.com");
// Создать объект XMLSocket. Это необходимо сделать здесь.
sock = new XMLSocket();
```

3. Разместить *socketObject.swf* на *moock.org* (в домене сервера сокетов).

В *chat.swf* владелец *graydient.com* должен воспользоваться фильмом-посредником *socketObject.swf* для связи в сервером сокетов *moock.org*, для чего требуется:

1. В кадре 1 фильма *chat.swf* загрузить *socketObject.swf* на уровень 1 (годится любой уровень или приемник):

```
loadMovieNum("http://www.moock.org/socketObject.swf", 1);
```

2. После загрузки *socketObject.swf* с помощью объекта `_level1.sock` подключиться и держать связь с сервером сокетов. Например:

```
// Создать текстовое поле для вывода
this.createTextField("out_txt", 1, 100, 100, 400, 100);
out_txt.border = true;

// Задать функцию обратного вызова для обработки соединения
_level1.sock.onConnect = function (success) {
  out_txt.text += ("\n Connected: " + success);
}
// Задать функцию обратного вызова для обработки загружаемого XML
_level1.sock.onXML = function (doc) {
  out_txt.text += ("\n XML arrived: " + doc);
}
// Подключиться к серверу
_level1.sock.connect("moock.org", 2001);

// Код кнопки, которая отправляет XML серверу сокетов
sendMessage_btn.onPress = function () {
  _level1.sock.send("<MESSAGE>hello world</MESSAGE>");
}
```

На фильм, взятый из локальной файловой системы и запущенный в автономном проигрывателе Flash или в броузере (то есть через подключаемый модуль), не распространяется действие песочницы. Он может загружать XML или переменные из любого домена и работать с данными любого фильма, который загрузит. Обратное неверно. Например, если локальный фильм загрузит другой фильм, *booklist.swf*, с *oreilly.com*, то *booklist.swf* не сможет получить доступ или управление для локального фильма, если в локальном фильме не будет следующего вызова *allowDomain( )*:

```
System.security.allowDomain("oreilly.com");
```

Общее практическое правило состоит в том, что «локальное может управлять загружаемым, но загружаемое не может управлять локальным».

Обратите внимание, что встроенные свойства, такие как `_framesloaded` и `_x`, не подвергаются ограничениям песочницы. Например, фильм, загрузивший другой фильм, может изменять расположение загруженного фильма независимо от домена его происхождения.

См. также

loadMovie( ), *loadVariables( )*, класс *LoadVars*, *XML.load( )*, *XML.sendAndLoad( )*, класс *XMLSocket*

System.showSettings(), метод

Flash 6 (не документирован)

открывает окно диалога настроек Flash Player

```
System.showSettings(tabID)
```

Аргументы

tabID

Целое число, указывающее конкретную вкладку, которую нужно показать.

Описание

Метод *showSettings( )* выводит окно диалога настроек Macromedia Flash Player, в котором пользователь может задать параметры воспроизведения фильма в проигрывателе. Во Flash Player 6 окно Settings содержит четыре вкладки, каждой из которых соответствует целочисленный идентификатор `tabID`: Privacy (0), Local Storage (1), Microphone (2) и Camera (3).

Обычно пользователь открывает окно настроек Macromedia Flash Player вручную, щелкнув правой кнопкой по фильму в Windows (или выполнив Ctrl-щелчок на Макинтоше) и выбрав «Settings...» в контекстном меню. Метод *showSettings( )* дает возможность помочь пользователю, управляя из программы отображением окон настроек, когда пользователю предлагается изменить параметр, требующийся для фильма. Например, если фильму требуется более 100 Кбайт локального дискового пространства, можно попросить пользователя увеличить границу и выполнить следующую команду, которая откроет окно Settings на той вкладке, где пользователь может задать граничное значение для локального дискового пространства:

```
System.showSettings(1);  // Показать вкладку настроек Local Storage
```

См. также

SharedObject.flush( ), *SharedObject.getSize( )*, `Stage.showMenu`

System.useCodepage, свойство

Flash 6

заставить Flash Player использовать кодовую страницу операционной системы вместо Unicode

```
System.useCodepage
```

Описание

Unicode – это международный набор символов, включающий в себя все символы всех мировых языков. Большинство современных приложений, включая Flash Player 6, используют Unicode для обработки и показа текста. Старые приложения, в том числе Flash Player 5, работают с текстом, используя так называемую *кодовую страницу* операционной системы (в сущности, набор местных символов).

От свойства `System.useCodepage` зависят:

- Отображение текстовых полей
- Операции *load( )*, *XML.send( )* и *XML.sendAndLoad( )*
- Операции *LoadVars.load( )*, *LoadVars.send( )* и *LoadVars.sendAndLoad( )*
- Действия *getURL( )*, *loadMovie( )* и *loadVariables( )*, при которых передаются/принимаются переменные
- Связь через *XMLSocket*
- Функции *escape( )* и *unescape( )*

Когда приложение, не поддерживающее Unicode, выводит символ на экран, оно ищет этот символ по его кодовой позиции в текущей кодовой странице системы. На экране отображается тот символ, который указан на кодовой странице во время показа. Если кодовая страница ввода (та, которая действовала при вводе символа) не совпадает с кодовой страницей отображения (той, которая действует, когда приложение показывает символ), введенный символ может не совпасть с отображаемым. Для решения этой проблемы был введен стандарт Unicode. Поскольку Flash 6 поддерживает Unicode, необходимость в обращении к свойству `System.useCodepage` не возникнет, если:

- Использовать Unicode (в кодировке UTF-8 или UTF-16) для всего внешнего текста
- Использовать escape-последовательности Unicode для всех символов, которых нельзя ввести с клавиатуры в среде разработки Flash

С помощью булева свойства `useCodepage` можно заставить Flash Player 6 использовать при выводе текста текущую кодовую страницу операционной системы вместо Unicode:

```
// Принудительное использование кодовой страницы операционной системы
System.useCodepage = true;
```

Свойство `System.useCodepage` может быть изменено в любой момент, например в начале и в конце операции, которая требует старой кодировки текста, такой как загрузка текстового файла, в котором использована особая кодировка.

При экспорте файлов *.swf* в формате Flash 6 из среды разработки Flash MX необходимость в применении свойства `useCodepage` возникает редко (при экспорте Flash MX автоматически преобразует в Unicode текст, введенный в среде разработки). Однако присваивание свойству `useCodepage` значения `true` не приводит к неправильному отображению текста в кодировке, отличной от Unicode, если одновременно выполняются следующие два условия:

- Текст в кодировке, отличной от Unicode, загружен во Flash Player из внешнего текстового файла или сценарием сервера.
- При кодировке внешнего текста использовалась та же самая кодовая страница, которая установлена в данный момент в операционной системе, в которой работает Flash Player.

Текст не будет показан правильно, если текущая кодовая страница операционной системы не соответствует кодовой странице при вводе! Хотя принудительное задание кодовой страницы системы может волшебным образом решить проблемы отображения на отдельном компьютере, присваивание `System.useCodepage` значения `true` редко помогает при выводе интернациональных текстов. Поэтому по возможности все внешние файлы должны использовать Unicode, а для `useCodepage` следует сохранить значение по умолчанию `false`.

По умолчанию на Западе в большинстве компьютеров под Windows установлена кодовая страница CP1252 (WinLatin1). Отображение из CP1252 в Unicode можно найти по адресу:

http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WINDOWS/CP1252.TXT

По умолчанию в Японии в большинстве компьютеров под Windows используется кодовая страница CP932 (Windows Shift-JIS). Отображение из CP932 в Unicode можно найти по адресу:

http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WINDOWS/CP932.TXT

Полный список кодовых страниц Windows есть по адресу:

http://www.microsoft.com/globaldev/reference/WinCP.asp

Частичный список кодовых страниц, используемых другими поставщиками, есть по адресу:

http://czyborra.com/charsets/codepages.html

Дополнительные сведения о поддержке Unicode во Flash см. в разделе «Кодировка символов» главы 4.

См. также «Тип String» в главе 4

targetPath(), глобальная функция

Flash 5

возвращает абсолютный путь к фильму или клипу

`targetPath (movieClip)`

Аргументы

movieClip Ссылка на объект клипа.

Возвращаемое значение

Строка, представляющая абсолютный путь к `movieClip` в точечной нотации (например, `"_level0.myMovie"`).

Описание

Функция *targetPath( )* возвращает ссылку на клип в виде строки, описывающей абсолютный путь к клипу (идентичной значению, возвращаемому *MovieClip.valueOf( )*). Функция *targetPath( )* иногда применяется для написания чувствительного к размещению кода, обрабатывающего клип в соответствии с тем, на какой временной диаграмме он находится. Для получения пути к клипу в слэш-нотации вместо точечной нотации применяется `MovieClip._target`.

Пример

Если `square` представляет собой клип, содержащийся в `shapes`, находящемся на главной временной диаграмме уровня `level0`, то внутри клипа `shapes` команда

```
targetPath(square);
```

возвратит:

```
"_level0.shapes.square"
```

См. также

*MovieClip.*_target, *MovieClip.valueOf( )*; «Функция targetPath()» в главе 13

tellTarget(), глобальная функция

Flash 3; устарела во Flash 5

выполнить инструкции, находящиеся в области видимости удаленного клипа

```
tellTarget (target) {
  инструкции
}
```

Аргументы

target Строка или ссылка, указывающая путь к фильму или экземпляру клипа (в строковом контексте ссылки преобразуются в пути).

statements Инструкции, которые должны быть выполнены в области видимости *target*.

Описание

Во Flash 3 и Flash 4 *tellTarget( )* была главным средством связи между двумя клипами (то есть управления одним клипом из другого). Она применялась для вызова таких функций, как *play( )*, *stop( )* и *gotoAndStop( )*, с удаленными клипами. Во Flash 4, где в ActionScript были добавлены переменные, с помощью *tellTarget( )* можно было также получать и устанавливать значения переменных удаленных клипов. Во Flash 5 такие действия лучше осуществлять с помощью оператора «точка» и оператора обращения к массиву, []. Другой альтернативой функции *tellTarget( )* является инструкция *with*, описанная в главе 6.

Применение

Функцию *tellTarget( )* лучше было бы описать как инструкцию, поскольку для нее требуется блок вложенных инструкций. Вопрос, однако, академический, поскольку *tellTarget( )* устарела.

Пример

```
// Переместить клип ball на 300 пикселов вправо и отправить
// его головку воспроизведения в кадр с меткой "redStripes".
tellTarget ("ball") {
  gotoAndStop("redStripes");
  _x += 300;
}
```

См. также

«Инструкция with» в главе 6, «'Объектность' клипов» в главе 13; «Использовать ли Tell Target?» в приложении C.

TextField, класс

Flash 6

отображение и обработка текста на экране

Конструктор

Отсутствует. Объекты *TextField* создаются вручную в среде разработки или программно методом *MovieClip.createTextField( )*.

Свойства

_alpha Степень непрозрачности: 0% – прозрачность; 100% – непрозрачность.

autoSize Подгонка размеров ограничивающего прямоугольника к размерам текста.

background Булево значение; включает и выключает отображение фона текстового поля.

backgroundColor

RGB-значение цвета фона текстового поля.

border Булево значение; включает и выключает показ рамки текстового поля.

borderColor RGB-значение цвета рамки текстового поля.

bottomScroll Индекс самой нижней видимой строки текстового поля, отсчитываемый с 1.

condenseWhite Булево значение; включает и выключает сжатие пробельных символов в тексте HTML.

embedFonts Булево значение; определяет использование экспортированных с фильмом шрифтов при выводе текста.

_height Высота рамки (в пикселах), ограничивающей текстовое поле.

hscroll Прокрутка текстового поля (в пикселах) по горизонтали.

html Булево значение; включает и выключает вывод HTML.

htmlText Исходный код HTML, который должен быть выведен в текстовое поле.

length Количество символов в текстовом поле.

maxChars Ограничивает максимальный объем вводимых пользователем данных.

maxhscroll Максимальное расстояние, на которое можно прокрутить текст влево.

maxscroll Последняя допустимая верхняя строка текстового поля.

multiline Булево значение; включает и выключает многострочный ввод.

_name Имя экземпляра объекта *TextField*.

_parent Ссылка на клип, в котором располагается текстовое поле.

password Булево значение; маскирует символы, отображаемые в поле.

restrict Ограничивает ввод текста группой символов.

_rotation Поворот текстового поля в градусах.

scroll Текущая верхняя строка, отображаемая в текстовом поле.

selectable Булево значение; включает и выключает возможность выделения пользователем.

tabEnabled Булево значение; включает и исключает текстовое поле из текущего порядка табуляции.

tabIndex Задает индекс текстового поля в пользовательском порядке табуляции.

_target Путь к текстовому полю в слэш-синтаксисе.

text Задает символы, которые должны быть показаны в поле.

textColor Задает цвет всех символов текстового поля.

textHeight Высота всего текста текстового поля, в пикселах.

textWidth Ширина самой длинной строки текстового поля, в пикселах.

type Определяет, может ли поле принимать данные от пользователя.

_url Сетевой адрес клипа, который содержит поле.

variable Связывает переменную со свойством text.

_visible Булево значение; включает и выключает отображение текстового поля.

_width Ширина ограничивающего прямоугольника текстового поля, в пикселах.

wordWrap Булево значение; включает и выключает автоматический перенос длинных строк.

_x Расположение текстового поля по горизонтали, в пикселах.

_xmouse Расположение курсора мыши по горизонтали относительно текстового поля.

_xscale Ширина текстового поля в процентах к оригиналу.

_y Расположение текстового поля по вертикали, в пикселах.

_ymouse Расположение курсора мыши по вертикали относительно текстового поля.

_yscale Высота текстового поля в процентах к оригиналу.

Методы класса

getFontList( ) Возвращает массив шрифтов, установленных в системе пользователя.

Методы

addListener( ) Регистрирует объект для получения извещений о событиях *onChanged( )* и *onScroller( )*.

getDepth( ) Возвращает позицию текстового поля в стеке фильма.

getNewTextFormat( )

Возвращает объект *TextFormat* по умолчанию для текстового поля.

getTextFormat( )

Возвращает объект *TextFormat* для заданных символов.

removeListener( )

Прекращает получение уведомлений о событиях указанным приемником.

removeTextField( )

Удаляет созданное на этапе исполнения текстовое поле.

replaceSel( ) Замещает выделенный текст (не меняя форматирования).

setNewTextFormat( )

Задает форматирование по умолчанию для текстового поля.

setTextFormat( )

Задает форматирование указанных символов.

Обработчики событий

onChanged( ) Функция обратного вызова, вызываемая, когда обнаруживается ввод данных пользователем.

onKillFocus( ) Функция обратного вызова, вызываемая при потере полем фокуса.

onScroller( ) Функция обратного вызова, вызываемая, когда изменяется свойство прокрутки.

onSetFocus( ) Функция обратного вызова, вызываемая, когда поле получает фокус.

События приемников

onChanged( ) Происходит, когда обнаруживается ввод данных пользователем.

onScroller( ) Происходит, когда изменяется свойство прокрутки.

Описание

Класс *TextField* позволяет управлять отображением текста на экране. Каждый объект *TextField* представляет собой прямоугольный контейнер, который может получать текст, изменять размер или формат и даже получать данные, веденные пользователем. Когда мы говорим «текстовое поле», то имеем в виду прямоугольный контейнер на экране; когда мы говорим об объекте или экземпляре *TextField*, то имеем в виду объект ActionScript, управляющий текстовым полем.

Объекты *TextField* не строятся с помощью оператора *new*, они автоматически создаются Flash, когда изображается текст на этапе разработки (с помощью инструмента Text) или при вызове метода *MovieClip.createTextField( )*. Например, следующий код создает текстовое поле с именем `theField_txt` и помещает его на глубину 1 на сцене с координатами (0,0), с шириной 200 и высотой 20. Ключевое слово `this` указывает на текущий клип. Суффикс «_txt» активизирует подсказку кода в окне Actions:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
```

Чтобы получить доступ к экземпляру *TextField* для поля, созданного на этапе разработки, надо установить для него тип Dynamic Text или Input Text и присвоить экземпляру имя с помощью инспектора свойств.

Следующие шаги описывают типичный порядок действий при создании управляемого через ActionScript текстового поля на этапе разработки:

1. Выбрать инструмент Text.
2. Нарисовать прямоугольник, достаточно большой для отображения на экране нужного текста.
3. Открыть инспектор свойств (Window → Properties).
4. В инспекторе свойств выбрать в качестве Type значение Dynamic Text или Input Text (тип по умолчанию Static Text не может обрабатываться с помощью ActionScript).

5. В инспекторе свойств в поле Instance Name ввести `someName_txt`. Имя экземпляра нельзя задать, пока не установлен тип поля, как на шаге 4.

О том, как удалить текстовое поле на этапе исполнения, рассказано в описании метода *TextField.removeTextField( )*.

Простой текст, содержащийся в текстовом поле, можно получить с помощью свойства `text` и задать с помощью метода *replaceSel( )* или свойства `text`. Текст HTML можно задать с помощью свойства `htmlText`, при условии, что свойству `html` тоже присвоено значение.

До Flash 6 идентификатор текстового поля в ActionScript (известный как его «переменная») служил для обозначения содержимого поля, но во Flash 6 идентификатор текстового поля является именем экземпляра объекта. Чтение и запись содержимого текстового поля во Flash 6 осуществляется через свойство *TextField*.`text`.

Во Flash 5 синтаксис, позволяющий поместить в текстовое поле `highScore_txt` значение `"15, 000"`, был таким:

```
highScore_txt = "15, 000";
```

Но во Flash 6 текст, выводимый в текстовом поле, задается свойством `text`:

```
highScore_txt.text = "15, 000";
```

Окно Var: в инспекторе свойств для текстового поля сохранено во Flash MX для обеспечения обратной совместимости, как и свойство *TextField*.`variable`. При разработке фильмов для Flash Player 6 присваивайте всем текстовым полям имя экземпляра, а доступ к их содержимому организуйте при помощи свойства `text`.

По умолчанию пользователь может выделять содержимое текстовых полей типа Dynamic и Input. Для того чтобы создать поле, содержимое которого выделить нельзя, следует задать его свойство `selectable` равным `false`:

```
output_txt.selectable = false;
```

Когда пользователь выделяет часть динамического или предназначенного для ввода поля, индексы выделенных символов хранятся в объекте *Selection*. С помощью объекта *Selection* можно узнать, какую часть текстового поля выделил пользователь, можно даже программно выделить часть текстового поля. Объект *Selection* может также переносить фокус ввода с клавиатуры в конкретное текстовое поле или сообщать, какое текстовое поле выбрано пользователем в данный момент. См. *Selection.onSetFocus( )*, *TextField. onKillFocus( )* и *TextField.onSetFocus( )*.

Для текстовых полей этапа разработки многие свойства объекта *TextField* можно задать через интерфейс инспектора свойств. Параметры, заданные на этапе разработки, на этапе исполнения отражаются в соответствующих свойствах экземпляра *TextField*.

Реагировать на изменения в прокрутке текстового поля или его содержимом можно с помощью событий *onChanged( )* и *onScroller( )* объекта *TextField*.

Для форматирования текста в текстовом поле применяется объект *TextFormat* или вывод в формате HTML. Подробности см. в следующих статьях для *TextField*: `embedFonts`, *getFontList( )*, *getTextFormat( )*, *getNewTextFormat( )*, `html`, `htmlText`, *setFontList( )*, *setTextFormat( )*, *setNewTextFormat( )* и *textColor( )*, а также в статье, посвященной классу *TextFormat*. Можно также задать цвет прямоугольной рамки и фона текстового поля; см. статьи для *TextField*: `border`, `borderColor`, `background` и `backgroundColor`.

До Flash 6 текстовые поля формально не были доступны как объекты, а начиная с Flash 6 класс *TextField* позволяет работать с текстовыми полями так, как это делается с клипами, устанавливая их положение, размер, поворот и т. д. На самом деле многие свойства текстовых полей непосредственно заимствованы у класса *MovieClip* (например, `_x`, `_y`, `_width`, `_height` и `_rotation`). С целью обеспечения обратной совместимости свойства текстовых полей, соответствующие свойствам клипов, существовавшим до Flash 6, снабжаются префиксом в виде символа подчеркивания (_). Для более тесной согласованности с другими языками ECMA-262 новые свойства, появившиеся только во Flash 6, не используют соглашения с символом подчеркивания.

Ошибки реализации

Во Flash 6 встроенные свойства, задаваемые в *TextField*.`prototype`, не наследуются экземплярами текстовых полей, поэтому нельзя присвоить всем текстовым полям в фильме пользовательские значения по умолчанию:

```
// Эта попытка дать всем текстовым полям границу по умолчанию
// во Flash 6 будет неудачной
TextField.prototype.border = true;
```

См. также *MovieClip.createTextField( )*, класс *TextFormat*; глава 17

TextField.addListener(), метод

Flash 6

регистрирует объект для получения событий onChanged() и onScroller()

theField.`addListener(`*listener*`)`

Аргументы

listener Объект, в котором определены методы *onChanged( )* и/или *onScroller( )*.

Возвращаемое значение

Булево значение, указывающее, успешным ли было добавление приемника. На практике во Flash Player 6 *addListener( )* всегда возвращает `true`. Однако в следующих версиях, возможно, будет возвращать `false`, если не хватит памяти для добавления приемника.

Описание

Метод *addListener( )* добавляет *listener* в список объектов, которые уведомляются при возникновении событий *onScroller( )* или *onChanged( )* объекта *theField*. Аргумент *listener* представляет собой объект, в котором определены методы с именами *onScroller* и *onChanged*, вызываемые, когда происходят соответствующие события для `theField`. Прекратить получение событий объектом приемника позволяет метод *TextField.removeListener( )*.

События приемника позволяют одному *TextField* запускать функции в нескольких объектах. Если в ответ на событие *TextField* должна запускаться только одна функция, то достаточно функции обратного вызова, синтаксис которой проще. Обратите внимание, что у *TextField* для одного и того же события могут быть функции обратного вызова и приемники. Функции обратного вызова выполняются раньше, чем методы приемников. Если в *TextField* зарегистрировано несколько приемников, их методы выполняются в том порядке, в котором добавлялись приемники.

Общие сведения о действии приемников и функций обратного вызова см. в главе 10.

Пример

Следующий код создает текстовое поле с именем demo_txt и назначает для него объект приемника txtListener, который отвечает на событие *onChanged( )*. Для сравнения показан и равносильный подход с применением функции обратного вызова. Чтобы увидеть этот код в действии, скопируйте его в новый фильм, экспортируйте этот фильм в режим Test Movie и введите текст в текстовое поле.

```
// Создать текстовое поле.
this.createTextField("demo_txt", 1, 10, 10, 200, 20);
// Вывести в поле какой-нибудь текст.
demo_txt.text = "type text here";
// Разрешить редактирование поля.
demo_txt.type = "input";
// Создать объект приемника.
txtListener = new Object();
// Создать в приемнике функцию onChanged. Этой функции
// передается имя поля, сгенерировавшего событие.
txtListener.onChanged = function (tfObject) {
  trace(tfObject + " changed. Listener invoked.");
}
// Добавить приемник к demo_txt.
demo_txt.addListener(txtListener);
// Сравните, как то же самое выполняется функцией обратного вызова.
demo_txt.onChanged = function () {
  trace(this + " changed. Callback invoked.");
}
```

См. также

TextField.onChanged( ), *TextField.onScroller( )*, *TextField.removeListener( )*; глава 10

TextField._alpha, свойство

Flash 6

степень непрозрачности: 0% – прозрачно; 100% – непрозрачно **чтение/запись**

theField._alpha

Описание

Свойство _alpha, выражаемое числом с плавающей точкой, задает степень непрозрачности (или, наоборот, прозрачности) *theField* в процентах: 0 соответствует полной прозрачности, а 100 – полной непрозрачности. Оно действует на текст, фон и рамку *theField*. Свойство _alpha действует только тогда, когда шрифт *theField* встроен и экспортирован с фильмом. Шрифт можно сделать встроенным на этапе разработки с помощью кнопки Character в инспекторе свойств либо на этапе исполнения с помощью свойства embedFonts. Встраивание шрифтов подробно рассмотрено в *TextField*.embedFonts.

Пример

Предположим, что шрифт экспортирован с фильмом, тогда следующий код делает theField_txt прозрачным на 50%:

```
theField_txt.embedFonts = true;
theField_txt._alpha = 50;
```

См. также

TextField._visible, *TextField*.embedFonts

TextField.autoSize, свойство[1]

Flash 6

подгонка размеров ограничивающего текстовое поле прямоугольника к размерам текста **чтение/запись**

theField.autoSize

Описание

Строковое свойство autoSize позволяет добиться соответствия ограничивающего прямоугольника текстового поля размеру текста, который в нем содержится. Оно применяется для того, чтобы гарантировать размер текстового поля, достаточный для отображения его текста, или чтобы динамически изменять размер текстового поля, когда меняется его содержимое (при вводе данных пользователем или присваивании значения свойству text). Например, создадим текстовое поле title_txt с текстом «Contact Information». Размер текста нам неизвестен, поэтому мы не можем точно задать ширину title_txt:

```
// Ширина и высота 80 и 20 взяты наугад...
this.createTextField("title_txt", 1, 0, 0, 80, 20);
title_txt.text = "Contact Information";
title_txt.border = true;
```

Задав для autoSize значение "left", можно гарантировать, что слова «Contact Information» будут видны:

```
title_txt.autoSize = "left";
```

У свойства autoSize четыре возможных значения: "none" (по умолчанию), "left", "right" и "center". Они определяют направление, в котором *theField* должно расширяться или сжиматься, чтобы соответствовать тексту: "none" означает, что размер не меняется, "left" — расширение вправо-вниз, "right" — расширение влево-вниз и "center" — равномерное расширение в обе стороны и вниз. Если autoSize присваивается булево значение, то false интерпретируется как синоним "none", а true — как синоним "left". У autoSize есть важное взаимодействие со свойствами multiline и wordWrap. В табл. C.22 показано поведение autoSize по отношению к multiline и wordWrap; читая таблицу слева направо, можно узнать, от чего зависит изменение размеров текстового поля. По умолчанию multiline и wordWrap равны false.

Таблица C.22. Поведение свойства autoSize

autoSize	multiline	wordWrap	Изменение вправо?	Изменение влево?	Изменение вниз?
false или "none"	true или false	true или false	нет	нет	нет
true или "left"	false	false	да	нет	нет
true или "left"	true	false	да	нет	да (только при переводе строки)

[1] С этим свойством связана одна забавная ошибка реализации для текстовых полей ввода (input textfields), создаваемых нединамически. Если текстовое поле ввода создать в редакторе и в инспекторе свойств задать для него выравнивание по правому краю, а потом в сценарии сменить его на выравнивание по центру, то размеры поля начнут самопроизвольно увеличиваться, как только поле получит фокус. – *Примеч. науч. ред.*

autoSize	multiline	wordWrap	Изменение вправо?	Изменение влево?	Изменение вниз?
true или "left"	false	true	нет	нет	да (только при переполнении текста)
true или "left"	true	true	нет	нет	да (при переполнении текста и при переводе строки)
"right"	false	false	нет	да	нет
"right"	true	false	нет	да	да (только при переводе строки)
"right"	false	true	нет	нет	да (только при переполнении текста)
"right"	true	true	нет	нет	да (при переполнении текста и при переводе строки)
"center"	false	false	да	да	нет
"center"	true	false	да	да	да (только при переводе строки)
"center"	false	true	нет	нет	да (только при переполнении текста)
"center"	true	true	нет	нет	да (при переполнении текста и при переводе строки)

Свойство autoSize отменяет абсолютные размеры, заданные в *createTextField( )*, _height или _ width.

Пример

Следующий код вызывает увеличение или сокращение рамки title_txt в обе стороны, когда в поле изменяется текст:

```
this.createTextField("title_txt", 1, 0, 0, 0, 0);
title_txt.text = "Title Goes Here";
title_txt.border = true;
title_txt.autoSize = "center";
```

Здесь для paragraph_txt задаются фиксированная ширина 100 и расширяемая нижняя граница:

```
this.createTextField("paragraph_txt", 1, 0, 0, 100, 0);
paragraph_txt.text = "The following sentence is a pangram.\n\n"
                     + "The quick brown fox jumps over a lazy dog.";
paragraph_txt.border = true;
paragraph_txt.multiline = true;
paragraph_txt.wordWrap = true;
paragraph_txt.autoSize = "left";
```

См. также

*TextField.*_height, *TextField.*multiline, *TextField.*_width, *TextField.*wordWrap

TextField.background, свойство

Flash 6; во Flash 4 и Flash 5 доступно только в среде разработки

булево значение; включает и выключает отображение фона текстового поля **чтение/запись**

theField.background

Описание

Булево свойство background определяет, следует ли отображать под текстом поля окрашенный прямоугольник (true) или сделать прямоугольник границ поля прозрачным (false, по умолчанию). Цвет прямоугольника определяется свойством backgroundColor (по умолчанию белый). Для изменения размеров прямоугольника предназначены свойства _width и _height. Рамка к прямоугольнику добавляется при помощи свойств border и borderColor.

Если вам нужен более интересный фон для текстового поля, задайте значение false для background и border и вручную нарисуйте фигуру под текстовым полем.

Пример

Задать белый фон для theField_txt:

```
this.createTextField("theField_txt", 1, 0, 0, 80, 20);
theField_txt.text = "hello world";
theField_txt.background = true;
```

Теперь задать красный фон:

```
theField_txt.backgroundColor = 0xFF0000;
```

См. также

TextField.backgroundColor, *TextField*.border, *TextField*.borderColor, *TextField*._width, *TextField*._height

TextField.backgroundColor, свойство

Flash 6

RGB-значение цвета фона текстового поля **чтение/запись**

theField.backgroundColor

Описание

Целочисленное свойство backgroundColor определяет цвет фона текстового поля. Цвет задается одним числом, чаще всего как 6 шестнадцатеричных цифр в формате 0x*RRGGBB*, где *RR*, *GG* и *BB* — двузначные числа от 00 до FF, представляющие красную, зеленую и синюю составляющие. Например, RGB-триплет (R: 51, G: 51, B: 102) эквивалентен шестнадцатеричному числу:

```
0x333366
```

Значение свойства по умолчанию – белый цвет (0xFFFFFF). Дополнительную информацию о тройках RGB можно найти в описании класса *Color*.

Свойство backgroundColor дает видимый эффект только тогда, когда свойство background имеет значение true. В противном случае фон не отображается, даже если задано значение backgroundColor. Например, следующий код пытается задать синий цвет фона для theField_txt, но этого цвета не будет видно, потому что еще не задано свойство background:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "this text has a blue background";
theField_txt.backgroundColor = 0x0000FF;
```

Чтобы стал виден синий фон, надо добавить такую строчку:

```
theField_txt.background = true;
```

Задание значения background не изменяет значения backgroundColor.

См также

TextField.background, *TextField*.border, *TextField*.borderColor, класс *Color*

TextField.border, свойство

Flash 6; во Flash 4 и Flash 5 доступно только в среде разработки

булево значение; включает и выключает вывод рамки текстового поля **чтение/запись**

theField.border

Описание

Булево свойство border определяет, надо выводить контур ограничивающего *theField* прямоугольника (true) или не надо (false). Цвет контура задается свойством borderColor (по умолчанию черный). Размер прямоугольника изменяется с помощью свойств _width и _height, а окраска фона прямоугольника – с помощью свойств background и backgroundColor.

Фон для текстового поля можно создать самостоятельно, для чего надо задать значение false для background и border и вручную нарисовать фигуру под текстовым полем.

Пример

Начертить черную рамку вокруг theField_txt:

```
this.createTextField("theField_txt", 1, 0, 0, 80, 20);
theField_txt.border = true;
theField_txt.text = "hello world";
```

Сделать рамку синей:

```
theField_txt.borderColor = 0x0000FF;
```

См. также

TextField.background, *TextField*.backgroundColor, *TextField*.borderColor, *TextField*._height, *TextField*._width

TextField.borderColor, свойство

Flash 6

RGB-значение цвета рамки текстового поля

theField.borderColor

Описание

Целочисленное свойство borderColor определяет цвет контура ограничивающего текстовое поле прямоугольника. Цвет задается в виде одного числа, чаще всего как 6 шестнадцатеричных цифр в формате 0x*RRGGBB*, где *RR*, *GG* и *BB* – двузначные числа от 00

до FF, представляющие красную, зеленую и синюю составляющие. Например, RGB-триплет (R: 51, G: 51, B: 102) эквивалентен шестнадцатеричному числу:

```
0x333366
```

По умолчанию для рамки задан черный цвет (0x000000). Дополнительную информацию о триплетах RGB можно найти в описании класса *Color*.

Действие свойства borderColor заметно только тогда, когда свойство border имеет значение true; если border имеет значение false, фон не отображается, даже если задано значение borderColor. Например, следующий код пытается задать зеленый цвет рамки для theField_txt, но этот цвет не будет виден, потому что еще не задано свойство border:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "this text has a green border";
theField_txt.borderColor = 0x00FF00;
```

Чтобы появилась зеленая рамка, надо добавить такую строчку:

```
theField_txt.border = true;
```

Значение border не влияет на значение borderColor.

См. также

TextField.background, *TextField*.backgroundColor, *TextField*.border, класс *Color*

TextField.bottomScroll, свойство

Flash 6

индекс самой нижней видимой строки текстового поля, отсчитываемый от 1 — **только для чтения**

theField.bottomScroll

Описание

Свойство bottomScroll представляет собой целое число, равное номеру самой нижней строки, отображаемой в данный момент в текстовом поле. Нумерация начинается с 1.

Если текстовое поле не может вместить весь текст, то лишние строки не выводятся, но остаются частью текстового поля. Верхняя строка имеет номер 1, и все строки текстового поля нумеруются в возрастающем порядке, включая те, которые не входят в видимые границы поля. См. рис. C.6 для *TextField*.scroll, где приведен пример нумерации строк текстового поля.

Диапазон отображаемых в текстовом поле строк определяется с помощью свойств scroll и bottomScroll. Свойство maxScroll возвращает индекс верхней видимой строки при максимально возможной прокрутке поля, но оно не сообщает общее количество строк в поле. Отдельного такого свойства нет, но индекс последней строки можно определить, временно присвоив scroll значение maxScroll и проверив значение bottomScroll:

```
theField_txt.oldScroll = theField_txt.scroll;
theField_txt.scroll = theField_txt.maxscroll;
trace("Number of lines in text field is " + theField_txt.bottomScroll);
theField_txt.scroll = theField_txt.oldScroll;
```

Упражнение: Создайте пользовательское свойство maxBottomScroll, которое будет хранить количество строк в поле. Напишите функцию *getMaxBottomScroll( )*, которая извлекала бы его значение.

Пример

Можно вычислить количество строк, отображаемых в данный момент в текстовом поле (иногда называемое *размером страницы* поля), если вычесть scroll-1 из bottomScroll. Следующий код добавляет во все объекты *TextField* метод *getPageSize( )*, возвращающий количество строк, доступных для просмотра в данном поле:

```
TextField.prototype.getPageSize = function () {
  var pageSize = this.bottomScroll - (this.scroll - 1);
  trace("Page size is " + pageSize);
  return pageSize;
}
```

Размер страницы текстового поля можно использовать, чтобы пропорционально проградуировать ручку прокрутки для текстового поля. Учтите, что количество отображаемых в поле строк может различаться в зависимости от высоты шрифта выводимого текста.

См. также

TextField.maxscroll, *TextField*.scroll; компонент *FScrollBar* в приложении E

TextField.condenseWhite, свойство

Flash 6 (не документировано)

управляет сжатием пробельных символов в тексте HTML **чтение/запись**

theField.condenseWhite

Описание

Если булево свойство condenseWhite имеет значение true, текст HTML в поле подвергается сжатию с удалением всех лишних пробельных символов (табуляции, переноса строки, пробела и т. д.). В результате вывод HTML поля выглядит как в типичном веб-броузере: группы пробелов ведут себя как один пробел между словами, а символ перевода строки "\n" больше не вызывает ее перевод, и вместо него следует использовать теги <P> и <BR>. В таком режиме разбор текста HTML происходит как в анализаторе XML во Flash, когда свойству *XML*.ignoreWhite присвоено значение true.

Свойство condenseWhite позволяет использовать во Flash существующий текст HTML без удаления лишних пробельных символов вручную. По умолчанию condenseWhite равно false. Даже если его значение true, оно не оказывает эффекта, если для свойства html текстового поля также не задано значение true.

Свойство condenseWhite — недокументированная возможность объекта *TextField*, но можно рассчитывать, что его поддержка сохранится и в будущих версиях. Очень осторожные разработчики могут вручную удалять пробельные символы из текста HTML, чтобы он выглядел во Flash так же, как в веб-броузере.

См. также

TextField.html, *TextField*.htmlText, *XML*.ignoreWhite

TextField.embedFonts, свойство

Flash 6

определяет использование экспортированных с фильмом шрифтов при выводе текста **чтение/запись**

theField.embedFonts

Описание

Булево свойство `embedFonts` определяет, выводить ли текст в поле с помощью *шрифтов устройств* (*device fonts*) системы пользователя (`false`) или шрифтов, встроенных в файл *.swf* (`true`). Если `embedFonts` равно `false`, то текст выводится без сглаживания, а отсутствующий шрифт заменяется системным шрифтом по умолчанию (обычно каким-либо вариантом Times New Roman). Если `embedFonts` равен `true` и шрифты поля экспортированы с файлом *.swf*, то к тексту применяется сглаживание. Если `embedFonts` равен `true`, но один или несколько шрифтов поля отсутствуют в файле *.swf*, то текст, шрифт для которого отсутствует, не отображается вообще. Встраивание полного шрифта Roman обычно увеличивает размер фильма на 20–30 Кбайт (для азиатских шрифтов увеличение может быть гораздо больше). Не рекомендуется использовать шрифты с кеглем менее 10 пунктов, потому что при этом в большинстве шрифтов сглаженный текст становится неразборчивым.

Содержимое текстового поля, которое повернуто или маскировано маскирующим слоем этапа разработки, не будет выведено на экран, если его шрифт не встроен в файл. Однако начиная с Flash Player 6.0.40.0 можно применять к текстовому полю прямоугольную маску на этапе исполнения с помощью *MovieClip.setMask( )*. Подробности см. в описании метода *MovieClip.setMask( )*.

Чтобы экспортировать (то есть встроить) шрифт в файл *.swf*, необходимо сделать одно из двух:

- Создать новый символ шрифта в библиотеке фильма.
- На этапе разработки вручную добавить фиктивное текстовое поле и задать его параметры символов в инспекторе свойств.

В первом случае весь шрифт всегда экспортируется полностью, тогда как при создании фиктивного текстового поля можно экспортировать лишь часть шрифта, сократив размер файла. Для того чтобы экспортировать часть шрифта с символом шрифта, надо воспользоваться программой для создания шрифтов, такой как Macromedia Fontographer, создав в ней пользовательский шрифт как подмножество исходного.

Для того чтобы экспортировать Arial Bold с помощью символа шрифта, следует:

1. Выбрать Window → Library.
2. Во всплывающем меню Options в правом верхнем углу окна выбрать New Font. Появится окно диалога Font Symbol Properties.
3. В списке Font выбрать шрифт Arial.
4. В группе Style установить флажок Bold.
5. В поле Name ввести `ArialBold` (имя, используемое только в библиотеке).
6. В библиотеке выбрать символ шрифта `ArialBold`.
7. Во всплывающем меню Options выбрать Linkage.
8. В окне диалога Symbol Linkage Properties для Linkage выбрать Export For ActionScript.
9. В окне Identifier ввести `ArialBold`. Использовать потом это имя в качестве значения свойства `font` объекта *TextFormat*.

Для того чтобы экспортировать любой шрифт с помощью фиктивного текстового поля, необходимо:

1. Выбрать инструмент Text.
2. Начертить на сцене прямоугольник.
3. Выбрать нужный шрифт в инспекторе свойств. Действительное имя шрифта использовать потом в качестве значения свойства `font` объекта *TextFormat*.
4. В инспекторе свойств щелкнуть по кнопке Character и выбрать экспортируемые символы.

Каждый вид стиля шрифта надо встраивать отдельно. Если в текстовом поле используются полужирный, курсив и полужирный курсив Courier New, то надо экспортировать все три разновидности шрифта, иначе текст будет выводиться неправильно. Подчеркивание, а также высота или цвет шрифта не считаются отдельными разновидностями.

Присваивание `embedFonts` значения `true` не вызывает загрузки каких-либо шрифтов вместе с фильмом, а лишь указывает, что содержимое *theField* надо выводить с применением экспортированных шрифтов, если они есть. Свойство `embedFonts` необходимо устанавливать отдельно для каждого текстового поля, использующего определенный шрифт, даже если в нескольких текстовых полях задан один и тот же шрифт. Однако если один и тот же шрифт встроен в несколько текстовых полей, это не отражается на размере файла: с фильмом загружается только один экземпляр этого шрифта.

Шрифт для текстового поля задается при помощи объекта *TextFormat* и методов *TextField.setTextFormat( )* и *TextField.setNewTextFormat( )*. Для получения списка шрифтов, установленных на машине пользователя, предназначен метод *TextField.getFontList( )*.

В операционных системах, поддерживающих Unicode (например, Windows XP и Mac OS X), если `embedFonts` равно `false` и нужен символ, которого нет в заданном шрифте, Flash автоматически ищет в системе шрифт, содержащий этот символ. Если подходящий шрифт найден, он будет использоваться, пока не встретится очередной отсутствующий символ, и тогда Flash снова станет искать подходящий шрифт.

Ошибки реализации

Теоретически, чтобы задать одно и то же значение `embedFonts` для всех текстовых полей фильма, следовало бы использовать такой синтаксис:

```
TextField.prototype.embedFonts = true;
```

Однако во Flash 6 встроенные свойства, задаваемые в `TextField.prototype`, не наследуются экземплярами текстовых полей.

Пример

Предположим, что экспортирован шрифт Verdana, тогда в результате выполнения следующего кода содержимое `theField_txt` будет выведено шрифтом Verdana, со сглаживанием:

```
// Создать текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
// Назначить использование встроенных шрифтов
theField_txt.embedFonts = true;
// Поместить в поле какой-нибудь текст
theField_txt.text = "How are you?"
```

```
// Создать и применить формат текста с шрифтом Verdana
defaultFormat = new TextFormat();
defaultFormat.font = "Verdana";
theField_txt.setTextFormat(defaultFormat);
```

См. также

TextField.getFontList( ), *TextField.setNewTextFormat( )*, *TextField.setTextFormat( )*, *TextFormat*.font

TextField.getDepth(), метод

Flash 6

возвращает позицию текстового поля в стеке содержимого

theField.getDepth()

Возвращаемое значение

Целое число – уровень глубины.

Описание

Текстовые поля, клипы и кнопки располагаются в проигрывателе в некотором порядке, определяющем, как объекты размещаются один над другим. Внутреннее свойство глубины указывает порядок каждого объекта в стеке содержимого. Метод *getDepth( )* возвращает целое число – расположение поля по глубине. На каждом уровне глубины может находиться только один объект, будь то текстовое поле, кнопка или клип. Объекты с большим значением глубины располагаются перед теми, у которых значение глубины меньше. Для текстовых полей, создаваемых на этапе исполнения с помощью *MovieClip.createTextField( )*, метод *getDepth( )* возвращает то значение глубины, которое было присвоено параметру depth. Например:

```
// Задает глубину 13
this.createTextField("theField_txt", 13, 0, 0, 150, 20);
trace(theField_txt.getDepth());  // Выводит: 13
```

Для текстовых полей, создаваемых вручную на этапе разработки, *getDepth( )* возвращает глубину в диапазоне от –16 383 до –1 включительно. В отличие от класса *MovieClip*, у *TextField* нет метода *swapDepths( )*, поэтому глубина текстового поля остается неизменной после того, как ей присвоено некоторое значение.

Значительно подробнее о глубинах объектов рассказано в разделе «Порядок расположения фильмов и экземпляров в стеке» главы 13.

См. также

MovieClip.createTextField( ), *MovieClip.getDepth( )*, *MovieClip.swapDepths( )*; «Порядок расположения фильмов и экземпляров в стеке» в главе 13

TextField.getFontList(), метод класса

Flash 6

возвращает массив шрифтов, установленных в системе пользователя

TextField.getFontList()

Возвращаемое значение

Массив строк, содержащих названия шрифтов.

Описание

Метод класса *getFontList( )* сообщает, какие шрифты установлены на машине, где воспроизводится фильм Flash. Любой из перечисленных шрифтов может быть использован в качестве шрифта устройства для текстового поля при условии, что свойство `embedFonts` этого поля равно `false`. Шрифт для текстового поля назначается при помощи свойства `font` объекта *TextFormat* и метода *TextField.setTextFormat( )*.

Метод *getFontList( )* перечисляет только семейства шрифтов, как они были бы показаны инспектором свойств для этого текстового поля. Разновидности шрифта, такие как полужирное или курсивное начертание, не указываются. Чтобы использовать полужирное или курсивное начертание шрифта, надо кроме свойства `font` задать свойства `bold` или `italic` объекта *TextFormat*, а затем применить формат с помощью *TextField.setTextFormat( )* или *TextField. setNewTextFormat( )*.

Обычно *getFontList( )* применяется для автоматического выбора шрифта вместо отсутствующего (см. пример ниже) и предоставления пользователю возможности выбрать шрифт.

Применение

Обратите внимание, что *getFontList( )* вызывается непосредственно как метод класса *TextField*, а не через экземпляр этого класса:

```
someTextField_txt.getFontList();  // Неверно...не сработает.
TextField.getFontList()           // Правильно.
```

Ошибки реализации

Во Flash 6 *getFontList( )* не возвращает имена шрифтов, экспортированных вместе с фильмом.

Пример

Функция *assignFont( )* принимает массив с названиями шрифтов и возвращает первый из них, который обнаружен на машине пользователя. Если не найден ни один шрифт, *assignFont( )* возвращает `null`:

```
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
theField_txt.text = "Flash is fun!";
defaultFormat = new TextFormat();
defaultFormat.font = assignFont(["ZapfChancery", "Verdana", "Arial", "_sans"]);
theField_txt.setTextFormat(defaultFormat);
function assignFont (fontList) {
  var systemFonts = TextField.getFontList();
  for (var i = 0; i < fontList.length; i++) {
    for (var j = 0; j < systemFonts.length; j++) {
      if (fontList[i] == systemFonts[j]) {
         return fontList[i];
      }
    }
  }
  return null;
}
```

См. также

TextField.`embedFonts`, *TextField.setTextFormat( )*, *TextFormat*.`bold`, *TextFormat*.`font`, *TextFormat*.`italic`, класс *TextFormat*

TextField.getNewTextFormat(), метод

Flash 6

возвращает объект TextFormat по умолчанию для текстового поля

theField.getNewTextFormat()

Возвращаемое значение

Объект *TextFormat*, описывающий формат по умолчанию для символов, помещаемых в *theField*.

Описание

Метод *getNewTextFormat( )* возвращает объект *TextFormat*, свойства которого описывают форматирование, применяемое ко всем новым символам, помещаемым в *theField*. Объекты *TextFormat* подробно рассмотрены в описании класса *TextFormat*.

Любой текст, добавляемый в поле с помощью *TextField*.text, *TextField.replaceSel( )* или пользователем, форматируется в соответствии с настройками объекта, возвращаемого *getNewTextFormat( )*. Значения свойств объекта *TextFormat* по умолчанию, применяемого к вновь созданному текстовому полю, показаны в табл. С.25 в «Описании» статьи, посвященой классу *TextFormat*.

Значения по умолчанию можно изменить с помощью *setNewTextFormat( )*. Однако *setNewTextFormat( )* не форматирует символы, уже имеющиеся в поле, а задает формат для тех символов, которые будут добавляться впоследствии. Для изменения формата символов, уже находящихся в текстовом поле, применяется метод *setTextFormat( )*. Чтобы узнать, какой формат имеют символы, уже находящиеся в текстовом поле, применяется метод *getTextFormat( )*.

Применение

Если точка ввода помещается в текстовое поле пользователем или методом *Selection.setSelection( )*, то новый формат текста устанавливается в соответствии с форматом символа справа от точки ввода (или слева, если точка ввода находится за последним символом). Значение, возвращаемое *getNewTextFormat( )*, отражает формат этого символа. В примере для *TextField.setNewTextFormat( )* показан код, который устанавливает новый формат текста, не копирующий уже существующий.

Пример

Следующий код получает размер шрифта по умолчанию для поля с именем numbers_txt, а затем увеличивает его на 10 пунктов:

```
// Создать текстовое поле
this.createTextField("numbers_txt", 1, 0, 0, 150, 40);
// Поместить в поле какой-нибудь текст
numbers_txt.text = "one two three four";
// Получить объект формата нового текста для numbers_txt
var numbersNewTextFormat = numbers_txt.getNewTextFormat();
// Получить размер шрифта по умолчанию для нового текста
trace(numbersNewTextFormat.size);  // Выводит: 12
// Добавить 10 пунктов и применить новые настройки по умолчанию
numbersNewTextFormat.size += 10;
numbers_txt.setNewTextFormat(numbersNewTextFormat);
// Ввести какой-нибудь текст, чтобы увидеть результат
numbers_txt.text = "five six seven";
```

См. также

TextField.getTextFormat( ), *TextField.setNewTextFormat( )*, *TextField.setTextFormat( )*, класс *TextFormat*

TextField.getTextFormat(), метод

Flash 6

возвращает объект TextFormat для заданных символов

```
theField.getTextFormat()
theField.getTextFormat(beginIndex)
theField.getTextFormat(beginIndex, endIndex)
```

Аргументы

beginIndex

Индекс первого символа, форматирование которого описывается возвращаемым объектом *TextFormat*. Если этот аргумент присутствует, он должен быть неотрицательным целым числом. Если он опущен, используется все поле.

endIndex

Индекс символа, следующего за последним, форматирование которого описывается возвращаемым объектом *TextFormat*. Если этот аргумент присутствует, он должен быть неотрицательным целым числом.

Возвращаемое значение

Объект *TextFormat*, описывающий форматирование указанных символов.

Описание

Метод *getTextFormat( )* позволяет выяснить формат одного или нескольких символов, уже находящихся в поле. Он возвращает объект *TextFormat*, свойства которого описывают форматирование указанных символов. Подробное описание объектов *TextFormat* см. в описании класса *TextFormat*.

Если *getTextFormat( )* вызывается с одним аргументом или *endIndex* равен *beginIndex* + 1, то возвращаемый объект *TextFormat* отражает форматирование одного символа в позиции *beginIndex*. Применим форматирование к первым четырем символам текстового поля, а затем проверим шрифт первого символа:

```
// Создать текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
// Поместить в поле какой-нибудь текст
theField_txt.text = "What time is it?";
// Создать объект TextFormat со свойством font равным "Arial"
sansFormat = new TextFormat("Arial");
// Применить форматирование к слову 'What', символы с 0 по 3
theField_txt.setTextFormat(0, 4, sansFormat);
// Получить объект TextFormat для первого символа
firstCharFormat = theField_txt.getTextFormat(0);
// Проверить свойство font
trace(firstCharFormat.font);  // Выводит: Arial
```

Если *getTextFormat( )* вызывается с двумя аргументами, то возвращаемый объект *TextFormat* представляет форматирование для диапазона символов с *beginIndex* по *endIndex* - 1. А если *getTextFormat( )* вызывается без аргументов, то возвращаемый объект *TextFormat* представляет форматирование всех символов поля.

Если у символов указанного диапазона какая-то характеристика формата (название шрифта, полужирное или курсивное начертание) оказывается различной, соответствующее свойство объекта *TextFormat* получает значение `null`. Продолжив наш пример, получим объект *TextFormat* для всего текста в `theField_txt` и убедимся, что одинаковые свойства всех символов возвращают значения, отличные от `null`:

```
// Получить объект TextFormat для всех символов theField_txt
allCharsFormat = theField_txt.getTextFormat();
// Проверим, все ли символы полужирные
trace(allCharsFormat.bold);  // Выводит: false
```

Но свойства, меняющиеся от символа к символу, возвращают `null`:

```
// Получим шрифт, которым отформатированы все символы
trace(allCharsFormat.font);  // Выводит: null (если шрифты разные)
```

Шрифт первых четырех символов `theField_txt` отличается от шрифта остальных символов, поэтому одним значением свойства `font` нельзя описать весь диапазон, и возвращается `null`.

Обратите внимание, что изменения в объекте *TextFormat*, возвращаемом *getTextFormat( )*, не оказывают никакого воздействия на текст в `theField`, пока этот объект не будет применен с помощью *setTextFormat( )*. Например, само по себе следующее присваивание свойству `font` не оказывает влияния на `theField_txt`:

```
allCharsFormat.font = "Courier New";
```

Но после вызова *setTextFormat( )* изменения в объекте применяются к формату:

```
theField_txt.setTextFormat(allCharsFormat);  // Применяет шрифт "Courier New"
                                             // ко всему текстовому полю
```

Не путайте *getNewTextFormat( )* с *getTextFormat( )* – первый метод возвращает формат, который будет применен к тексту, добавляемому в *theField*, а второй возвращает формат символов, уже имеющихся в текстовом поле.

Объекты *TextFormat* подробно рассмотрены в описании класса *TextFormat*.

См. также

TextField.getNewTextFormat( ), *TextField.setNewTextFormat( )*, *TextField.setTextFormat( )*, класс *TextFormat*

TextField._height, свойство

Flash 6

высота ограничивающего текстовое поле прямоугольника в пикселах **чтение/запись**

theField.`_height`

Описание

Каждое текстовое поле определяет прямоугольную область, называемую его *ограничивающим прямоугольником* (*bounding box*), внутри которой выводится текст. Свойство `_height` представляет собой неотрицательное число с плавающей точкой и задает высоту ограничивающего прямоугольника в пикселах. Изменяя `_height`, мы изменяем ограничивающий прямоугольник поля, в результате чего становится видна большая или меньшая часть текста. При изменении `_height` не изменяется высота содержащегося в поле текста. Для изменения высоты текста предназначены свойства *TextFormat*.`size` и *TextField*.`_yscale`. Узнать высоту текста в поле можно с помощью `textHeight`.

Во Flash 6 появилась возможность изменять размеры ограничивающего прямоугольника в соответствии с текстом, с помощью свойства autoSize.

Изменение высоты поля в среде разработки или с помощью _yscale находят отражение в свойстве _height. Для отображения контура или заливки ограничивающего прямоугольника предназначены свойства border и background.

Пример

```
// Создать текстовое поле высотой 10 пикселов
// (с рамкой, чтобы видеть ограничивающий прямоугольник)
this.createTextField("theField_txt", 1, 0, 0, 150, 10);
theField_txt.border = true;
// Поместить в поле какой-нибудь текст
theField_txt.text = "Almost time for breakfast.";
// Сделать высоту theField_txt равной 20 пикселам
theField_txt._height = 20;
// Прибавить 15 пикселов к высоте theField_txt
theField_txt._height += 15;
```

См. также

TextField.autoSize, *TextField*.textHeight, *TextField*._width, *TextField*._yscale, класс ***TextFormat***

TextField.hscroll, свойство

Flash 6

величина горизонтальной прокрутки текстового поля в пикселах **чтение/запись**

theField.hscroll

Описание

Если текст поля слишком велик по ширине или высоте, то текст, выступающий за пределы физической области просмотра ограничивающего прямоугольника, не виден. Для просмотра текста можно выполнить горизонтальную или вертикальную прокрутку поля с помощью свойств hscroll и scroll. Свойство hscroll содержит целое число, указывающее количество пикселов, на которое текст поля прокручен влево. По умолчанию оно равно 0, что указывает на отсутствие горизонтальной прокрутки текста (первый символ в поле располагается сразу у левого края ограничивающего прямоугольника). Если текст поля может быть выведен полностью, не будучи усеченным по горизонтали, hscroll остается равным 0 и не может быть изменено. Но если текст поля нельзя полностью показать по горизонтали в отведенном для этого пространстве, то можно присвоить hscroll положительное целое число, не превышающее maxhscroll, в результате чего текст будет смещен влево на соответствующее количество пикселов. Если в поле отображается несколько строк, все они сдвигаются одновременно. Если ввод данных пользователем приводит к прокрутке текста, это отражается в hscroll. Обратите внимание, что hscroll прокручивает текст не по одному символу, а по одному пикселу. Не так ведет себя свойство scroll, прокручивающее текст по одной строке, а не по одному пикселу.

Свойство hscroll обычно применяется для создания интерфейса текстового поля с горизонтальной прокруткой или вывода текста в стиле бегущей строки с котировками акций, как в следующем примере.

Пример

В примере C.5 показана простая, но гибкая бегущая строка, в которой свойство `hscroll` используется для прокрутки текста по горизонтали в бесконечном цикле.

Пример C.5. Бегущая строка

```
// Создать текстовое поле (с рамкой, чтобы видеть ограничивающий прямоугольник)
this.createTextField("ticker_txt", 1, 0, 0, 100, 20);
ticker_txt.border = true;
// Задать скорость ticker_txt (пикселов/20 мс)
ticker_txt.speed = 1;
// Поместить в поле какой-нибудь текст
ticker_txt.text = "Here is the latest news from moock.org: " + "Colin drinks a cup of tea!";
// Задать буфер из пробелов, чтобы текст вплывал и выплывал из текстового поля
// Сначала узнаем размер пробела в формате этого поля
ticker_txt.spaceSize = ticker_txt.getNewTextFormat().getTextExtent(" ").width;
// Вычислим, сколько пробелов нужно для заполнения поля
ticker_txt.spacesRequired = Math.ceil(ticker_txt._width / ticker_txt.spaceSize);
// Создать строку пробелов
for (var i = 0; i < ticker_txt.spacesRequired; i++) {
  ticker_txt.spacebuffer += " ";
}
// Добавить пробелы в начало и конец текста
ticker_txt.text = ticker_txt.spacebuffer + ticker_txt.text + ticker_txt.spacebuffer;
// Функция для прокрутки текста
ticker_txt.hscrollInterval = function () {
  // Если мы в конце текста...
  if (this.hscroll == this.maxhscroll) {
    // ...начать сначала
    this.hscroll = 0;
  }
  // Прокрутить текст на заданную величину
  this.hscroll += this.speed;
}
// Вызывать функцию прокрутки каждые 20 мс
setInterval(ticker_txt, "hscrollInterval", 20);
```

См. также

setInterval( ), ***TextField.getNewTextFormat( )***, *TextField*.maxhscroll, *TextField*.scroll, *TextField*.width, *TextField*.wordWrap, *TextField*._yscale, ***TextFormat.getTextExtent( )***; описание компонента *FScrollBar* в приложении G

TextField.html, свойство

Flash 6; во Flash 5 доступно только в среде разработки

булево значение; включает и выключает показ HTML **чтение/запись**

theField.html

Описание

Булево свойство `html` определяет, должно ли содержимое свойства `htmlText` выводиться как форматированный HTML (`true`) или как обычный текст (`false`). По умолчанию имеет значение `false`. Свойство `html` действует только в том случае, если значение `true` ему присваивается раньше, чем присваивается значение свойству `htmlText`. Во Flash 5 HTML был единственным способом получить в текстовом поле сложное форматирование. Во Flash 6 управлять форматированием можно с помощью класса *TextFormat*.

Дополнительные сведения см. в *TextField*.htmlText.

Пример

Следующий код создает текстовое поле, текст которого выводится курсивом:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.html = true;
theField_txt.htmlText = '<I>This will display italicized text.</I>';
```

В текстовом поле выводится текст:

```
This will display italicized text.
```

Если пропустить строку 2, то содержимое поля будет выглядеть так:

```
<I>This will display italicized text.</I>
```

См. также

TextField.condenseWhite, *TextField*.htmlText, *TextField*.text, класс *TextFormat*; приложение E

TextField.htmlText, свойство

Flash 6

исходный код HTML, который должен быть выведен в текстовом поле **чтение/запись**

theField.htmlText

Описание

Подобно свойству text свойство htmlText задает текст, который должен выводиться на экране, но дополнительно, если свойство html равно true, теги HTML в htmlText интерпретируются как инструкции форматирования. Чтобы показать отформатированный HTML в текстовом поле, сначала присвойте html значение true, а затем присвойте htmlText исходный код HTML:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.html = true;
theField_txt.htmlText = '<B>This will display bold text.</B>';
```

По сравнению с веб-броузером набор поддерживаемых тегов ограничен, но все же предоставляет хорошие возможности для управления текстом. Подробности см. в приложении E.

Присваивание значения свойству htmlText изменяет значение text и наоборот. В большинстве случаев при использовании текста HTML следует задать для свойства html значение true и обращаться к тексту поля через свойство htmlText. При работе с текстом, не являющимся HTML, присвойте свойству html значение false и обращайтесь к тексту поля через свойство text.

Если свойство html равно false, то теги HTML в тексте, присвоенном htmlText, видны как обычный текст, и htmlText ведет себя так же, как свойство text. Теги HTML, присваиваемые непосредственно свойству text, не интерпретируются как HTML и всегда отображаются буквально.

Однако если html равно true и свойству htmlText присваиваются теги HTML, то значение text будет таким же, как значение htmlText, но с удалением всех тегов HTML.

Создадим текстовое поле с поддержкой HTML и присвоим какой-нибудь HTML его свойству `htmlText`:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20); theField_txt.html = true;
theField_txt.htmlText = '<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12"'
                        + 'COLOR="#000000">This field contains <B>HTML!</B>'
                        + '</FONT></P>';
```

После присваивания `htmlText` будет иметь значение:

```
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000">
 This field contains <B>HTML!</B></FONT></P>
```

Но значение `text` будет таким:

```
This field contains HTML!
```

Наоборот, если `html` имеет значение `true`, Flash автоматически добавляет разметку HTML в `htmlText`, если исходный код HTML, присвоенный `htmlText`, не полностью описывает форматирование текстового поля. Например, зададим значение `htmlText` без тегов `<P>` или `<FONT>`:

```
theField_txt.htmlText = "This field contains <B>HTML!</B>";
```

Посмотрев теперь на значение `theField_txt.htmlText`, можно обнаружить, что добавились теги `<P>` и `<FONT>`:

```
trace(theField_txt.htmlText);
// Выводит:
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000">
This field contains <B>HTML!</B></FONT></P>
```

В то же время в `text` содержится значение:

```
This field contains HTML!
```

Если *theField*.`html` равно `true` и *theField* отформатировано с помощью *setTextFormat( )*, то `htmlText` представляет форматирование тегами HTML. Например:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 30);
theField_txt.html = true;
theField_txt.text = "Stretch often while you work.";
// Создать объект TextFormat с полужирным шрифтом.
boldFormat = new TextFormat();
boldFormat.bold = true;
boldFormat.size = 20;
// Применить этот формат к слову "often", символы с 8 по 12.
theField_txt.setTextFormat(8, 13, boldFormat);
// Получить значение htmlText. Обратите внимание на теги <B> вокруг слова "often".
trace(theField_txt.htmlText);
// Выводит:
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000">
Stretch <FONT SIZE="20"><B>often</B></FONT> while you work.</FONT></P>
```

Обратите, однако, внимание, что новый формат текста для *theField* (заданный *setNewTextFormat( )*) не используется в `htmlText`, что показывает следующий код:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 100);
theField_txt.html = true;
// Задать для нового формата текста полужирный шрифт размером 20 пунктов.
tformat = theField_txt.getNewTextFormat();
tformat.size = 20;
```

```
tformat.bold = true;
theField_txt.setNewTextFormat(tformat);
// Этот текст, присвоенный htmlText, не принимает форматирование
theField_txt.htmlText = '<P>This text is size 12, not size 20.</P>';
// Этот текст, присвоенный text, будет полужирным высотой 20 пунктов
theField_txt.setNewTextFormat(tformat);
theField_txt.text = 'This text is size 20, bold.';
```

В табл. С.23 показаны результаты различных присваиваний свойству `htmlText`.

Таблица С.23. Отображение текста при задании значения свойства htmlText

html	Что присваивается htmlText	Содержимое htmlText	Содержимое text	Что выводится на экран
false	<I>Курсивное начертание</I>	<I>Курсивное начертание</I>	<I>Курсивное начертание</I>	<I>Курсивное начертание</I>
false	Обычный текст	Обычный текст	Обычный текст	Обычный текст
true	<I>Курсивное начертание</I>	<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"><I>Курсивное начертание</I></ FONT></P>	Курсивное начертание	*Курсивное начертание*
true	Обычный текст	<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000">Обычный текст</FONT></ P>	Обычный текст	Обычный текст

Учтите, что при присваивании значения одному из свойств `htmlText` и `text` изменяется и второе свойство. Независимо от значения *theField*.`html` присваивание `text` нового значения заменяет значение `htmlText`, и наоборот. Напротив, последовательные *конкатенации* (не повторное присваивание) не меняют значения другого свойства:

```
theField_txt.htmlText = "<B>hello</B>";
theField_txt.text += " world";
// Теперь значение theField_txt.htmlText такое:
// <P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12"
// COLOR="#000000">hello world</FONT></P>
```

Но при дописывании `text` к `htmlText` могут возникнуть проблемы. Когда мы присвоили значение свойству `text`, Flash удалил тег `<B>` из `htmlText`, потому что присваивание нового значения `text` удаляет любое специальное форматирование поля. Кроме того, при каждом конкатенировании или присваивании `htmlText` вставляется тег нового абзаца (`<P>`). Поэтому смешивать присваивания `text` и `htmlText` в целом не рекомендуется. Каждое присваивание `htmlText` должно состоять из целого абзаца, иначе при последующих конкатенациях появится нежелательный перенос строки.

Включая исходный код HTML в текстовое поле с включенной поддержкой HTML, следует преобразовать `<`, `>`, `&`, `"` и `'` в эквивалентные сущности HTML: `<`, `>`, `&`, `"` и `'`, как описано в *http://moock.org/asdg/technotes/sourceInHtmlField*.

Чтобы показать теги HTML в поле с выключенной поддержкой HTML, присвойте `html` значение `false` и пользуйтесь свойством `text`, например:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
```

```
theField_txt.html = false;
theField_txt.text = 'This field contains <B>HTML!</B>';
```

Если свойства embedFonts и html оба имеют значение true, то для каждой гарнитуры шрифта и каждой разновидности, использованной в htmlText, должен иметься соответствующий шрифт, экспортированный с фильмом (как описано в embedFonts). Если мы используем в htmlText Arial bold, Arial italic и Arial bold italic, то должны экспортировать все три разновидности шрифта. Допустим, например, что создается текстовое поле с именем output_txt. В инспекторе свойств для поля output_txt выберем шрифт Arial и зададим начертание Italic. В диалоговом окне Character Options зададим встраивание всех символов (All Characters) шрифта Arial italic. Затем зададим вывод HTML в output_txt. Наконец, присвоим текстовому полю следующее значение:

```
output_txt.htmlText = '<P><I>My</I>, what <B>lovely</B>'
                    + '<FONT SIZE="24">eyes</FONT> you have!</P>';
```

При воспроизведении фильма в текстовом поле появится следующий текст:

My

Все остальное, что было присвоено output_txt, пропало! Выведен только курсивный текст. Остальной текст требует других разновидностей Arial, которые мы не встраивали, – «what» и «eyes you have» без курсива, а «lovely» должно выводиться полужирным шрифтом.

Подчеркивание не считается разновидностью шрифта – так же, как и размер и цвет. Однако если embedFonts равно false, Flash полностью опирается на шрифты, установленные в системе пользователя, и тогда обычный, полужирный и курсивный текст будет отображаться только в том случае, если соответствующая разновидность шрифта установлена в системе.

Подробнее об использовании HTML в текстовых полях рассказано в приложении E.

См. также

getURL(), *TextField*.embedFonts, *TextField*.html, *TextField*.text, класс *TextFormat*; приложение E

TextField.length, свойство

Flash 5; во Flash 6 введена объектно-ориентированная версия

количество символов в текстовом поле **только для чтения**

theField.length

Описание

Целочисленное свойство length возвращает количество символов в *theField*.text (аналогично свойству *String*.length), включая пробельные. Если свойству *theField*.htmlText присвоен исходный код HTML, *theField*.length по-прежнему возвращает количество символов в *theField*.text (совпадающем с htmlText, но с исключением тегов HTML). Свойство length текстового поля дает количество символов, а не ширину поля. Для получения ширины поля в пикселах предназначено свойство *theField*._width. Для получения значения конкретного символа в *theField* применяется *theField*.text.charAt(*n*), где *n* задает номер символа, начинающийся с нуля.

Пример

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
```

```
theField_txt.text = "welcome";
trace(theField_txt.text.length);  // Выводит: 7
```

См. также

String.charAt( ), *String*.length, *TextField*.htmlText, *TextField*.text, *TextField*._width

TextField.maxChars, свойство

Flash 6; во Flash 5 доступно только в среде разработки

ограничивает допустимый объем вводимых пользователем данных **чтение/запись**

theField.maxChars

Описание

Целочисленное свойство maxChars применяется только с текстовыми полями для ввода данных пользователем и ограничивает объем текста, вводимого пользователем в поле. По умолчанию равно null, что позволяет вводить неограниченный объем текста (как и значение 0). Положительные значения позволяют ввести указанное количество символов, а отрицательные полностью запрещают ввод текста. Свойство maxChars не распространяется на изменение текста, осуществляемое через ActionScript. Свойствам text и htmlText можно присваивать любые значения независимо от ограничений, налагаемых maxChars.

Свойство maxChars часто используют в формах ввода, которые требуют данных в определенном формате. Например, с его помощью при вводе даты можно ограничить день двумя цифрами, месяц двумя цифрами и год четырьмя цифрами. Чтобы еще больше ограничить ввод данных, можно с помощью *TextField*.restrict задать набор допустимых символов для текстового поля.

Пример

Следующий код ограничивает количество символов, которые можно ввести в userName_txt, числом 15:

```
this.createTextField("userName_txt", 1, 0, 0, 200, 20);
userName_txt.type = "input";
userName_txt.text = "your name here";
userName_txt.maxChars = 15;
```

См. также

TextField.htmlText, *TextField*.restrict, *TextField*.text, *TextField*._width, *String.charAt( )*, *String*.length

TextField.maxhscroll, свойство

Flash 6

максимальное расстояние, на которое можно прокрутить текст влево **только для чтения**

theField.maxhscroll

Описание

Целочисленное свойство maxhscroll определяет максимальное расстояние, на которое можно прокрутить текст поля по горизонтали влево. Обычно применяется для того, чтобы определить, перемещен ли текст влево до возможного предела, или чтобы прокрутить текст влево на максимально возможное расстояние.

Если для *theField*.type задано значение "dynamic", maxhscroll сообщает, на сколько самая длинная строка *theField* выходит за правый край поля (если hscroll равно 0). Когда *theField*.type равно "input", maxhscroll включает в себя также небольшое пустое пространство в правой части *theField* (приблизительно четверть ширины поля). Этот буфер оставляет место для ввода данных пользователем. Если hscroll равно maxhscroll, то самая длинная строка текста *theField* оказывается вровень с правым краем поля (за вычетом свободного пространства, если *theField*.type равно "input").

На рис. C.4 показано maxhscroll для динамического текстового поля (которое имеет тип "dynamic"). Текст, показанный серым цветом, не выводится во Flash Player, а приведен для облегчения понимания.

Рис. C.4. *Свойство maxhscroll для динамического текстового поля*

На рис. C.5 показано maxhscroll для того же самого поля, когда его свойство type имеет значение "input". Обратите внимание на увеличение maxhscroll, произошедшее в результате изменения type. У поля на рис. C.5 hscroll имеет значение 150, чтобы ясно показать дополнительное пространство, отведенное для ввода данных пользователем.

Рис. C.5. *Свойство maxhscroll для текстового поля ввода*

Пример

Следующий код прокручивает title_txt до конца вправо. Более реалистический вариант применения maxhscroll рассмотрен в примере C.5 для *TextField*.hscroll.

```
this.createTextField("title_txt", 1, 0, 0, 30, 20);
title_txt.text = "Welcome to my web site.";
title_txt.hscroll = title_txt.maxhscroll;
```

См. также

TextField.bottomScroll, *TextField*.hscroll, *TextField*.maxscroll, *TextField*.scroll; компонент *FScrollBar* в приложении G

TextField.maxscroll, свойство

Flash 6; соответствует свойству maxscroll во Flash 4 и Flash 5

последняя допустимая верхняя строка текстового поля **только для чтения**

theField.maxscroll

Описание

Целочисленное свойство maxscroll сообщает наибольшее допустимое значение scroll для текстового поля. Оно представляет собой номер последней строки текстового поля, которая может быть верхней строкой в области просмотра. Обратите внимание, что maxscroll не равно количеству строк текста; последнее следует получить вручную, как описывается в *TextField*.bottomScroll.

Образец кода для прокрутки см. в примере C.7 для *TextField*.scroll.

См. также

TextField.bottomScroll, *TextField*.hscroll, *TextField*.maxhscroll, *TextField*.scroll; компонент *FScrollBar* в приложении G

TextField.multiline, свойство

Flash 6; во Flash 5 доступно только в среде разработки

булево значение; включает и выключает многострочный ввод **чтение/запись**

theField.multiline

Описание

Булево свойство multiline определяет, может ли пользователь разбивать строку в *theField* с помощью клавиш Return или Enter. Относится только к данным, вводимым пользователем, и не влияет на текст, присваиваемый с помощью ActionScript. По умолчанию устанавливается значение false, которое не разрешает пользователю вводить в поле более одной строки текста, фактически отключая клавишу Enter во время ввода текста (подобно тегу HTML <INPUT TYPE="TEXT">). Значение true разрешает пользователю вводить в текстовое поле более одной строки текста (подобно тегу HTML <TEXTAREA>).

Чтобы включить перенос по словам строк, превышающих размеры поля, применяется свойство *TextField*.wordWrap. Объем текста, который пользователю разрешено ввести в текстовое поле, можно ограничить с помощью свойства *TextField*.maxChars.

Пример

Следующий код создает два текстовых поля. Первое, nameInput_txt, допускает ввод только одной строки, а второе, addressInput_txt, – ввод нескольких строк:

```
// Создать поле nameInput_txt
this.createTextField("nameInput_txt", 1, 0, 0, 200, 20);
nameInput_txt.type = "input";
nameInput_txt.border = true;
nameInput_txt.multiline = false;
// Создать поле addressInput_txt
this.createTextField("addressInput_txt", 2, 0, 40, 200, 60);
addressInput_txt.type = "input";
addressInput_txt.border = true;
addressInput_txt.multiline = true;
```

См. также

TextField.maxChars, *TextField*.restrict, *TextField*.type, *TextField*.wordWrap

TextField._name, свойство

Flash 6

имя экземпляра объекта TextField **чтение/запись**

theField.`_name`

Описание

Строковое свойство `_name` задает имя экземпляра поля, которое первоначально определяется в инспекторе свойств на этапе разработки или в методе *MovieClip.createTextField( )* на этапе исполнения. Присваивание значения `_name` задает имя экземпляра для *theField*, хотя необходимость в таком действии едва ли может возникнуть. После присваивания ссылки на прежнее имя экземпляра возвращают `undefined`:

```
// Создать текстовое поле с именем экземпляра address_txt
this.createTextField("address_txt", 1, 0, 0, 200, 60);
// Присвоить новое имя экземпляра
address_txt._name = "location_txt";
// Обратиться к полю через новое имя экземпляра
location_txt.text = "Toronto, Canada";
```

Свойство `_name` чаще применяется для того, чтобы идентифицировать конкретное поле при обработке группы полей. Следующий код обрабатывает некую гипотетическую форму. Он ищет поле `email_txt` на текущей временной диаграмме (`this`) и вызывает для него функцию проверки допустимости данных:

```
for (var p in this) {
  if (this[p] instanceof TextField) {
    // Показать имена всех полей
    trace("found " + this[p]._name);
    // Искать email_txt
    if (this[p]._name == "email_txt") {
      validateAddress(this[p].text);
    }
  }
}
```

См. также *MovieClip.createTextField( )*, *MovieClip*.`_name`; «Цикл *for-in*» в главе 8

TextField.onChanged(), обработчик события и событие приемника

Flash 6

вызывается, когда обнаруживается ввод данных пользователем

theField.`onChanged(`*changedField*`)`
listener.`onChanged(`*changedField*`)`

Аргументы

changedField Объект *TextField*, ввод данных в который был обнаружен.

Описание

Событие *onChanged( )* генерируется, когда пользователь вводит или удаляет текст в *theField*, и обычно применяется для запуска процедур проверки допустимости данных в ответ на действия пользователя, продолжающего ввод с клавиатуры. В число возможных применений входят встроенная проверка орфографии, проверка почтовых адресов или функция автозаполнения текстового поля.

Обратите внимание, что *onChanged( )* не генерируется, когда для *theField* задается новый текст с помощью ActionScript. Например, следующее событие не запускает *onChanged( )*:

```
theField_txt.text = "Some new text...";
```

Если *theField*.restrict имеет значение true и пользователь пытается ввести запрещенный символ, нажатие клавиши игнорируется и *onChanged( )* не генерируется.

Событие *onChanged( )* может обрабатываться функцией обратного вызова, прикрепленной к текстовому полю. Следующий код регистрирует функцию обратного вызова, отвечающую на *onChanged( )*:

```
// Создать текстовое поле для приема данных, вводимых пользователем
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
// Присвоить функцию обратного вызова onChanged() с помощью литерала функции
theField_txt.onChanged = function (changedField) {
  trace("The user typed something in " + this);
};
```

В теле функции обратного вызова можно ссылаться на theField_txt с помощью ключевого слова this. Возможно также использование параметра *changedField* для получения ссылки на текстовое поле, в котором произошло изменение.

Событие *onChanged( )* можно также обрабатывать с помощью приемника событий, добавленного с помощью *addListener( )* и заданного параметром *listener*. Используя приемник событий, надо задать параметр *changedField*, чтобы определить, какое текстовое поле изменилось. Следующий код регистрирует приемник событий, реагирующий на *onChanged( )*:

```
changedListener = new Object();
changedListener.onChanged = function (changedField) {
  trace("The user typed something in " + changedField);
};
theField_txt.addListener(changedListener);
```

Иметь дело с функцией обратного вызова проще, но при этом реагировать на событие может только одна функция. Объектов приемников, реагирующих на одно событие, может быть сколько угодно. Прекратить обработку события *onChanged( )* функцией обратного вызова можно посредством оператора *delete*:

```
// Оператора () после имени функции быть не должно!
delete theField_txt.onChanged;
```

Обработка приемником события *onChanged( )* прекращается с помощью метода *removeListener( )*:

```
theField_txt.removeListener(changedListener);
```

Ошибки реализации

Если текст вводится в поле очень быстро (например, при одновременном нажатии нескольких клавиш), то новое значение theField.text может оказаться еще не заданным, когда выполняется обработчик *onChanged( )*. Если ожидается, что ввод может быть быстрым, воспользуйтесь *setInterval( )* и дождитесь, когда станет доступно новое значение text.

См. также

TextField.addListener( ), *TextField.removeListener( )*, *TextField*.type; глава 10

TextField.onKillFocus(), обработчик события

Flash 6

функция обратного вызова, вызываемая в случае потери полем фокуса

theField.onKillFocus(*newFocus*)

Аргументы

newFocus Объект *TextField*, *MovieClip* или *Button*, который получил фокус, или null, если ни один объект не получил фокус.

Описание

Событие *onKillFocus( )* генерируется, когда *theField* теряет фокус. Текстовое поле обладает фокусом, когда в нем находится точка ввода текста или когда в нем есть выделенные символы. Поле получает фокус в следующих случаях:

- Пользователь совершает на нем щелчок кнопкой мыши или перемещается на него с помощью клавиши Tab.
- *Selection.setFocus( )* назначает фокус программно.

Поле теряет фокус (и выполняется *onKillFocus( )*) в следующих случаях:

- Пользователь совершает щелчок кнопкой мыши за пределами поля или уходит с него с помощью клавиши Tab, при этом фокус может быть перенесен в другое место (на текстовое поле, клип или кнопку). Щелчок на кнопке или клипе не перемещает на них фокус. Если фокус перенесен на другой объект, доступ к нему внутри *onKillFocus( )* можно получить через параметр *newFocus*, в противном случае *newFocus* имеет значение null.
- *Selection.setFocus( )* назначает фокус программно другому объекту. Доступ к новому объекту внутри *onKillFocus( )* можно получить через параметр *newFocus*.

Если свойство selectable текстового поля равно false, оно не может быть помещено в фокус пользователем, но можно программно перенести в него фокус с помощью *Selection.setFocus( )*.

В следующем коде создается функция обратного вызова, реагирующая на событие *onKillFocus( )*. Запустите его в тестовом режиме и попробуйте делать щелчки на текстовом поле и вне его. Происходящие события будут сопровождаться выводом сообщений в окно Output.

```
// Создать текстовое поле для ввода данных пользователем
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
// Присвоить функцию обратного вызова onKillFocus() с помощью литерала функции
theField_txt.onKillFocus = function (newFocus) {
  trace(this + " lost focus. Focus is now " + newFocus);
};
```

В теле функции обратного вызова можно ссылаться на theField_txt с помощью ключевого слова this. Обработка события *onKillFocus( )* функцией обратного вызова прекращается с помощью оператора *delete*:

```
// Оператора () после имени функции быть не должно!
delete theField_txt.onKillFocus;
```

Событие *onKillFocus( )* позволяет запускать проверку допустимости данных, введенных в текстовое поле, когда пользователь завершает ввод текста. Либо, совместно с *onSetFocus( )*, можно выделять цветом поле, получившее фокус (см. пример для *TextField.onSetFocus( )*). Для того чтобы обрабатывать все события фокуса централизованно, а не отдельно для каждого экземпляра, применяется *Selection.onSetFocus( )*.

См. также

Selection.onSetFocus( ), *Selection.setFocus( )*, *TextField.onSetFocus( )*, *TextField*.selectable, *TextField*.tabEnabled, *TextField*.tabIndex, *TextField*.type; глава 10

TextField.onScroller(), обработчик события и событие приемника

Flash 6

вызывается при изменении любого свойства прокрутки

theField.onScroller(*scrolledField*)
listener.onScroller(*scrolledField*)

Аргументы

scrolledField Объект *TextField*, состояние прокрутки которого изменилось.

Описание

Событие *onScroller( )* генерируется при изменении любого из следующих свойств текстового поля: scroll, maxscroll, hscroll или maxhscroll. Оно извещает о том, что произошла вертикальная или горизонтальная прокрутка *theField*, либо что в *theField* появилось новое содержимоее, изменяющее максимальное значение прокрутки по вертикали или горизонтали. Обычно *onScroller( )* применяют для синхронизации интерфейса полосы прокрутки с содержимым *theField*. Так, если пользователь введет больше текста, чем помещается в однострочном поле, текст прокручивается по горизонтали и генерируется *onScroller( )*, позволяя коду изменить ширину и положение горизонтальной полосы прокрутки.

Событие *onScroller( )* может обрабатываться функцией обратного вызова, прикрепленной к текстовому полю. Следующий код регистрирует функцию обратного вызова, которая реагирует на *onScroller( )*. Запустите его в тестовом режиме, введите в поле несколько строк текста и посмотрите, как работает код.

```
// Создать текстовое поле, принимающее несколько строк данных от пользователя
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
theField_txt.multiline = true;
//  Присвоить функцию обратного вызова onScroller() с помощью литерала функции
theField_txt.onScroller = function (scrolledField) {
  trace("scroll is "     + this.scroll);
  trace("maxscroll is "  + this.maxscroll);
  trace("hscroll is "    + this.hscroll);
  trace("maxhscroll is " + this.maxhscroll);
};
```

В теле функции обратного вызова можно ссылаться на theField_txt с помощью ключевого слова this. Можно было также воспользоваться параметром *scrolledField*.

Событие *onScroller( )* можно также обрабатывать посредством приемника событий, добавленного с помощью *addListener( )* и заданного параметром *listener*. Работая

с приемником событий, следует с помощью параметра *scrolledField* определить, какое текстовое поле было прокручено. Следующий код регистрирует приемник событий, реагирующий на *onScroller( )*:

```
scrollerListener = new Object();
scrollerListener.onScroller = function (scrolledField) {
  trace("scroll is "     + scrolledField.scroll);
  trace("maxscroll is "  + scrolledField.maxscroll);
  trace("hscroll is "    + scrolledField.hscroll);
  trace("maxhscroll is " + scrolledField.maxhscroll);
};
theField_txt.addListener(scrollerListener);
```

Иметь дело с функцией обратного вызова проще, но при этом реагировать на событие может только одна функция. Объектов приемников, реагирующих на одно событие, может быть сколько угодно. Обработка события *onScroller( )* функцией обратного вызова прекращается с помощью оператора *delete*:

```
// Оператора () после имени функции быть не должно!
delete theField_txt.onScroller;
```

Прекратить обработку приемником события *onScroller( )* позволяет метод *removeListener( )*:

```
theField_txt.removeListener(scrollerListener);
```

Применение

Обратите внимание, что событие *onScroller( )* генерируется при простом щелчке кнопкой мыши в пределах поля.

Ошибки реализации

Во Flash 6, если свойствам текстового поля `scroll` или `hscroll` присваиваются значения в том же кадре, в котором происходит создание поля, *onScroller( )* не генерируется. Даже если значения присваиваются в этом кадре после создания поля, *onScroller( )* может не выполниться:

```
// Создать текстовое поле .
this.createTextField("theField_txt", 1, 0, 0, 200, 40);
theField_txt.text = "a\nb\nc\nd\ne\nf\ng\nh\ni\nj\nk\nl\nn\nm\no\np";
// Присвоить функцию обратного вызова onScroller().
theField_txt.onScroller = function (scrolledField) {
  trace("scroll is " + this.scroll);
  trace("maxscroll is " + this.maxscroll);
};
// Прокрутить поле. onScroller() не выполнится.
theField_txt.scroll = 4;
```

Чтобы заставить выполниться обработчик события *onScroller( )*, вызовите его вручную:

```
theField_txt.onScroller();
```

К сожалению, нет простого способа выполнить вручную все приемники *onScroller( )*. (Придется составить список всех приемников и выполнять каждый метод *onScroller( )* отдельно.)

Пример

Следующий код обнаруживает, когда theField_txt прокручено до конца вниз:

```
theField_txt.onScroller = function (scrolledField) {
  if (this.scroll == this.maxScroll) {
    trace("End of field reached.");
  }
}
```

См. также

TextField.addListener( ), *TextField*.hscroll, *TextField*.maxhscroll, *TextField*.maxscroll, *TextField.removeListener( )*, *TextField*.scroll; глава 10; компонент *FScrollBar* в приложении G

TextField.onSetFocus(), обработчик события

Flash 6

функция обратного вызова, вызываемая при получении полем фокуса

theField.onSetFocus(*oldFocus*)

Аргументы

oldFocus Объект *TextField*, *MovieClip* или *Button*, у которого ранее был фокус, или null, если ни у одного объекта не было фокуса.

Описание

Событие *onSetFocus( )* генерируется, когда *theField* получает фокус. Текстовое поле обладает фокусом, когда в нем находится точка ввода текста или когда в нем есть выделенные символы. Поле получает фокус в следующих случаях:

- Пользователь совершает на нем щелчок кнопкой мыши или перемещается на него с помощью клавиши Tab.
- *Selection.setFocus( )* назначает фокус программно.

Если свойство selectable текстового поля имеет значение false, оно не может быть помещено в фокус пользователем, но *Selection.setFocus( )* позволяет перенести в него фокус программно.

В следующем коде создается функция обратного вызова, которая реагирует на событие *onSetFocus( )*:

```
// Создать текстовое поле для ввода данных пользователем
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
// Присвоить функцию обратного вызова onSetFocus() с помощью литерала функции
theField_txt.onSetFocus = function (newFocus) {
  trace(this + " now has focus. Old focus was " + newFocus);
};
```

В теле функции обратного вызова можно ссылаться на theField_txt посредством ключевого слова this. Прекратить обработку события *onSetFocus( )* функцией обратного вызова можно с помощью оператора *delete*:

```
// Оператора () после имени функции быть не должно!
delete theField_txt.onSetFocus;
```

Для того чтобы обрабатывать все события фокуса централизованно, а не отдельно для каждого экземпляра, используется *Selection.onSetFocus( )*.

Пример

В следующем коде методы *onSetFocus( )* и *onKillFocus( )* применяются для выделения цветом поля при нахождении в нем фокуса, что дает пользователю удобную подсказку в интерфейсе:

```
this.createTextField("userName_txt", 1, 0, 0, 200, 20);
userName_txt.type = "input";
userName_txt.border = true;
userName_txt.background = true;
userName_txt.backgroundColor = 0xA4660B;
userName_txt.onSetFocus = function (oldFocus) {
  this.backgroundColor = 0xF5BF70;
}
userName_txt.onKillFocus = function (newFocus) {
  this.backgroundColor = 0xA4660B;
}
```

А вот как можно применить этот эффект ко всем текстовым полям фильма:

```
TextField.prototype.onSetFocus = function (oldFocus) {
  this.backgroundColor = 0xF5BF70;
}
TextField.prototype.onKillFocus = function (newFocus) {
  this.backgroundColor = 0xA4660B;
}
```

См. также

Selection.onSetFocus( ), ***Selection.setFocus( )***, *TextField*.background, *TextField*.backgroundColor, ***TextField.onKillFocus( )***, *TextField*.selectable, *TextField*.tabEnabled, *TextField*.tabIndex, *TextField*.type; глава 10

TextField._parent, свойство

Flash 6

ссылка на клип, в котором располагается текстовое поле — **чтение/запись (запись не имеет ясного результата)**

theField._parent

Описание

Свойство _parent представляет собой объектную ссылку на клип, содержащий theField. Так, если есть текстовое поле с именем title_txt в клипе с именем mainSection_mc, то title_txt._parent указывает на mainSection_mc. Свойство *TextField*._parent применяется реже, чем родственное ему *MovieClip*._parent, но иногда оказывается удобным при работе с текстовыми полями, распределенными по разным клипам.

Применение

Значением свойства _parent вполне можно сделать другой клип (или что-либо другое), но смысла в этом не много, поскольку меняется лишь ссылка, а не физическая иерархия клипа и текстового поля.

См. также

MovieClip._parent, *TextField*._name

TextField.password, свойство

Flash 6; во Flash 5 доступно только в среде разработки

маскирует символы, отображаемые в поле **чтение/запись**

theField.password

Описание

Булево свойство password определяет, должны ли символы поля отображаться нормально (false) или маскироваться на экране для соблюдения секретности (true). Если свойство имеет значение true, то все символы, включая пробелы, отображаются как звездочки (*). Например, слова «hi there» будут отображаться как «********». Это дает пользователю возможность вводить текст так, чтобы его нельзя было подсмотреть.

Если password имеет значение true, то любой вводимый текст не изменяется и может быть получен как обычно через свойство text. Свойство password не оказывает влияния на другие свойства отображения строк, такие как multiline и wordWrap; текст в маскируемом поле может переноситься и занимать несколько строк. Команды «вырезать» и «вставить» в маскированном текстовом поле не действуют.

Поля с password, имеющим значение true, не защищены, хотя и замаскированы на экране. Их содержимое остается видимым, если, например, оно передается сценарию сервера. О защите приложений Flash см. *http://www.macromedia.com/desdev/security* и *http://www.macromedia.com/desdev/mx/flash/extreme/extreme003.html*.

Пример

Следующий код создает типичные текстовые поля для ввода имени пользователя и пароля:

```
this.createTextField("nameInput_txt", 1, 0, 0, 200, 20);
nameInput_txt.type = "input";
nameInput_txt.border = true;
this.createTextField("passInput_txt", 2, 0, 30, 200, 20);
passInput_txt.type = "input";
passInput_txt.border = true;
passInput_txt.password = true;
```

См. также *TextField*.multiline, *TextField*.text, *TextField*.type, *TextField*.wordWrap

TextField.removeListener(), метод

Flash 6

отменяет отправку уведомлений о событии указанному приемнику

theField.removeListener(*listener*)

Аргументы

listener

Объект, ранее зарегистрированный в качестве приемника для *theField*.

Возвращаемое значение

Булево значение; true, если *listener* найден и удален, и false, если *listener* не был зарегистрирован в качестве приемника событий для *theField*. Возвращаемое значение полезно при отладке кода, который удаляет приемники.

Описание

Метод *removeListener( )* прекращает получение объектом *listener* уведомлений о событиях *onScroller( )* и *onChanged( )*, происходящих в *theField*. Обращаться к нему следует только после того, как *listener* зарегистрирован в качестве приемника событий с помощью *addListener( )*. Объект приемника не уничтожается и может многократно удаляться и добавляться. Когда необходимость в приемнике событий исчезает, его лучше удалить. Например, можно удалить приемник, управляющий полосой прокрутки, когда приложение переходит в выключенное состояние при открытии модального окна диалога.

Пример

```
// Создать текстовое поле для ввода данных пользователем.
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.type = "input";
// Создать приемник.
txtListener = new Object();
// Добавить в приемник функцию onScroller.
txtListener.onScroller = function (tfObject) {
  trace(tfObject + " scroll property changed.");
}
// Добавить приемник в theField_txt.
theField_txt.addListener(txtListener);
// Удалить приемник из theField_txt.
theField_txt.removeListener(txtListener);
// Этот код тоже удаляет приемник из theField_txt,
// но выводит сообщение об ошибке, если приемник не найден.
if (!theField_txt.removeListener(txtListener)) {
  trace("ERROR. No such listener on theField_txt.");
}
```

См. также

TextField.addListener( ), *TextField.onChanged( )*, *TextField.onScroller( )*; глава 10

TextField.removeTextField(), метод

Flash 6

удаляет текстовое поле на этапе исполнения

theField.removeTextField()

Описание

Если *theField* представляет собой текстовое поле, созданное на этапе исполнения с помощью *MovieClip.createTextField( )*, то метод *removeTextField( )* удаляет *theField* и соответствующее визуальное представление его на экране. Последующие ссылки на *theField* или его свойства и методы возвращают undefined. Если *theField* – текстовое поле, созданное на этапе разработки с помощью инструмента Text, *removeTextField( )* не оказывает на него действия. Чтобы удалить текстовое поле, созданное на этапе разработки, поместите его в клип и удалите весь клип с помощью *MovieClip.removeMovieClip( )*.

Пример

```
// Создать текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
```

```
// Когда-нибудь позднее...удалить это текстовое поле
theField_txt.removeTextField();
```

См. также *MovieClip.createTextField( ), MovieClip.removeMovieClip( )*

TextField.replaceSel(), метод

Flash 6

заменить выделенный текст (не нарушая форматирования)

theField.replaceSel(*newText*)

Аргументы

newText Строка новых символов, которая заменит текущее выделение.

Описание

Метод *replaceSel( )* удаляет выделенные символы в *theField* и заменяет их на *newText*. Если в *theField* нет выделенных символов, *replaceSel( )* вставляет *newText* в точку ввода *theField* или перед первым символом, если *theField* не в фокусе. Полученное значение отражается в свойстве *theField*.text.

Возможности применения *replaceSel( )* могут показаться ограниченными, но на самом деле это один из самых важных методов *TextField*, поскольку это единственный способ вставить в текстовое поле новый текст, не нарушая существующего форматирования символов. Чтобы изменить значение текстового поля с помощью *replaceSel( )*, сначала надо поместить поле в фокус, затем расположить в нужном месте точку ввода и потом вставить текст, например:

```
// Поместить поле в фокус
Selection.setFocus("theField_txt");
// Поместить точку ввода после четвертого символа
Selection.setSelection(4, 4);
// Вставить новый текст
theField_txt.replaceSel("text to insert");
```

Очевидно, такая процедура утомительна. В примере 6 реализован пользовательский метод, который берет на себя работу с выделением.

Чтобы применить новый формат к тексту, вставляемому с помощью *replaceSel( )*, надо вызвать *setNewTextFormat( )* после того, как *Selection.setSelection( )* установит точку ввода. Подробности см. в разделе «Применение» статьи, посвященной *TextField.setNewTextFormat( )*.

В некоторых случаях требуется заменить сделанное пользователем выделение, когда выполняется нажатие кнопки (например, при вставке текста в приложении типа текстового процессора). Сделать это оказывается сложнее, чем можно предположить, поскольку выделение теряется, когда фокус перемещается на кнопку во время щелчка. Чтобы не потерять выделение, мы зарегистрируем его в обработчике события появления курсора мыши (rollover) для кнопки, например:

```
replace_btn.onRollover = function () {
  // Запомним, какое поле было в фокусе
  oldFocus = Selection.getFocus();
  // Запомним диапазон выделения
  oldBegin = Selection.getBeginIndex();
  oldEnd = Selection.getEndIndex();
}
```

```
replace_btn.onRelease = function () {
  // Если последним объектом в фокусе было текстовое поле...
  if (eval(oldFocus) instanceof TextField) {
    // ...поместить его в фокус
    Selection.setFocus(oldFocus);
    // Сделать выделение и вставить новый текст
    Selection.setSelection(oldBegin, oldEnd);
    eval(oldFocus).replaceSel("text to insert");
  }
}
```

Пример

Пример С.6 добавляет метод *appendText( )* ко всем объектам *TextField*. Метод вставляет текст в конец поля с помощью *replaceSel( )*.

Пример С.6. Пользовательская функция appendText( ) с применением replaceSel( )

```
/*
 *     Метод: TextField.appendText
 *  Описание: Удобный метод для дописывания текста
 *            к текстовому полю без нарушения форматирования.
 * Параметры: theText     Дописываемый текст.
 */
TextField.prototype.appendText = function (theText) {
  // Запомнить значение scroll, потому что replaceSel() может его изменить.
  var origScroll = this.scroll;
  // Поместить поле в фокус.
  Selection.setFocus(this);
  // Расположить точку ввода за последним символом.
  Selection.setSelection(this.length, this.length);
  // Дописать текст.
  this.replaceSel(theText);
  // Восстановить исходное значение scroll.
  this.scroll = origScroll;
}
```

Упражнение: Создайте пользовательскую функцию *replaceText( )*, принимающую индексы начального и конечного символов диапазона, который требуется заменить. Для этого адаптируйте обработчик *appendText( )* примера С.6 или воспользуйтесь примером под заголовком «Описание» из данной статьи.

См. также

Selection.setFocus( ), *Selection.setSelection( )*, *TextField.setNewTextFormat( )*, *TextField*.text

TextField.restrict, свойство

Flash 6

ограничивает ввод текста группой символов **чтение/запись**

theField.restrict = *restrictedChars*;

Аргументы

restrictedChars

Не являясь аргументом в строгом смысле, *restrictedChars* представляет строку, содержащую все допустимые для поля символы.

Описание

Строковое свойство `restrict` задает символы, которые пользователь может вводить в *theField*. Если `restrict` содержит `null` (по умолчанию) или `undefined`, то допускается ввод любых символов. Если `restrict` содержит непустую строку символов, то лишь эти символы будет разрешено вводить, а попытки ввода других символов будут игнорироваться. В следующем примере разрешается ввод только букв *y* и *n*. Кроме того, `maxChars` ограничивает длину вводимого текста одним символом:

```
this.createTextField("yesnoInput_txt", 1, 0, 0, 200, 20);
yesnoInput_txt.type = "input";
yesnoInput_txt.border = true;
yesnoInput_txt.restrict = "yn";
yesnoInput_txt.maxChars = 1;
```

Обратите внимание, что действие `restrict` соответствует кодам символов, а не клавишам, поэтому буквы верхнего и нижнего регистра, такие как *Y* и *y*, следует указывать отдельно, например:

```
yesnoInput_txt.restrict = "ynYN";
```

Интересно, что если указан только один регистр, то вводимый текст преобразуется к этому регистру. Например, следующий код вызывает преобразование букв *y* и *n* в *Y* и *N*:

```
yesnoInput_txt.restrict = "YN";
```

Символы можно задавать посредством строковых литералов, как выше, или с помощью escape-последовательностей Unicode вида `\uxxxx`, где `xxxx` – шестнадцатеричная кодовая позиция разрешенного символа. Обозначения Unicode обычно употребляются для задания международных символов. Следующий код присваивает значение «ynYN» с помощью escape-последовательностей:

```
yesnoInput_txt.restrict = "\u0079\u006e\u0059\u004e";
```

Задание большого количества символов, разрешаемых в текстовом поле, может оказаться утомительной задачей, поэтому `restrict` поддерживает специальные последовательности, представляющие диапазоны и исключения. Они известны программистам, знакомым с регулярными выражениями. Знак «–» обозначает диапазон символов. Например:

```
// Разрешить для ageInput_txt ввод цифр: 0123456789
ageInput_txt.restrict = "0-9";
```

Если диапазонов несколько, они перечисляются без пробелов:

```
// Разрешить для passInput_txt ввод только букв (верхнего и нижнего регистра) и цифр
passInput_txt.restrict = "a-zA-Z0-9";
```

В диапазон включаются все символы между кодовыми позициями Unicode начального и конечного символов. Вот, например, ограничение для ввода только символов пунктуации (шестнадцатеричные кодовые позиции от 21 до 2F):

```
theField_txt.restrict = "\u0021-\u002F";
```

Обычно `restrict` присваивается строка символов, входящих в число допустимых для ввода, но предварив строку символом ^, можно задать символы, которые исключаются из числа допустимых к вводу в *theField*. Например:

```
// Включить все символы, кроме цифр
theField_txt.restrict = "^0-9";
```

```
// Включить все буквы верхнего и нижнего регистра, исключая x и X
theField_txt.restrict = "a-zA-Z^xX";
```

Для включения символов \, ^ и – используются обозначения \\\\, \\^ и \\-. Например, следующий код разрешает вводить только буквы от «a» до «z» и символы \, ^ и -:

```
theField_txt.restrict = "a-z\\^\\-\\\\";
```

Свойство restrict не ограничивает возможности сценариев по манипуляциям с текстом, оно относится только к данным, вводимым пользователем. Это означает, что свойствам text и htmlText могут быть присвоены любые значения независимо от ограничений restrict. Кроме того, restrict не распространяет свое действие на экспорт шрифтов с фильмом. Экспорт шрифтов настраивается в среде разработки с помощью кнопки Character в инспекторе свойств для текстового поля. Подробнее об экспорте шрифтов см. *TextField*.embedFonts.

Ошибки реализации

Присваивание restrict пустой строки ("") должно было бы полностью запретить ввод, но во Flash Player 6 в результате разрешается вводить любой символ, как при задании для restrict значения null. Чтобы полностью запретить ввод в поле, задайте для его свойства type значение "dynamic" и для свойства selectable значение false.

См. также

TextField.embedFonts, *TextField*.maxChars, *TextField*.password, *TextField*.selectable, *TextField*.type; «Escape-последовательности в стиле Unicode» в главе 4; приложение B

TextField._rotation, свойство

Flash 6

поворот текстового поля в градусах **чтение/запись**

theField._rotation

Описание

Свойство _rotation имеет тип числа с плавающей точкой и задает количество градусов, на которое *theField* повернуто вокруг точки регистрации его родительского фильма. Оно отражает изменения, осуществляемые на этапе разработки и программно. Числа в диапазоне от 0 до 180,0 поворачивают *theField* по часовой стрелке. Числа в диапазоне от 0 до –180,0 поворачивают его против часовой стрелки. Положительные и отрицательные значения угла поворота, в том числе запредельные, подробно описаны в статье, посвященной *MovieClip*._rotation.

Применение

Для того чтобы текстовое поле после поворота было видимо, его шрифт должен быть встроенным! См. *TextField*.embedFonts.

Пример

Следующий код вращает текстовое поле по кругу. Предполагается, что в системе при экспорте доступен шрифт Arial.

```
this.createTextField("spinner_txt", 1, 200, 200, 100, 20);
spinner_txt.embedFonts = true;
spinner_txt.text = "moock.org";
spinner_txt.setTextFormat(new TextFormat("Arial"));
spinner.rotInterval = setInterval(function () {spinner_txt._rotation+=10}, 10);
```

См. также *MovieClip*._rotation, *TextField*.embedFonts

TextField.scroll, свойство

Flash 6; соответствует свойству scroll во Flash 4 и Flash 5

текущая верхняя строка, показываемая в текстовом поле

theField.scroll

Описание

Целочисленное свойство scroll представляет собой номер (индекс) строчки, которая в данный момент отображается первой в области просмотра поля. Если текст в поле занимает больше строк, чем помещается в физической области просмотра поля, лишние строки скрыты, но остаются частью текстового поля. Верхняя строка имеет номер 1, и все строки текстового поля нумеруются в возрастающем порядке, включая те, которые не входят в просматриваемую область поля, как показано на рис. С.6.

Рис. С.6. *Индексы строк текстового поля*

Присваивание значения свойству scroll вызывает прокрутку текстового поля, делая строку с соответствующим номером верхней в области просмотра поля. Например, если присвоить свойству scroll текстового поля, показанного на рис. С.5, значение 3, поле покажет:

```
а могут –
мысли
в
```

Пример

Совместно используя свойства scroll и maxscroll, можно создавать кнопки для прокрутки текстового поля. В следующем примере текст прокручивается вниз на одну строку при каждом щелчке по кнопке:

```
scrollButton.onPress = function () {
  if (theField_txt.scroll < theField_txt.maxscroll) {
    theField_txt.scroll++;
  }
};
```

А в этом – вверх на одну строку при каждом щелчке по кнопке:

```
scrollButton.onPress = function () {
  if (theField_txt.scroll > 1) {
    theField_txt.scroll--;
  }
};
```

Пример простых кнопок прокрутки, используемых в фильмах, можно найти в сетевом хранилище кода. Во Flash 6 большинство задач прокрутки можно решить с помощью компонента FScrollBar из состава Flash UI Components, поэтому необходимость в разработке собственных графических элементов для прокрутки возникает редко. Однако у `scroll` есть другие интересные применения, такие как автопрокрутка, продемонстрированная в примере С.7 (также имеется в сетевом хранилище). В примере С.7 в класс *TextField* добавляются методы, осуществляющие прокрутку вверх, вниз и автоматическую прокрутку на заданное количество строк и с заданной скоростью. Когда прокрутка достигает самого верха или самого низа текстового поля, для этого поля вызывается пользовательская функция обратного вызова (*onTopReached( )* или *onBottomReached( )*), в которой можно, например, разместить код для показа следующей страницы в слайд-шоу.

Пример С.7. Реализация автопрокрутки для TextField

```
// Создать текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "line1\nline2\nline3\nline4\nline5\nline6\nline7\nline8";
// Добавить метод для прокрутки любого текстового поля вниз
TextField.prototype.scrollDown = function (numLines) {
  if (this.scroll + numLines < this.maxscroll) {
    this.scroll += numLines;
  } else {
    this.scroll = this.maxscroll;
    this.stopAutoScroll();
    this.onBottomReached();
  }
}
// Добавить метод для прокрутки любого текстового поля вверх
TextField.prototype.scrollUp = function (numLines) {
  if (this.scroll - numLines > 1) {
    this.scroll -= numLines;
  } else {
    this.scroll = 1;
    this.stopAutoScroll();
    this.onTopReached();
  }
}
// Добавить метод для автоматической прокрутки любого текстового поля
// Каждые speed миллисекунд прокручивается numLines строк, вверх или вниз
TextField.prototype.startAutoScroll = function (speed, numLines, dir) {
  if (typeof speed == "number" && typeof numLines == "number") {
    if (dir == "down") {
      this.autoScroller = setInterval(this, "scrollDown", speed, numLines);
    } else if (dir == "up") {
      this.autoScroller = setInterval(this, "scrollUp", speed, numLines);
    }
  }
}
// Добавить метод для прекращения автоматической прокрутки
TextField.prototype.stopAutoScroll = function () {
  clearInterval(this.autoScroller);
}
// Достигнув низа, быстро прокрутить вверх
theField_txt.onBottomReached = function () {
```

```
    this.startAutoScroll(100, 1, "up");
  }
  // Достигнув верха, начать прокрутку вниз
  theField_txt.onTopReached = function () {
    this.startAutoScroll(1000, 1, "down");
  }
  // Прокручивать 1 строку в секунду
  theField_txt.startAutoScroll(1000, 1, "down");
  // Прекратить прокрутку:
  // theField_txt.stopAutoScroll();
```

См. также

TextField.bottomScroll, *TextField*.hscroll, *TextField*.maxhscroll, *TextField*.scroll; компонент *FScrollBar* в приложении G

TextField.selectable, свойство

Flash 6

булево; управляет возможностью пользователя выделять символы **чтение/запись**

theField.selectable

Описание

Булево свойство selectable управляет возможностью пользователя выделять символы в *theField*. По умолчанию выделение возможно во всех текстовых полях (selectable имеет значение true). Чтобы создать текстовое поле без выделения только для показа, задайте для его свойства selectable значение false:

```
this.createTextField("output_txt", 1, 0, 0, 200, 20);
output_txt.selectable = false;
output_txt.text = "This is output only.";
```

Чтобы создать текстовое поле, в которое пользователь может вводить данные, задайте для его свойства type значение "input" и для свойства selectable значение true:

```
this.createTextField("input_txt", 1, 0, 0, 200, 20);
input_txt.selectable = true;
input_txt.border = true;
input_txt.type = "input";
input_txt.text = "This field can accept input.";
```

Если selectable имеет значение false, поле не может принимать данные от пользователя и быть помещено им в фокус. Однако программным образом поле всегда можно помещать в фокус и выделять с помощью объекта *Selection* независимо от значения его свойства selectable.

См. также

Объект *Selection*, *TextField*.maxChars, *TextField.onKillFocus( )*, *TextField.onSetFocus( )*, *TextField*.restrict, *TextField*.type

TextField.setNewTextFormat(), метод

Flash 6

задает форматирование по умолчанию для текстового поля

theField.setNewTextFormat(*textFormatObject*)

Аргументы

textFormatObject Экземпляр класса *TextFormat*.

Описание

Метод *setNewTextFormat( )* задает форматирование, по умолчанию заданное для *theField*. После его вызова любой текст, добавленный в *theField* с помощью *TextField*.text или *TextField.replaceSel( )* или пользователем, форматируется в соответствии со значениями свойств *textFormatObject*. Однако *setNewTextFormat( )* не оказывает воздействия на символы, уже имеющиеся в *theField*. Для измененния формата символов, находящихся в *theField*, предназначен метод *setTextFormat( )*.

Метод *setNewTextFormat( )* очень удобен для задания форматирования по умолчанию в текстовом поле, содержимое которого будет предположительно изменяться. Допустим, например, что результат пользователя должен отображаться синим цветом шрифтом Courier New высотой 10 пунктов. Когда игрок зарабатывает очки, его результат изменяется, и поле обновляется. Чтобы форматирование поля не менялось при каждом обновлении, вызовем *setNewTextFormat( )* сразу после создания поля:

```
// Создать текстовое поле.
this.createTextField("score_txt", 1, 50, 50, 100, 20);
// Назначить TextFormat по умолчанию. Это надо сделать перед присваиванием текста!
score_txt.setNewTextFormat(new TextFormat("Courier New", 10, 0x0000FF));
// Вывести результат.
score_txt.text = "0000";
// Обновить результат. Форматирование сохранено.
score_txt.text = "0020";
```

Формат, заданный для *theField* по умолчанию, можно выяснить посредством метода *getNewTextFormat( )*. Объекты *TextFormat* подробно рассмотрены в статье, посвященной классу *TextFormat*. Обратите внимание, что текст можно также форматировать с помощью HTML, как описано в статьях, посвященных свойствам *TextField*.html и *TextField*.htmlText.

Применение

Когда точка ввода помещается в текстовое поле пользователем или с помощью *Selection.setSelection( )*, формат текста по умолчанию устанавливается таким же, как у символа справа от точки ввода (или слева, если это последний символ). Чтобы помешать последующему вводу получить форматирование от уже имеющегося текста, необходимо вручную восстановить формат текста по умолчанию с помощью интервала, как показано в примере С.8. Такой способ, однако, не совсем удачен, поскольку на медленных машинах интервал может не срабатывать достаточно быстро, чтобы перехватывать все нажатия клавиш, и иногда символы получают формат имеющегося текста.

Пример С.8. Восстановление формата нового текста после выделения

```
// Создать текстовое поле для приема данных от пользователя
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
theField_txt.type = "input";
// Записать оригинальный формат нового текста в свойство
theField_txt.newTextFormat = theField_txt.getNewTextFormat();
// Определить функцию обратного вызова для событий setFocus
theField_txt.onSetFocus = function (oldFocus, newFocus) {
  // Если поле в фокусе, запустить интервал, который
  // непрерывно восстанавливает первоначальный newTextFormat
  if (newFocus == this) {
    this.restoreInterval = setInterval(this, "restoreNewTextFormat", 1);
```

```
    } else if (oldFocus == this) {
      // Stop resetting if the field is unfocused
      clearInterval(this.restoreInterval);
    }
  }
  // Создать функцию интервала для восстановления newTextFormat
  theField_txt.restoreNewTextFormat = function () {
    this.setNewTextFormat(this.newTextFormat);
  }
  // Зарегистрировать текстовое поле для получения событий setFocus
  Selection.addListener(theField_txt);
  // Поместить в поле какой-нибудь текст
  theField_txt.text = "What time is it?"
  // Создать объект TextFormat с шрифтом "Arial"
  sansFormat = new TextFormat("Arial");
  // Применить этот формат к слову 'What', символы с 0 по 3
  theField_txt.setTextFormat(0, 4, sansFormat);
```

Если текст вставляется с помощью метода *replaceSel( )*, то установить формат нового текста легко – достаточно вызвать *setNewTextFormat( )* после обращения к *Selection.setSelection( )*, как показано в примере C.9.

Пример C.9. Задание формата нового текста перед заменой выделения

```
// Поместить фокус в поле, которое нужно обновить
Selection.setFocus(theField_txt);
// Поместить точку ввода после первого символа
Selection.setSelection(1,1);
// Теперь задать newTextFormat
theField_txt.setNewTextFormat(theField_txt.newTextFormat);
// Перейти к вызову replaceSel()
theField_txt.replaceSel("this is the new text");
```

См. также

Класс *Selection*, *TextField.getTextFormat( )*, *TextField.getNewTextFormat( )*, *TextField.replaceSel( )*, *TextField.setTextFormat( )*, класс *TextFormat*

TextField.setTextFormat(), метод

Flash 6

задает форматирование для указанного диапазона символов

theField.setTextFormat(*textFormatObject*)
theField.setTextFormat(*beginIndex*, *textFormatObject*)
theField.setTextFormat(*beginIndex*, *endIndex*, *textFormatObject*)

Аргументы

textFormatObject

Экземпляр класса *TextFormat*.

beginIndex Индекс первого символа, форматирование которого будет определено объектом *textFormatObject*. Должен быть неотрицательным целым числом.

endIndex Индекс символа, следующего за последним, форматирование которого будет определено объектом *textFormatObject*. Должен быть неотрицательным целым числом.

Описание

Метод *setTextFormat( )* форматирует диапазон символов от *beginIndex* до *endIndex* в соответствии со свойствами *textFormatObject*. Свойства *textFormatObject* со значениями null или undefined не оказывают воздействия на форматируемые символы. Метод *setTextFormat( )* может вызываться с одним, двумя или тремя аргументами. При вызове *setTextFormat( )* с двумя аргументами форматирование применяется к единственному символу с индексом *beginIndex*. Например, выделим полужирным шрифтом первый символ текстового поля:

```
// Создать текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
// Поместить в поле какой-нибудь текст
theField_txt.text = "Products and Services";
// Создать объект TextFormat с полужирным начертанием
boldFormat = new TextFormat();
boldFormat.bold = true;
// Применить форматирование к букве 'P', символу 0
theField_txt.setTextFormat(0, boldFormat);
```

Когда *setTextFormat( )* вызывается с тремя аргументами, форматирование применяется к диапазону символов с *beginIndex* по *endIndex*-1. Например, следующий код выделяет курсивом слово «and»:

```
// Создать объект TextFormat с курсивным начертанием
italicFormat = new TextFormat();
italicFormat.italic = true;
// Применить форматирование к слову 'and', символы с 9 по 11 (inclusive)
theField_txt.setTextFormat(9, 12, italicFormat);
```

А если с одним, то формат применяется ко всем символам поля. Следующий код задает для всего текста в theField_txt шрифт Arial:

```
// Создать объект TextFormat, задающий шрифт "Arial"
arialFormat = new TextFormat();
arialFormat.font = "Arial";
// Применить форматирование ко всему тексту
theField_txt.setTextFormat(arialFormat);
```

Обратите внимание, что последовательное форматирование действует неразрушающим образом: изменение формата происходит только для свойств, значения которых заданы, поэтому буква «P» остается полужирной, а слово «and» остается выделенным курсивом.

Задание значения свойства text разрушает все форматирование поля. Например, если мы допишем в theField_txt.text слово «Overview», все символы будут переведены в формат для нового текста этого поля:

```
// Сбрасывает форматирование theField_txt
theField_txt.text += "Overview";
```

Для добавления текста с сохранением форматирования предназначен метод *replaceSel( )*.

Не путайте *setNewTextFormat( )* с *setTextFormat( )*; первый устанавливает формат, применяемый к тексту, добавляемому в *theField* в дальнейшем, а второй задает формат символов, уже присутствующих в *theField*. Оба метода правильные, но важен порядок, в кото-

ром они используются. Вызывайте *setTextFormat( ) после* задания значения свойства `text`; вызывайте *setNewTextFormat( ) перед* заданием значения свойства `text`.

Здесь, прежде чем вызывать *setTextFormat( )*, мы присваиваем текст, а потом применяем к нему форматирование:

```
// Создать текстовое поле
this.createTextField("title_txt", 1, 0, 0, 150, 40);
// Поместить в поле какой-нибудь текст
title_txt.text = "What's New?";
// Теперь создадим и применим объект TextFormat
title_txt.setTextFormat(new TextFormat("Arial", 14, 0xFF0000));
```

Если по ошибке применить формат раньше, чем будет присвоен текст, ничего не произойдет:

```
// Порядок действий неверен...
title_txt.setTextFormat(new TextFormat("Arial", 14, 0xFF0000));
title_txt.text = "What's New?";
```

Наоборот, имея дело с методом *setNewTextFormat( )*, необходимо сначала применить формат, а потом присваивать полю текст:

```
title_txt.setNewTextFormat(new TextFormat("Arial", 14, 0xFF0000));
title_txt.text = "What's New?";
```

Метод *setNewTextFormat( )* удобен, когда предполагаются изменения текста, потому что он применяется ко всем будущим присваиваниям значения поля:

```
// Форматирование текста "Latest News": шрифт Arial, цвет красный, высота 14 пунктов.
title_txt.text = "Latest News";
```

Подробное описание объектов *TextFormat* см. в классе *TextFormat*. Обратите внимание, что текст можно также форматировать средствами HTML, как описано в *TextField*.html и *TextField*.htmlText.

Пример

Следующий код добавляет во все объекты *TextField* метод *rainbow( )*, раскрашивающий текст поля случайным образом:

```
// Вот метод
TextField.prototype.rainbow = function () {
  for (var i = 0; i < this.length; i++) {
    var randColor = Math.floor(Math.random() * 0xFFFFFF);
    this.setTextFormat(i, new TextFormat(null, null, randColor));
  }
}
// Создадим текстовое поле, на котором его опробуем
this.createTextField("theField_txt", 1, 50, 50, 160, 20);
// Задать какой-нибудь текст
theField_txt.text = "Ooh, what pretty colors!";
// Попробуем!
theField_txt.rainbow();
```

См. также

TextField.getNewTextFormat( ), *TextField.getTextFormat( )*, *TextField*.html, *TextField*.htmlText, *TextField.replaceSel( )*, *TextField.setNewTextFormat( )*, класс *TextFormat*

TextField.tabEnabled, свойство

Flash 6

булево значение; включает и исключает текстовое поле из текущего порядка табуляции **чтение/запись**

theField.tabEnabled

Описание

Булево свойство tabEnabled управляет включением *theField* в состав *порядка табуляции* для текстовых полей, клипов и кнопок, находящихся в данный момент на экране. Порядок табуляции – это очередность, в которой объекты получают фокус при нажатии в фильме клавиши Tab. Последняя дает пользователю возможность перемещаться между текстовыми полями в форме во время ввода текста.

Однако в фильме Flash есть два разных порядка табуляции: автоматический и пользовательский. Автоматический порядок табуляции устанавливается Flash по умолчанию, когда не определен пользовательский порядок табуляции. В нем участвуют все кнопки и текстовые поля ввода, находящиеся на экране, обход которых осуществляется слева направо и сверху вниз. Текстовые поля, свойство type которых имеет значение "dynamic", в этот порядок не входят. Пользовательский порядок табуляции определяется ActionScript и может содержать произвольную последовательность объектов *Button*, *MovieClip* и *TextField*. Пользовательский порядок табуляции определяется значением свойства tabIndex для каждой кнопки, клипа или текстового поля, которые должны быть в него включены. Если хотя бы для одного объекта, находящегося в данный момент на экране, определено свойство tabIndex, то действует пользовательский порядок табуляции.

Для текстовых полей свойство tabEnabled имеет два применения:

- Оно может исключить текстовое поле с определенным tabIndex из пользовательского порядка табуляции.
- Оно может исключить текстовое поле из автоматического порядка табуляции.

Обе ситуации встречаются не слишком часто, поскольку обычно текстовые поля должны быть доступны для ввода с клавиатуры. Временное отключение текстового поля ввода достигается заданием для его свойства type значения "dynamic", что также удаляет его из порядка обхода при табуляции.

Обратите внимание, что поведение tabEnabled зависит от значения свойства tabIndex. Влияние возможных комбинаций tabEnabled и tabIndex на текстовые поля ввода показано в табл. C.24.

Таблица C.24. Зависимость между свойствами tabEnabled и tabIndex для текстовых полей ввода

tabEnabled	tabIndex	TextField включается в пользовательский порядок табуляции	TextField включается в автоматический порядок табуляции
true или undefined	undefined	нет	да
true или undefined	положительное целое	да	N/A
false	undefined	нет	нет
false	положительное целое	нет	N/A

Свойство клипа tabChildren оказывает воздействие на содержащиеся в нем текстовые поля, клипы и кнопки. Чтобы исключить из табуляции группу текстовых полей, поместите их в клип и задайте для свойства клипа tabChildren значение false.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

Пример

Следующий код создает два текстовых поля и исключает второе из автоматического порядка табуляции:

```
// Создать поля
this.createTextField("field1_txt", 1, 40, 70, 100, 20);
field1_txt.border = true;
field1_txt.type = "input";
this.createTextField("field2_txt", 2, 150, 70, 100, 20);
field2_txt.border = true;
field2_txt.type = "input";
field2_txt.tabEnabled = false;
```

См. также

Button.tabEnabled, *MovieClip*.tabChildren, *MovieClip*.tabEnabled, *TextField*.tabIndex, *TextField*.type

TextField.tabIndex, свойство

Flash 6

задает индекс текстового поля в пользовательском порядке табуляции **чтение/запись**

theField.tabIndex

Описание

Целочисленное свойство tabIndex задает положение *theField* в пользовательском порядке табуляции. Если нет текстовых полей, клипов или кнопок с определенным свойством tabIndex, Flash определяет порядок табуляции автоматически и при нажатии клавиши Tab перемещает фокус между кнопками и текстовыми полями ввода в порядке слева направо и сверху вниз. С помощью свойства tabIndex мы определяем для клавиши Tab наш собственный порядок, отменяя автоматический. Например, пользовательский порядок табуляции может заменить перемещение по строкам в таблице полей ввода на перемещение по колонкам или вообще запретить доступ с клавиатуры к текстовому полю, клипу или кнопке.

Для того чтобы создать пользовательский порядок табуляции, требуется назначить очередность доступа к объекту при табуляции в виде целого числа tabIndex. При выводе каждого кадра ActionScript динамически создает порядок табуляции, исходя из приоритетов, установленных для всех объектов, показываемых в данный момент. Движение направлено от самого маленького к самому большому числу, поэтому при нажатии клавиши Tab сначала фокусируется объект с tabIndex, равным 1, затем с tabIndex, равным 2, и т. д. Объекты, свойство _visible которых равно false, не участвуют в порядке табуляции.

Создадим четыре текстовых поля в сетке два на два; затем зададим свой порядок в направлении справа налево и вверх (противоположный автоматическому):

```
// Создадим поля
this.createTextField("row1col1_txt", 1, 40, 70, 100, 20);
row1col1_txt.border = true;
row1col1_txt.type = "input";
this.createTextField("row1col2_txt", 2, 150, 70, 100, 20);
row1col2_txt.border = true;
row1col2_txt.type = "input";
this.createTextField("row2col1_txt", 3, 40, 100, 100, 20);
row2col1_txt.border = true;
row2col1_txt.type = "input";
this.createTextField("row2col2_txt", 4, 150, 100, 100, 20);
row2col2_txt.border = true;
row2col2_txt.type = "input";
// Назначим пользовательский порядок табуляции
row2col2_txt.tabIndex = 1;
row2col1_txt.tabIndex = 2;
row1col2_txt.tabIndex = 3;
row1col1_txt.tabIndex = 4;
```

Если есть хотя бы один объект, у которого определено свойство `tabIndex`, то ни к каким объектам, свойство `tabIndex` которых не задано, нельзя обратиться с помощью клавиши Tab. Обратите внимание, что `tabIndex` действует только на текстовые поля для ввода, имеющие тип `"input"`. По умолчанию текстовые поля для ввода подключены к табуляции, поэтому устанавливать свойство `tabEnabled` (равным `false`) надо только для того, чтобы исключить экземпляр из порядка табуляции при сохранении значения его свойства `tabIndex`.

Применение

Клавиша Tab не будет надежно действовать в режиме Test Movie, если не включить Control → Disable Keyboard Shortcuts. Следует проверить действие клавиатуры при воспроизведении в автономном проигрывателе, если оно предполагается, а также в проигрывателе Flash для всех предполагаемых веб-броузеров.

См. также

Button.tabIndex, *MovieClip*.tabIndex, *TextField*.tabEnabled, *TextField*.type

TextField._target, свойство

Flash 6

путь к текстовому полю в слэш-синтаксисе **только для чтения**

theField._target

Описание

Свойство `_target` представляет путь к *theField* как строку в слэш-нотации в стиле Flash 4. Например, если `name_txt` располагается на главной временной диаграмме фильма, свойство `_target` для `name_txt` имеет значение `"/name_txt"`. Строка со слэш-нотацией преобразуется в объектную ссылку посредством функции *eval( )*. А строка, содержащая путь к текстовому полю в точечной нотации, возвращается функцией *targetPath( )*. Слэш-нотация устарела, поэтому данное свойство применяется редко и поддерживается только для обратной совместимости.

См. также

Button._target, *MovieClip*._target, *targetPath( )*

TextField.text, свойство

Flash 6

задает символы, которые должны быть показаны в поле **чтение/запись**

theField.text

Описание

Строковое свойство text задает символы, отображаемые в *theField* после того, как этому свойству присвоено значение. Извлекая свойство text, мы получаем все текстовое содержимое *theField*, включая данные, введенные пользователем. Если текст не помещается в ограничивающем прямоугольнике *theField*, то либо изменяются размеры ограничивающего прямоугольника *theField*, либо лишние знаки переносятся на следующую строку или переполняют *theField*. Поведение текстового содержимого зависит от значений свойств autoSize, multiline и wordWrap. Текст, переполняющий поле, просматривается путем горизонтальной или вертикальной прокрутки (см. hscroll и scroll).

Форматирование текста осуществляется с помощью методов *setTextFormat( )* и *setNewTextFormat( )*. Присваивание свойству text нового значения снимает все предшествующее форматирование. Для добавления в поле текста без нарушения имеющегося форматирования применяется метод *replaceSel( )*.

Ввод в текстовое поле текста в формате HTML полностью рассмотрен в описании *TextField*.htmlText. Учтите, что htmlText и text переписывают друг друга. Независимо от значения свойства html присваивание значения свойству text изменяет значение htmlText, и наоборот.

Применение

Во Flash 4 и Flash 5 текстовые поля представлялись простыми переменными, а не законченными объектами, поэтому было принято присваивать текст непосредственно переменной текстового поля, например:

```
theField_txt = "Welcome to my site.";
```

Во Flash 6 такое действие приводит к затиранию экземпляра *TextField*. Вот правильный синтаксис для Flash 6:

```
theField_txt.text = "Welcome to my site.";
```

Подробности см. в *TextField*.variable.

Пример

Свойство text — главное средство программного вывода текста на экран и главный кандидат на пример «hello world» для Flash, например:

```
// Создать текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
// Под барабанную дробь...
theField_txt.text = "hello world!";
```

Следующий код копирует значение текстового поля ввода в окно Output каждые 500 миллисекунд. Запустите его в среде разработки и введите данные в текстовое поле.

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.type = "input";
theField_txt.border = true;
setInterval(showInput, 500);
function showInput () {
```

```
    trace(theField_txt.text);
  }
```

См. также

Объект *Selection*, *TextField*.html, *TextField*.htmlText, *TextField.replaceSel( )*, *TextField*.variable

TextField.textColor, свойство

Flash 6

задает цвет всех символов текстового поля **чтение/запись**

theField.textColor

Описание

Целочисленное свойство textColor — это вспомогательное средство, позволяющее быстро и без труда задать цвет текста (не цвет фона) во всем текстовом поле. По умолчанию установлен черный цвет (то есть шестнадцатеричное 0x000000). Следующий код делает цвет текста title_txt красным:

```
this.createTextField("title_txt", 1, 0, 0, 200, 20);
title_txt.text = "The Story of Princess Gwendolyn";
title_txt.textColor = 0xFF0000;
```

Для задания цвета фона предназначено свойство backgroundColor. Цвет и формат отдельных символов устанавливаются с помощью *setTextFormat( )* или htmlText. Любое присваивание значения textColor затирает информацию о цветах символов в свойстве htmlText и объектах *TextFormat*. После задания значения textColor весь новый текст поля выводится указанным цветом. Функционально присваивание значения textColor идентично созданию объекта *TextFormat* с заданным свойством color и применению этого формата ко всему полю с помощью *setTextFormat( )*, а также применению к любому новому тексту с помощью *setNewTextFormat( )*. Значение textColor отражается в htmlText, если свойство html имеет значение true.

См. также

Класс *Color*, *TextField*.backgroundColor, *TextField*.borderColor, *TextField*.html, *TextField*.htmlText, *TextField.setNewTextFormat( )*, *TextField.setTextFormat( )*

TextField.textHeight, свойство

Flash 6

высота всего текста текстового поля в пикселах **только для чтения**

theField.textHeight

Описание

Целочисленное свойство textHeight указывает (в пикселах) высоту всего массива текста, имеющегося в *theField*. Это не размер текста, отображаемого в данный момент на экране в ограничивающем прямоугольнике *theField*, а размеры всех символов в *theField*.text, включая те, которые можно увидеть только при вертикальной прокрутке. Свойство textHeight суммирует размеры самых крупных символов в каждой строке, определяемые метриками шрифта, включая верхние и нижние выносные элементы, независимо от их наличия у конкретных символов и регистра этих символов. Свойство textHeight измеряется в пикселах, а не в количестве строк или в размерах страницы (расчет размера страницы см. в примере для *TextField*.bottomScroll).

Например, для Times New Roman высотой 15 пунктов:

- Для строки «an» textheight равно 15.
- Для строки «angel» textHeight тоже равно 15, хотя в нее входят буквы с нижним (g) и верхним (l) выступающими элементами.
- Для строки «ANGEL» textHeight тоже равно 15, хотя в ней есть прописные буквы.
- Строка «An \n angel» занимает две строчки, и для нее textHeight равно 30 (независимо от того, видны ли эти строки на экране).

(Замечание: указанные здесь метрики для шрифта Times New Roman относятся к одной конкретной системе и могут быть разными на других платформах.)

Если свойство autoSize имеет значение true, то textHeight всегда на 4 пиксела меньше _height, что оставляет небольшой зазор между верхней и нижней границами *theField* и отображаемыми символами.

Прежде чем присваивать высоту строки текстовому полю, следует выяснить ее значение с помощью метода *TextFormat.getTextExtent( )*.

Пример

Следующий упрощенный код полосы прокрутки изменяет размеры рукоятки в зависимости от величины переполнения текста в поле. При каждой прокрутке поля вычисляется отношение _height к textHeight, что позволяет определить масштаб отображения рукоятки. Расположение рукоятки определяется значениями свойств scroll и maxscroll. Для проверки этого кода создайте предварительно два прямоугольных клипа с именами scrollbg и scrollhandle.

```
// Создать многострочное текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
theField_txt.border = true;
theField_txt.text = "Line one \nLine two \nLine three \nLine four";
// Разрешить пользователю ввод нескольких строк
theField_txt.type = "input";
theField_txt.multiline = true;
// Задать размеры фона полосы прокрутки
scrollbg._width = 20;
scrollbg._height = theField_txt._height;
// Разместить полосу прокрутки
scrollbg._x = theField_txt._x + theField_txt._width;
scrollbg._y = theField_txt._y;
// Задавать размеры и положение рукоятки полосы прокрутки во время прокрутки поля
theField_txt.onScroller = function () {
  // Размеры рукоятки полосы прокрутки
  scrollhandle._width = 20;
  scrollhandle._height = theField_txt._height
                       * (theField_txt._height / theField_txt.textHeight);
  // Положение рукоятки полосы прокрутки
  scrollhandle._x = theField_txt._x + theField_txt._width;
  var trackSize = scrollbg._height - scrollhandle._height;
  scrollhandle._y = theField_txt._y
                       + trackSize * ((this.scroll-1)  / (this.maxscroll-1));
}
// Задать начальное положение рукоятки полосы прокрутки
theField_txt.onScroller();
```

См. также

TextField.autoSize, *TextField*._height, *TextField*.maxscroll, ***TextField.onScroller( )***, *TextField*.scroll, *TextField*.textWidth, *TextField*._width, ***TextFormat.getTextExtent( )***; компонет ***FScrollBar*** в приложении G

TextField.textWidth, свойство

Flash 6

ширина всего текста текстового поля в пикселах **чтение/запись**

theField.textWidth

Описание

Целочисленное свойство textWidth измеряет в пикселах ширину самой длинной строки текста в *theField*, включая те символы, которые можно увидеть только при горизонтальной прокрутке. Свойство textWidth суммирует ширину всех символов самой длинной строки, определяемую метриками шрифтов этих символов. Символы могут значительно различаться по ширине в зависимости от кегля шрифта, регистра и самого символа. Например, для шрифта Times New Roman размером 12 пунктов:

- Ширина символа «a» равна 5.
- Ширина символа «A» равна 9.
- Ширина символа «I» равна 3.
- Ширина символа «w» равна 9.

Если свойство autoSize имеет значение true, то textWidth всегда на 4 пиксела меньше _width, что оставляет небольшой зазор между самой длинной строкой текста и левым и правым краями поля.

Прежде чем присваивать ширину строки текстовому полю, следует выяснить ее значение при помощи *TextFormat. getTextExtent( )*.

Пример

Свойство textWidth позволяет создавать интерфейсы горизонтальной прокрутки или изменять масштаб графических элементов в соответствии с размерами строки текста. Следующий код создает текстовое поле и чертит над ним линию по всей ширине текста. Обратите внимание на учет зазора в расчете ширины строки:

```
// Создать текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "Welcome to my web site";
theField_txt.autoSize = true;
// Задать толщину линии
this.lineStyle(3);
// Установить начальную точку
this.moveTo(theField_txt._x + 2, theField_txt._y);
// Повести линию в конечную точку
this.lineTo(theField_txt._x + theField_txt.textWidth, theField_txt._y);
```

См. также

TextField.autoSize, *TextField*._height, *TextField*.scroll, *TextField*.maxhscroll, *TextField*.textHeight, *TextField*._width, ***TextFormat.getTextExtent( )***

TextField.type, свойство

Flash 6

определяет, может ли поле принимать данные от пользователя **чтение/запись**

theField.type

Описание

Строковое свойство type определяет режим работы поля при вводе и выводе. Допустимы лишь два значения: "input" и "dynamic" (по умолчанию). Динамические текстовые поля предназначены для отображения информации (только вывод), а текстовые поля ввода могут получать информацию от пользователя (ввод и вывод).

Текстовые поля видов Multiline и Password из Flash 5 представляются свойствами multiline и password, а не типом.

Если type имеет значение "input", пользователь может вводить в поле текст, а программа получать введенные данные через свойство text (см. пример кода в описании *TextField*.text). Если значение type равно "dynamic", то пользователь не может вводить в поле данные. Однако выделять символы можно, если явно не присвоить свойству selectable значение false. В текстовые поля ввода можно вводить текст Unicode, если Unicode поддерживается операционной системой.

Поскольку данные, вводимые пользователем в текстовое поле ввода, всегда представляют собой строки, их необходимо явно преобразовывать перед употреблением в не строковом контексте, как показано в следующем простом примере калькулятора, складывающего значения двух текстовых полей ввода:

```
// Допустим, что значение myFirstInput равно 5, а mySecondInput - 10, и мы складываем поля.
// НЕВЕРНО: "Сложение" полей выводит в myOutput "Total: 510",
// поскольку оператор + интерпретируется как конкатенация строк.
myOutput.text = "Total: " + (myFirstInput.text + mySecondInput.text);
// ПРАВИЛЬНО: Чтобы получить правильный результат, нужно сначала преобразовать поля в числа.
myOutput.text = "Total: " + (parseFloat(myFirstInput.text) + parseFloat(mySecondInput.text));
```

Пример

В следующем примере создается заголовок, который нельзя выделить и в который нельзя вводить данные:

```
this.createTextField("title_txt", 1, 0, 0, 200, 20);
title_txt.selectable = false;
title_txt.type = "dynamic";
title_txt.text = "Tea Brewing Around the World";
```

В следующем примере создается поле для ввода пользователем своего имени. Обычно к полям ввода с помощью свойства border добавляют рамку, показывая тем самым пользователю, что поле можно редактировать:

```
this.createTextField("firstName_txt", 1, 0, 0, 200, 20);
firstName_txt.type = "input";
firstName_txt.border = true;
```

См. также

parseFloat( ), *TextField*.border, *TextField*.multiline, *TextField*.password, *TextField*.restrict, *TextField*. selectable, *TextField*.text

TextField._url, свойство

Flash 6

сетевой адрес клипа, содержащего поле **только для чтения**

theField._url

Описание

Свойство _url представляет собой строку, содержащую URL (Uniform Resource Locator), и указывает адрес *theField*._parent в Интернете или на локальном диске. Свойство _url всегда содержит абсолютный, а не относительный URL. Подробности см. в описании *MovieClip*._url.

См. также *MovieClip*._url, *TextField*._parent

TextField.variable, свойство

Flash 6

связывает переменную со свойством text **чтение/запись**

theField.variable

Описание

Строковое свойство variable обеспечивает обратную совместимость, задавая обычную переменную, значение которой отражает значение *theField*.text. Это дает возможность непосредственно присваивать значение текстовому полю в стиле Flash 5. В следующем примере создается объект текстового поля с именем экземпляра firstName_txt, который связывается с переменной userName. Последующие присваивания переменной userName выводятся в поле firstName_txt и отражаются его свойством text.

```
this.createTextField("firstName_txt", 1, 0, 0, 200, 20);
firstName_txt.variable = "userName";
// Выводит на экран "Colin"
userName = "Colin";
// Получить свойство text
trace(firstName_txt.text);  // Выводит: Colin
```

И наоборот, присваивания firstName_txt.text отражаются в переменной userName:

```
firstName_txt.text = "Bruce";
trace(userName);  // Выводит: Bruce
```

Ссылку на переменную можно строить в точечной нотации, благодаря чему переменные можно использовать в объектах и на других временных диаграммах:

```
firstName_txt.variable = "_level0.someClip.someVariable";
```

При экспорте в формат Flash 5 и более ранних версий свойство variable следует задавать в среде разработки Flash MX в инспекторе свойств в поле Var:. При написании кода для Flash 6 свойства variable следует избегать, применяя вместо него свойство text.

Обратите внимание, что свойство variable хранит имя переменной, а не ее значение. Оно применяется только для установления связи между обычной переменной и *theField*.text:

```
trace(firstName_txt.variable);  // Выводит: userName
```

См. также *TextField*.htmlText, *TextField*.text

TextField._visible, свойство

Flash 6

включает и выключает отображение текстового поля **чтение/запись**

theField.`_visible`

Описание

Булево свойство `_visible` определяет, видимо в данный момент поле *theField* (`true`) или скрыто (`false`). Свойство `_visible` позволяет быстро скрыть текстовое поле. Скрытые поля можно по-прежнему контролировать из ActionScript, и в целом они ведут себя, как обычные поля, но только не отображаемые на экране. Но скрытое поле не может получать фокус и данные от пользователя, поэтому для него не генерируются события *onChanged( )*, *onKillFocus( )* и *onSetFocus( )*. Хотя событие *onScroller( )* по-прежнему генерируется в ответ на изменение содержимого поля.

Изначально свойство `_visible` имеет значение `true` для всех текстовых полей, даже для полностью прозрачных или находящихся за пределами сцены. Свойство `_visible` изменяется только тогда, когда его непосредственно модифицирует сценарий. Его следует воспринимать как программный способ, позволяющий показывать или скрывать поле, а не как совокупность параметров, определяющих видимость поля, например положения и прозрачности.

Для того чтобы скрыть поле, предпочтительнее изменить значение свойства `_visible`, а не делать его полностью прозрачным или удалять со сцены, потому что Flash Player не перерисовывает поля, свойство `_visible` которых равно `false`, что повышает скорость вывода на экран.

Пример

Этот код скрывает `theField_txt`:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.text = "You won't see this!";
theField_txt._visible = false;
```

См. также *TextField*.`_alpha`

TextField._width, свойство

Flash 6

ширина ограничивающего текстовое поле прямоугольника в пикселах **чтение/запись**

theField.`_width`

Описание

Каждое текстовое поле определяет прямоугольную область, называемую его *ограничивающим прямоугольником* (*bounding box*), внутри которой размещается текст. Свойство `_width` представляет собой неотрицательное число с плавающей точкой, задающее ширину ограничивающего прямоугольника в пикселах. Изменяя `_width`, мы изменяем ограничивающий прямоугольник поля, в результате чего становится видна большая или меньшая часть текста. При изменении `_width` не изменяется ширина содержащегося в поле текста. Для изменения размера текста служат объект `TextFormat.size` и свойство `_xscale`. Узнать ширину текста в поле можно с помощью `textWidth`. Во Flash 6 размеры ограничивающего прямоугольника можно изменять посредством свойства `autoSize` в соответствии с заполняющим поле текстом.

Изменение ширины поля в среде разработки или с помощью _xscale находит отражение в свойстве _width. Для отображения контура или закраски ограничивающего прямоугольника предназначены свойства border и background.

Пример

```
// Задать начальную ширину равной 200
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Room to grow!";
// Сделать ширину ограничивающего прямоугольника вдвое больше, чем ширина текста в поле
theField_txt._width = theField_txt.textWidth * 2;
```

См. также

TextField.autoSize, *TextField*._height, *TextField*.textWidth, *TextField*.wordWrap, *TextField*._xscale, класс ***TextFormat***

TextField.wordWrap, свойство

Flash 6

включает и выключает автоматический перенос длинных строк **чтение/запись**

theField.wordWrap

Описание

Булево свойство wordWrap включает (true) или выключает (false) автоматический перенос строк в *theField*. Оно участвует во вводе данных пользователем и в программном присваивании текста, и по умолчанию имеет значение false. Если перенос строк включен, то в длинных строках текста расставляются мягкие переносы, то есть строка, ширина которой превышает ширину ограничивающего прямоугольника theField, автоматически переходит на следующую строчку. Эта функция влияет только на отображение данных: при переносе в текст не добавляются символы возврата каретки или перевода строки. Если перенос строк отключен, текст длинных строк выходит за край ограничивающего прямоугольника *theField* и часть строки оказывается скрытой. Если wordWrap равен true, то при изменении _width длинные строки могут переноситься в разных местах (то есть заливка текстом производится заново). В текстовое поле может быть добавлен жесткий перенос с помощью escape-последовательности \n или константы новой строки.

Ошибки реализации

Во Flash 6 свойство wordWrap необходимо задать раньше, чем текстовому полю будет присвоен какой-либо текст, иначе переносы могут быть не расставлены. Например:

```
// Сначала задать перенос строк
theField_txt.wordWrap = true;
// Теперь присвоить текст
theField_txt.text = "This is a fairly long line so it might wrap when displayed.";
```

Пример

Для того чтобы создать обычное окно для ввода нескольких абзацев текста, задайте значение true для свойств wordWrap и multiline. Например:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 100);
theField_txt.type = "input";
theField_txt.border = true;
theField_txt.wordWrap = true;
```

```
theField_txt.multiline = true;
theField_txt.text = "Type input here.";
```

См. также *TextField*.multiLine, *TextField*._width

TextField._x, свойство

Flash 6

расположение текстового поля по горизонтали, в пикселах **чтение/запись**

theField._x

Описание

Свойство _x имеет тип числа с плавающей точкой и определяет расположение левого края *theField* по горизонтали. Оно измеряется в одном из двух возможных пространств координат:

- Если *theField* находится на главной временной диаграмме, то _x измеряется относительно левого края сцены. Например, при _x, равном 20, левая граница *theField* находится на 20 пикселов правее левого края сцены, а –20 означает расположение на 20 пикселов левее.
- Если *theField* находится на временной диаграмме экземпляра клипа, то _x измеряется относительно точки регистрации этого родительского экземпляра. Например, при _x, равном 20, левая граница *theField* находится на 20 пикселов правее точки регистрации его родительского экземпляра, а –20 означает расположение на 20 пикселов левее.

Свойство _x (и все горизонтальные координаты во Flash) увеличивается в правую сторону и уменьшается в левую. Дробные значения _x аппроксимируются во Flash с помощью сглаживания (размытия).

Если *theField* содержится в масштабированном или повернутом экземпляре, то его координатное пространство тоже масштабируется или поворачивается. Например, если родительский для *theField* объект увеличен до 200% и повернут на 90° по часовой стрелке, то _x будет возрастать вниз, а не вправо, а единица измерения _x будет равна 2, а не 1 пикселу.

Пример

В следующем примере свойство _x применяется для выравнивания двух полей (у них одинаковая координата x, поэтому они выравниваются по одной вертикальной линии):

```
// Создать первое поле
this.createTextField("firstName_txt", 1, 250, 0, 200, 20);
firstName_txt.border = true;
firstName_txt.type = "input";
// Создать второе поле
this.createTextField("lastName_txt", 2, 0, 0, 200, 20);
lastName_txt.border = true;
lastName_txt.type = "input";
// Выровнять второе поле
lastName_txt._x = firstName_txt._x;
lastName_txt._y = firstName_txt._y + firstName_txt._height + 10;
```

В следующем примере во все текстовые поля добавляется метод *slideTo( )*, перемещающий поле в заданное место в левом или правом направлении с заданной скоростью:

```
// Метод, запускающий интервал для скольжения
TextField.prototype.slideTo = function (xPos, pixelsPerSec) {
```

```
    var moveAmount = pixelsPerSec/20;
    if (xPos < 0) {
      moveAmount = -moveAmount;
    }
    this.slideInterval = setInterval(this, "slide", 50, moveAmount, xPos);
  }
  // Метод, перемещающий текстовое поле
  TextField.prototype.slide = function (moveAmount, xPos) {
    this._x += moveAmount;
    if (Math.abs(this._x - xPos) < 1) {
      clearInterval(this.slideInterval);
    }
    updateAfterEvent();
  }
  // Создать поле
  this.createTextField("theField_txt", 1, 0, 0, 200, 20);
  theField_txt.border = true;
  theField_txt.text = "Watch me go for a ride!";
  // Плавно переместить к координате 500 со скоростью 100 пикселов/сек.
  theField_txt.slideTo(500, 100);
```

См. также *MovieClip.globalToLocal( )*, *MovieClip.localToGlobal( )*, *TextField._y*

TextField._xmouse, свойство

Flash 6

расположение курсора мыши по горизонтали относительно текстового поля — **только для чтения**

theField.`_xmouse`

Описание

Свойство `_xmouse` имеет тип числа с плавающей точкой и указывает расположение активной точки курсора мыши по горизонтали относительно левого края *theField*. Последовательные координаты `_xmouse`, всегда измеряемые относительно главной сцены, можно узнать, прочитав вместо этого свойство `_root._xmouse`.

См. также *MovieClip*.`_xmouse`, *TextField*.`_ymouse`

TextField._xscale, свойство

Flash 6

ширина текстового поля в процентах относительно оригинала — **чтение/запись**

theField.`_xscale`

Описание

Свойство `_xscale` имеет тип числа с плавающей точкой и задает ширину ограничивающего прямоугольника *theField* относительно его исходной ширины, выраженную в процентном отношении. Исходная ширина – это либо ширина поля на этапе разработки, либо последняя заданная ширина (через *MovieClip.createTextField( )* или `_width`). Текст в поле масштабируется только в том случае, если свойству `embedFonts` присвоено значение `true`, в ином случае `_xscale` на него не влияет. Не так ведет себя свойство `_yscale`, которое всегда влияет на масштабирование текста *theField*. Обратите внимание, что масштабирование текста в одном горизонтальном направлении искажает его пропорции. Для точного управления высотой текста в пунктах следует применять объект *TextFormat*.

Если текущая ширина *theField* совпадает с первоначальной, _xscale имеет значение 100. При_xscale, равном 200, ширина *theField* удваивается по сравнению с первоначальной. А если _xscale равно 50, то ширина *theField* уменьшается вдвое.

Если значение _xscale отрицательно, а embedFonts равно true, то поле *theField* переворачивается по горизонтали, как если бы через его левый край проходило вертикальное зеркало (то есть становится зеркальным отражением самого себя), после чего с отрицательным значением обращаются, как с положительным. Для горизонтального переворота текстового поля без изменения его размеров следует задать свойство _xscale равным -_xscale.

Пример

```
this.createTextField("field1_txt", 1, 0, 0, 200, 20);
field1_txt.text = "This is twice as wide.";
field1_txt.border = true;
this.createTextField("field2_txt", 2, 0, 25, 200, 20);
field2_txt.text = "This is backwards!";
field2_txt.border = true;
// Удвоить ширину theField_txt
field1_txt._xscale *= 2;
// Создать зеркальное отражение theField_txt
field2_txt._xscale = -field2_txt_xscale;
```

См. также

MovieClip._xscale, *TextField*._width, *TextField*._yscale, класс *TextFormat*

TextField._y, свойство

Flash 6

расположение текстового поля по вертикали, в пикселах **чтение/запись**

theField._y

Описание

Свойство _y имеет тип числа с плавающей точкой и определяет расположение левого края *theField* по горизонтали. Оно измеряется в одном из двух возможных пространств координат:

- Если *theField* находится на главной временной диаграмме, то _y измеряется относительно верхнего края сцены. Например, при _y, равном 20, верхняя граница *theField* находится на 20 пикселов ниже верхнего края сцены, а -20 означает расположение на 20 пикселов выше.
- Если *theField* находится на временной диаграмме экземпляра клипа, то _y измеряется относительно точки регистрации этого родительского экземпляра. Например, при _y, равном 20, верхняя граница theField находится на 20 пикселов ниже точки регистрации его родительского экземпляра, а -20 означает расположение на 20 пикселов выше.

Свойство _y (и все горизонтальные координаты во Flash) увеличивается вниз и уменьшается вверх – в противоположность декартовым координатам. Дробные значения _x аппроксимируются во Flash с помощью сглаживания (размытия).

Если *theField* содержится в масштабированном или повернутом экземпляре, то его координатное пространство тоже масштабируется или поворачивается. Так, если родительский для *theField* объект увеличен до 200% и повернут на 90° по часовой стрел-

ке, то _y будет возрастать влево, а не вниз, а единица измерения _y будет равна 2, а не 1 пикселу.

Пример

Свойство _y часто применяется совместно с _height, чтобы размещать или выравнивать текстовые поля, расположенные одно под другим по вертикали, как показано в следующей функции *makeFields( )*:

```
// Опробуйте этот код...
makeFields(this, 6, 5, 100, 100, 200, 20);
// Создать группу текстовых полей
function makeFields (clip, numFields, bufferSpace, startX, startY, width, height) {
  // Цикл для создания запрошенного количества полей
  for (var i = 0; i < numFields; i++) {
    // Создать текущее поле
    clip.createTextField("field" + i + "_txt", i+1, startX, startY, width, height);
    // Сохранить ссылку на текущее поле
    var thisField = clip["field" + i + "_txt"];
    // Показать границу и разрешить ввод
    thisField.border = true;
    thisField.type = "input";
    // Переместить текущее поле, если оно не первое
    if (i > 0) {
      var prevField = clip["field" + (i-1) + "_txt"];
      thisField._y = prevField._y + prevField._height + bufferSpace;
    }
  }
}
```

См. также *MovieClip.globalToLocal( )*, *MovieClip.localToGlobal( )*, *TextField._x*

TextField._ymouse, свойство

Flash 6

расположение курсора мыши по вертикали относительно текстового поля — **только для чтения**

theField._ymouse

Описание

Свойство _ymouse имеет тип числа с плавающей точкой и указывает расположение активной точки курсора мыши по вертикали относительно верхнего края *theField*. Чтобы получать последовательные координаты _ymouse, всегда измеряемые относительно главной сцены, вместо этого свойства следует применять _root._ymouse.

См. также *MovieClip*._ymouse, *TextField*._xmouse

TextField._yscale, свойство

Flash 6

высота текстового поля в процентах относительно оригинала — **чтение/запись**

theField._yscale

Описание

Свойство _yscale имеет тип числа с плавающей точкой и задает высоту ограничивающего прямоугольника *theField* относительно его исходной высоты, выраженной в процентах. Исходная высота – это либо высота поля на этапе разработки, либо последняя заданная высота (через *MovieClip.createTextField( )* или _height).

Если текущая высота *theField* совпадает с первоначальной, то _yscale имеет значение 100. При _yscale, равном 200, высота *theField* удваивается по сравнению с исходной. Если же _yscale равно 50, то высота *theField* уменьшается вдвое. Изменения _yscale отражаются на значении _height.

Свойство _yscale влияет также на размер текста в поле, но он всегда масштабируется пропорционально (в одинаковых процентных отношениях по горизонтали и по вертикали), чтобы предотвратить искажение. Не так ведет себя свойство _xscale, которое влияет на масштабирование текста *theField* только в том случае, если свойству embedFonts присвоено значение true. Хотя _yscale может изменять размер текста, эти изменения не отражаются в объекте *TextFormat*, возвращаемом *getTextFormat( )*. Для точного управления высотой текста в пунктах вместо *TextField*._yscale следует применять свойство size объекта *TextFormat*.

Если значение _yscale отрицательно, а embedFonts равно true, то поле *theField* переворачивается по вертикали, как если бы через его верхний край проходило горизонтальное зеркало (то есть становится зеркальным отражением самого себя), после чего с отрицательным значением обращаются, как с положительным. Для вертикального переворота текстового поля без изменения его размеров достаточно установить свойство _yscale, равным -_yscale (отрицательному _yscale).

Пример

Следующий код увеличивает высоту theField_txt на 500%:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "This is five times as high.";
theField_txt._yscale = 500;
// Для сохранения пропорций исходного ограничивающего прямоугольника
// можно сделать _xscale равным _yscale:
theField_txt._xscale = 500;
```

См. также

MovieClip._yscale, *TextField*._height, *TextField*._xscale, класс *TextFormat*

TextFormat, класс

Flash 6

получить или задать визуальное форматирование текстового поля

Конструктор

```
new TextFormat()
new TextFormat(font, size, color, bold, italic, underline, url,
               target, align, leftMargin, rightMargin, indent, leading)
```

Аргументы

font Необязательная строка, задающая начальное значение свойства font.

size Необязательное целое число, задающее начальное значение свойства `size`.

color Необязательное целое число, задающее начальное значение свойства `color`.

bold Необязательная булева величина, задающая начальное значение свойства `bold`.

italic Необязательная булева величина, задающая начальное значение свойства `italic`.

underline Необязательная булева величина, задающая начальное значение свойства `underline`.

url Необязательная строка, задающая начальное значение свойства `url`.

target Необязательная строка, задающая начальное значение свойства `target`.

align Необязательная строка, задающая начальное значение свойства `align`.

leftMargin Необязательное положительное целое число, задающее начальное значение свойства `leftMargin`.

righMargin Необязательное положительное целое число, задающее начальное значение свойства `rightMargin`.

indent Необязательное положительное целое число, задающее начальное значение свойства `indent`.

leading Необязательное положительное целое число, задающее начальное значение свойства `leading`.

Свойства

align Задает тип горизонтального выравнивания абзаца.

blockIndent Задает отступ абзаца.

bold Булева величина, задающая вывод полужирных символов.

bullet Определяет, следует ли помещать маркеры перед абзацами.

color Задает цвет символов.

font Задает шрифт указанных символов.

indent Задает отступ первой строки абзаца.

italic Булева величина, задающая вывод символов курсивом.

leading Задает межстрочный интервал.

leftMargin Задает размер левого поля абзаца.

rightMargin Задает размер правого поля абзаца.

size Задает размер шрифта в пунктах.

tabStops Задает позиции горизонтальной табуляции в пикселах.

target Задает окно или фрейм для гипертекстовой ссылки.

underline Булево значение, задающее вывод символов с подчеркиванием.

url Задает гипертекстовую ссылку.

Методы

getTextExtent( )

Возвращает (в пикселах) размеры строки, вывод которой осуществляется в соответствии с параметрами объекта *TextFormat*.

Описание

Внутренне визуальное форматирование текстового поля описывается *профилями стилей*, хранимыми во Flash Player. Каждый профиль содержит такие настройки формата, как величина шрифта, его гарнитура, начертание (жирное, курсивное) и ширина полей. Каждый символ в текстовом поле ассоциирован с одним из этих внутренних профилей. Например, в текстовом поле

Please hurry up!

представлены два профиля стиля: один с полужирным начертанием, а другой без него. Символы *P*, *l*, *e*, *a*, *s* и *e* ассоциируются с полужирным профилем, а остальные – с профилем обычной жирности. При необходимости Flash вводит дополнительные внутренние профили, чтобы можно было отслеживать весь текст, отображаемый в проигрывателе. ActionScript не предоставляет прямого доступа к внутренним профилям проигрывателя, но может показать профиль любого символа в виде объекта *TextFormat*. Свойства объекта *TextFormat* управляют особенностями вывода конкретного символа или характеристиками отображения, общими для группы символов.

Объект *TextFormat* для имеющегося текстового поля можно получить с помощью метода *TextField.getTextFormat( )*. Например, создадим текстовое поле, а потом получим объект *TextFormat*, который описывает форматирование первого символа поля:

```
// Создать текстовое поле
this.createTextField("theField_txt", 1, 0, 0, 150, 40);
// Поместить в поле какой-нибудь текст
theField_txt.text = "Please hurry up!"
// Получить объект TextFormat для первого символа theField_txt (индекс 0)
firstCharFormat = theField_txt.getTextFormat(0);
// Проверить, выделен ли первый символ жирным шрифтом
trace(firstCharFormat.bold);  // Выводит: "false"
```

Использование объекта *TextFormat* тесно связано с соответствующим объектом *TextField*. Например, объект *TextFormat* существующего текстового поля может быть получен с помощью *TextField.getTextFormat( )*. При изучении последующего материала обращайте особое внимание на то, какие свойства и методы относятся к классу *TextFormat*, а какие входят в класс *TextField*. Чтобы задать формат одного или нескольких символов текстового поля, надо создать объект *TextFormat*, задать значения его свойств и применить формат с помощью метода *TextField.setTextFormat( )* (как иногда говорят, «применить объект формата текста к текстовому полю»).

Метод *setTextFormat( )* может дополнительно принимать индексы начального и конечного символов, к которым должен быть применен объект формата. Например:

```
// Создать объект TextFormat
emphasisFormat = new TextFormat();
// Задать для этого формата полужирный шрифт
emphasisFormat.bold = true;
```

```
// Применить этот формат к слову 'Please', символы с 0 по 5
theField_txt.setTextFormat(0, 6, emphasisFormat);
```

Существует также возможность задать свойства объекта *TextFormat* во время его создания (указав `null` для тех свойств, которые мы не хотим задавать):

```
emphasisFormat = new TextFormat(null, null, null, true);
```

Текст, помещаемый в поле, форматируется в соответствии со стилем профиля по умолчанию, представляемого в ActionScript отдельным объектом *TextFormat*, называемым *форматом нового текста* (*new text format*). Профиль стиля, действующий в поле по умолчанию, можно увидеть, вызвав *TextField.getNewTextFormat( )*, и модифицировать, создав объект *TextFormat*, задав его свойства и передав объект методу *TextField.setNewTextFormat( )*. Подробнее о модификации формата нового текста рассказано в описаниях *TextField.getNewTextFormat( )* и *TextField.setNewTextFormat( )*.

Описание доступных форматов абзацев и символов см. в перечнях свойств для *TextFormat*. Учтите, что каждое свойство доступно также в виде аналогичных тегов HTML, описываемых для *TextField*.htmlText и в приложении Е. Значения по умолчанию свойств объекта *TextFormat* текстового поля приведены в табл. С.25.

Таблица С.25. Значения свойств TextFormat по умолчанию

Свойство	Значение по умолчанию	Тип данных
bullet	false	булев
tabStops	пустой объект	объект
blockIndent	0	число
leading	0	число
indent	0	число
rightMargin	0	число
leftMargin	0	число
align	"left"	строка
underline	false	булев
italic	false	булев
bold	false	булев
target	""(пустая строка)	строка
url	""(пустая строка)	строка
color	0	число
size	12	число
font	"Times New Roman"	строка

Применение

При записи в свойство `text` объекта *TextField* уничтожается любое форматирование, связанное с полем. Сохранить форматирование при добавлении в поле текста позволяет метод *TextField.replaceSel( )*.

См. также

TextField.getNewTextFormat( ), *TextField.getTextFormat( )*, *TextField*.htmlText, *TextField.replaceSel( )*, *TextField.setNewTextFormat( )*, *TextField.setTextFormat( )*, класс *TextField*

TextFormat.align, свойство

Flash 6

задает тип горизонтального выравнивания абзаца **чтение/запись**

theFormat.align

Описание

Строковое свойство align определяет размещение текста слева, справа или в центре поля. Оно применяется к абзацам, которые представляют собой участки текста, ограниченные в коде разделителями строк (\n, <BR>, <P> или newline). Чтобы задать форматирование для текстового поля в целом, надо вызвать *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
// Создать объект TextFormat
format = new TextFormat();
format.align = "center";
// Задать расположение по центру для всего поля
theField_txt.setTextFormat(format);
```

Для того чтобы задать выравнивание для первого абзаца поля, примените формат к символу с индексом 0:

```
theField_txt.setTextFormat(0, format);
```

А выравнивание для какого-нибудь следующего абзаца можно задать, применив формат к символу, следующему сразу за символом перевода строки:

```
// Выровнять второй абзац по центру
theField_txt.setTextFormat(23, format);
```

Чтобы задать выравнивание для ряда абзацев, примените формат с указанием индексов символов, входящих в нужные абзацы:

```
theField_txt.setTextFormat(0, 24, format);
```

Если в абзаце есть перенос на следующую строку, заданный диапазон должен содержать первый символ последующей строки, иначе формат не будет применен к перенесенной строке.

Если в текстовом поле отключен автоматический перенос слов и текст абзаца превышает по размеру ширину поля, текст будет выровнен по левому краю независимо от значения align.

Значение align по умолчанию (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно "left" (влево). Если в текстовом поле нет текста, то значением align будет null. В только что созданном объекте *TextFormat* оно тоже будет null, поэтому выравнивание не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().align);  // Выводит: "left"
var format = new TextFormat();
trace(format.align);                                   // Выводит: "null"
```

См. также

TextField.wordWrap, *TextFormat*.blockIndent, *TextFormat*.indent, *TextFormat*.leftMargin, *TextFormat*.rightMargin

TextFormat.blockIndent, свойство

Flash 6

задает отступ абзаца **чтение/запись**

theFormat.blockIndent

Описание

Целочисленное свойство blockIndent задает величину отступа абзаца от левого края текстового поля в пикселах. Абзацы представляют собой участки текста, ограниченные в коде разделителями строк (\n, <BR>, <P> или newline). Чтобы задать форматирование для текстового поля в целом, надо вызвать *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
// Создать объект TextFormat
format = new TextFormat();
format.blockIndent = 30;
// Отступ абзацев для всего поля
theField_txt.setTextFormat(format);
```

Для того чтобы задать отступ первого абзаца в текстовом поле, примените формат к символу с индексом 0:

```
theField_txt.setTextFormat(0, format);
```

Отступ для какого-нибудь следующего абзаца можно задать, применив формат к символу, следующему сразу за символом перевода строки:

```
// Отступ для второго абзаца
theField_txt.setTextFormat(23, format);
```

Отступ для нескольких абзацев задается применением формата с указанием индексов символов, входящих в нужные абзацы:

```
theField_txt.setTextFormat(0, 24, format);
```

Если в абзаце есть перенос на следующую строку, заданный диапазон должен содержать первый символ последующей строки, иначе формат не будет применен к перенесенной строке.

Если в текстовом поле отключен автоматический перенос строк, большие значения blockIndent могут вызвать выход текста за правый край текстового поля. Если перенос строк включен, большие значения blockIndent вызовут перенос слов. Свойства blockIndent и leftMargin кумулятивны; например, если leftMargin равно 15 и blockIndent абзаца составляет 10, общий отступ абзаца составит 25 пикселов. Поэтому свойство leftMargin можно представлять как средство размещения поля , а blockIndent — как средство размещения абзаца.

Значение по умолчанию для blockIndent (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно 0. Если в текстовом поле нет текста, значением blockIndent будет null. В только что созданном объекте *TextFormat* оно тоже будет null, поэтому отступ абзаца не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().blockIndent);  // Выводит: 0
var format = new TextFormat();
trace(format.blockIndent);                                   // Выводит: "null"
```

См. также

TextField.wordWrap, *TextFormat*.align, *TextFormat*.indent, *TextFormat*.leftMargin, *TextFormat*.rightMargin

TextFormat.bold, свойство

Flash 6

булево значение; задает вывод полужирных символов **чтение/запись**

theFormat.bold

Описание

Булево свойство bold определяет, надо ли выводить символы в **полужирном** (true) или обычном (false) начертании.

Полужирное начертание символов для текстового поля в целом задается при помощи вызвова *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Press any key."
// Создать объект TextFormat
format = new TextFormat();
format.bold = true;
// Выделить полужирным шрифтом все поле
theField_txt.setTextFormat(format);
```

Чтобы задать полужирное начертание для одного символа поля, примените формат с одним аргументом индекса:

```
theField_txt.setTextFormat(1, format);     // Сделать полужирным второй символ ("r")
```

А полужирное начертание для группы символов задается путем применения формата с указанием индексов начального и конечного символов:

```
theField_txt.setTextFormat(6, 9, format);  // Сделать полужирными символы
                                           // с индексами 6, 7 и 8 ("any")
```

Значение bold по умолчанию (в объекте *TextFormat*, возвращаемом *TextField. getTextFormat( )*) для текстового поля, в котором есть текст, равно false. Если в текстовом поле нет текста, то значение bold будет равно null. В только что созданном объекте *TextFormat* оно тоже будет null, поэтому полужирное начертание не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().bold);  // Выводит: "false"
var format = new TextFormat();
trace(format.bold);                                   // Выводит: "null"
```

См. также

TextFormat.italic

TextFormat.bullet, свойство

Flash 6

определяет, надо ли помещать маркеры перед абзацами **чтение/запись**

theFormat.bullet

Описание

Булево свойство bullet позволяет создавать маркированный список (эквивалентно тегу HTML <UL>) из абзацев, находящихся в текстовом поле. Если bullet равно true, то первая строка каждого форматируемого абзаца (любого участка текста, заканчивающегося символом перевода строки) будет начинаться с маленького круглого значка и выводиться с отступом. Если в текстовом поле отключен автоматический перенос по словам, то последующие строки абзаца также будут выводиться с отступом, если они попадают в диапазон действия формата. Задать вместо круглого значка другой символ нельзя. Вложенные маркированные списки не поддерживаются.

Для того чтобы преобразовать в маркированный список все текстовое поле, вызовите *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "Item one.\nItem two.\nItem three."
// Создать объект TextFormat
format = new TextFormat();
format.bullet = true;
// Сделать все текстовое поле маркированным списком
theField_txt.setTextFormat(format);
```

Для добавления маркера к первому абзацу поля надо применить формат к символу с индексом 0:

```
theField_txt.setTextFormat(0, format);
```

Маркеры для любого следующего абзаца добавляются путем применения формата к символу, следующему сразу за символом перевода строки:

```
// Сделать маркированным второй абзац
theField_txt.setTextFormat(10, format);
```

Маркировка группы абзацев задается применением формата с указанием индексов начального и конечного символов в нужном диапазоне:

```
// Сделать маркированными только второй и третий абзацы
theField_txt.setTextFormat(10, 21, format);
```

Если в абзаце есть перенос на следующую строку, то заданный диапазон должен содержать первый символ последующей строки, иначе формат не будет применен к перенесенной строке.

Значение по умолчанию для bullet (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно false. Если в текстовом поле нет текста, значением bullet будет null. В только что созданном объекте *TextFormat* оно тоже будет null, поэтому маркер абзаца не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().bullet);  // Выводит: "false"
var format = new TextFormat();
trace(format.bullet);                                   // Выводит: "null"
```

См. также *TextField*.wordWrap, *TextFormat*.blockIndent

TextFormat.color, свойство

Flash 6

задает цвет символов **чтение/запись**

theFormat.color

Описание

Целочисленное свойство color задает цвет символов как шестнадцатеричный триплет RGB 0x*RRGGBB*, где *RR*, *GG* и *BB* — двузначные шестнадцатеричные числа, представляющие красную, зеленую и синюю составляющие. Так, следующий код устанавливает для символов красный цвет:

```
theFormat.color = 0xFF0000;
```

Подробно о триплетах RGB рассказано в описании класса *Color*.

Для того чтобы задать цвет для текстового поля в целом, надо вызвать *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Play with color."
// Создать объект TextFormat
format = new TextFormat();
format.color = 0xFF0000;
// Сделать все символы поля красными
theField_txt.setTextFormat(format);
```

Другая возможность состоит в применении вспомогательного свойства *TextField*.textColor прямо в поле, без объекта *TextFormat*:

```
theField_txt.textColor = 0xFF0000;
```

Цвет для одного символа поля можно задать, применив формат с одним аргументом индекса:

```
theField_txt.setTextFormat(0, format);       // Вывести первый символ красным цветом
```

Цвет группы символов задается путем применения формата с указанием индексов начального и конечного символов:

```
theField_txt.setTextFormat(10, 15, format);  // Вывести красным цветом символы 10,
                                             // 11, 12, 13 и 14 ("color")
```

Значение color по умолчанию (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно 0 (черный). Если в текстовом поле нет текста, то значение color будет равно null. В только что созданном объекте *TextFormat* оно тоже будет null, поэтому цвет не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().color);  // Выводит: 0
var format = new TextFormat();
trace(format.color);                                   // Выводит: "null"
```

См. также Класс *Color*, *TextField*.textColor

TextFormat.font, свойство

Flash 6

задает шрифт указанных символов **чтение/запись**

theFormat.font

Описание

Строковое свойство `font` задает имя шрифта, которым должны выводиться символы. Для текстовых полей с отключенным встраиванием шрифтов значение `font` должно точно совпасть с названием шрифта на машине, где воспроизводится фильм. Например, следующий код устанавливает для `theField_txt` шрифт Arial (под Windows):

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.embedFonts = false;
theField_txt.text = "This text is in Arial."
format = new TextFormat();
format.font = "Arial";
theField_txt.setTextFormat(format);
```

Список установленных на машине шрифтов можно получить посредством метода *TextField.getFontList( )*. Если в текстовом поле включено использование встроенных шрифтов, значением `font` могут быть только:

- Идентификатор связи символа шрифта, экспортированного с фильмом
- Название шрифта, экспортированного с полем, созданным на этапе разработки

Дополнительные сведения об экспорте шрифтов можно найти в описании *TextField*.`embedFonts`.

Для того чтобы задать шрифт для текстового поля в целом, следует вызвать *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
// Показывать все поле шрифтом Arial
theField_txt.setTextFormat(format);
```

Шрифт для одного символа поля задается применением формата с одним аргументом индекса:

```
theField_txt.setTextFormat(0, format);        // Вывести шрифтом Arial первый символ
```

Для группы символов можно задать шрифт, применив формат с указанием индексов начального и конечного символов:

```
theField_txt.setTextFormat(0, 4, format);     // Вывести шрифтом Arial слово "This"
```

Значение `font` по умолчанию (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно `"Times New Roman"`. Если в текстовом поле нет текста, то значение `font` будет равно `null`. В только что созданном объекте *TextFormat* оно тоже будет `null`, поэтому шрифт не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().font); // Выводит: "Times New Roman"
var format = new TextFormat();
trace(format.font);                                  // Выводит: "null"
```

Для полей с отключенным встраиванием шрифтов в свойстве `font` может содержаться разделяемый запятыми список шрифтов. Flash будет выводить текст поля первым найденным шрифтом из этого списка. Так, значение `"Arial,Helvetica,_sans"` (без пробелов между названиями!) заставит проигрыватель искать Arial, затем Helvetica (если Arial не будет найден), затем _sans (если не будут найдены ни Arial, ни Helvetica). Эта функция Flash Player не документирована, но она копирует выбор шрифтов для HTML в броузерах.

См. также *TextField*.`embedFonts`, *TextField.getFontList( )*

TextFormat.getTextExtent(), метод[1]

Flash 6

возвращает (в пикселах) размеры строки, отображаемой в соответствии с заданным объектом TextFormat

theFormat.getTextExtent(*text*)

Аргументы

text

Строка, подлежащая измерению в данном формате символа.

Возвращаемое значение

Объект со свойствами height и width.

Описание

Метод *getTextExtent( )* возвращает объект, свойства height и width которого сообщают предположительный размер *text*, отформатированного в соответствии с *theFormat*. Он используется не для того, чтобы измерить текст, уже присвоенный текстовому полю, а чтобы определить размеры произвольной строки в конкретном формате символа. Эти теоретические измерения выполняются при создании динамических интерфейсов. Например, строку, которая оказывается слишком длинной в некотором формате, можно переформатировать, чтобы она соответствовала имеющемуся пространству. В следующем примере текст обрезается с добавлением многоточия (...), если он не помещается в текстовом поле.

Метод *getTextExtent( )* измеряет только формат символов: он не учитывает форматы абзацев (перенос по словам, переносы строк, поля, маркеры, отступы, отступы абзацев и т. д.). Например, следующий код с отступом:

```
f = new TextFormat();
f.indent = 30;  // Добавить отступ.
trace(f.getTextExtent("test").width);
```

дает такой же результат, как этот код без отступа:

```
f = new TextFormat();
trace(f.getTextExtent("test").width);
```

Пример

Следующий код добавляет во все текстовые поля метод с именем *setEllipsisText( )*. С помощью *getTextExtent( )* этот метод проверяет ширину строки, прежде чем присвоить ее текстовому полю. Если строка слишком длинная и не помещается в поле, этот метод обрезает строку и добавляет многоточие (...):

```
/*
 *     Метод: TextField.setEllipsisText()
 *  Описание: Обрезает текст, чтобы он поместился в текстовом поле, и добавляет
 *            при необходимости многоточие
 * Параметры: theText     Текст, который надо присвоить текстовому полю
 */
TextField.prototype.setEllipsisText = function (theText) {
```

[1] Здесь есть ошибка реализации, которая может встретиться при разработке объектно-ориентированных приложений. Объект, возвращаемый функцией *getTransform( )*, не наследует никаких свойств и методов класса *Object* и вообще не имеет таких свойств, как __proto__ и constructor. Аналогичная ошибка встречается в объектах, возвращаемых функциями *Color.getTransform( )* и *Sound.getTransform( )*. – *Примеч. науч. ред.*

```
    // Получить формат для добавляемого нового текста
    var format = this.getNewTextFormat();
    // Инициализировать подстроку, добавляемую в текст
    var theSubstring = theText;
    // Получить ширину области отображения текста. Вычесть 8, чтобы учесть
    // промежуток между текстом и границей текстового поля.
    var textRegion = this._width - 8;
    // Предположим, что строка слишком велика
    var tooWide = true;
    // Проверим правильность этого предположения
    for (var i = theSubstring.length - 1; tooWide == true && i > 0; i--) {
      // Если строка слишком велика для этого поля...
      if (format.getTextExtent(theSubstring).width > textRegion) {
         // ...удалить символ, добавить многоточие и снова проверить.
         theSubstring = theText.substr(0, i) + "...";
      } else {
         // ...если строка умещается, прекратить проверку.
         tooWide = false;
      }
    }
    // Присвоить текстовому полю правильную строку
    this.text = theSubstring;
  }
  // Пример использования setEllipsisText()
  this.createTextField("theField_txt", 1, 100, 100, 150, 60);
  theField_txt.border = true;
  theField_txt.setEllipsisText("If this text is too long it will be truncated.");
```

См. также

TextField.border, *TextField*._height, *TextField*.textHeight, *TextField*.textWidth, *TextField*._width, *TextFormat*.size

TextFormat.indent, свойство

Flash 6

задает отступ первой строки абзаца **чтение/запись**

theFormat.indent

Описание

Целочисленное свойство indent задает величину отступа первой строки абзаца от левого края поля в пикселах. Абзацы представляют собой участки текста, ограниченные в коде разделителями строк (\n,
 или newline). Для того чтобы задать отступ для всех абзацев текстового поля, надо вызвать *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 60);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
// Создать объект TextFormat
format = new TextFormat();
format.indent = 15;
// Задать отступ первых строк для всего поля
theField_txt.setTextFormat(format);
```

Отступ первой строки для первого абзаца в текстовом поле задается применением формата к символу с индексом 0:

```
theField_txt.setTextFormat(0, format);
```

Для того чтобы задать отступ первой строки следующего абзаца, примените формат к символу, следующему сразу за символом перевода строки:

```
// Сделать отступ во втором абзаце:
theField_txt.setTextFormat(23, format);
```

Отступ первой строки группы абзацев задается применением формата с указанием индексов символов, входящих в нужные абзацы:

```
theField_txt.setTextFormat(0, 24, format);
```

Если в текстовом поле отключен автоматический перенос строк, то большие значения `indent` могут вызвать выход текста за правый край текстового поля. Если перенос строк включен, большие значения `indent` вызовут перенос слов.

Значение по умолчанию для `indent` (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно 0. Если в текстовом поле нет текста, то значение `indent` будет равно `null`. В только что созданном объекте *TextFormat* оно тоже будет `null`, поэтому отступ абзаца не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().indent);  // Выводит: 0
var format = new TextFormat();
trace(format.indent);                                   // Выводит: "null"
```

См. также *TextField*.wordWrap, *TextFormat*.align, *TextFormat*.blockIndent

TextFormat.italic, свойство

Flash 6

булево значение; задает вывод символов курсивом **чтение/запись**

theFormat.italic

Описание

Булево свойство `bold` определяет, надо ли выводить символы *курсивом* (`true`) или в обычном (`false`) начертании.

Курсивное начертание символов для текстового поля в целом задается посредством вызова *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "What a nice hat!"
// Создать объект TextFormat
format = new TextFormat();
format.italic = true;
// Выделить курсивом все поле
theField_txt.setTextFormat(format);
```

Для того чтобы задать начертание курсивом для одного символа поля, примените формат с одним аргументом индекса:

```
theField_txt.setTextFormat(5, format);      // Выделить курсивом шестой символ ("a").
```

Курсивное начертание для группы символов можно задать, применив формат с указанием индексов начального и конечного символов:

```
theField_txt.setTextFormat(7, 11, format);  // Выделить курсивом символы
                                            // с индексами 7, 8, 9 и 10 ("nice")
```

Значение italic по умолчанию (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно false. Если в текстовом поле нет текста, то значение italic будет равно null. В только что созданном объекте *TextFormat* оно тоже будет null, поэтому курсивное начертание не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().italic);  // Выводит: "false"
var format = new TextFormat();
trace(format.italic);                                   // Выводит: "null"
```

См. также *TextFormat*.bold

TextFormat.leading, свойство

Flash 6

задает межстрочное расстояние **чтение/запись**

theFormat.leading

Описание

Целочисленное свойство leading задает размер вертикального промежутка между строками текста. Оно применяется к абзацам, которые представляют собой участки текста, ограниченные в коде разделителями строк (\n,
 или newline). Если leading равно 0, межстрочное расстояние определяется метриками шрифта. Если в leading содержится любое положительное целое, оно задает размер в пикселах дополнительного пространства, добавляемого между строками. Отрицательные значения leading нельзя задать с помощью ActionScript, но можно задать на этапе разработки с помощью инспектора свойств для текста.

Межстрочное расстояние для текстового поля в целом задается при помощи вызова *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 50, 60);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
theField_txt.wordWrap = true;
// Создать объект TextFormat
format = new TextFormat();
format.leading = 5;
// Применить leading ко всему полю
theField_txt.setTextFormat(format);
```

Для того чтобы задать межстрочное расстояние для первого абзаца в текстовом поле, примените формат к символу с индексом 0:

```
theField_txt.setTextFormat(0, format);
```

Межстрочное расстояние для следующего абзаца можно задать, применив формат к символу, следующему сразу за символом перевода строки:

```
theField_txt.setTextFormat(23, format);
```

Межстрочное расстояние для группы абзацев задается применением формата с указанием индексов символов, входящих в нужные абзацы:

```
theField_txt.setTextFormat(0, 24, format);
```

Если в абзаце есть перенос на следующую строку, то заданный диапазон должен содержать первый символ последующей строки, иначе формат не будет применен к перенесенной строке.

Значение по умолчанию для `leading` (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно 0. Если в текстовом поле нет текста, то значение `leading` будет равно `null`. В только что созданном объекте *TextFormat* оно тоже будет равно `null`, поэтому межстрочное расстояние абзаца не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().leading);  // Выводит: 0
var format = new TextFormat();
trace(format.leading);                                   // Выводит: "null"
```

TextFormat.leftMargin, свойство

Flash 6

задает размер левого поля абзаца **чтение/запись**

theFormat.leftMargin

Описание

Целочисленное свойство `leftMargin` задает расстояние по горизонтали между левым краем текстового поля и левым краем абзаца в пикселах. Отрицательные значения `leftMargin` ActionScript преобразует в 0. Учтите, что независимо от значения `leftMargin` Flash автоматически добавляет небольшой промежуток между текстом и границей поля. Невозможно выровнять текст точно по краю текстового поля. Свойства `leftMargin` и `rightMargin` можно применять к одному абзацу одновременно.

Для того чтобы задать левое поле для текстового поля в целом, надо вызвать *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 40);
theField_txt.border = true;
theField_txt.text = "This is paragraph one.\nThis is paragraph two."
theField_txt.wordWrap = true;
// Создать объект TextFormat
format = new TextFormat();
format.leftMargin = 10;
// Задать левое поле шириной 10 пикселов для всего текстового поля
theField_txt.setTextFormat(format);
```

Левое поле первого абзаца в текстовом поле можно задать, применив формат к символу с индексом 0:

```
theField_txt.setTextFormat(0, format);
```

Для того чтобы задать левое поле следующего абзаца, примените формат к символу, следующему сразу за символом перевода строки:

```
theField_txt.setTextFormat(23, format);
```

Левое поле для группы абзацев задается применением формата с указанием индексов символов, входящих в нужные абзацы:

```
theField_txt.setTextFormat(0, 24, format);
```

Если в абзаце есть перенос на следующую строку, то заданный диапазон должен содержать первый символ последующей строки, иначе формат не будет применен к перенесенной строке.

Свойства `blockIndent` и `leftMargin` кумулятивны. Например, если `leftMargin` равно 15 и `blockIndent` абзаца составляет 10, то общий отступ абзаца составит 25 пикселов.

Поэтому свойство leftMargin можно представлять как средство размещения поля, а blockIndent – как средство размещения абзаца.

Значение по умолчанию для leftMargin (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно 0. Если в текстовом поле нет текста, то значение leftMargin будет равно null. В только что созданном объекте *TextFormat* оно тоже будет null, поэтому левое поле абзаца не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().leftMargin);  // Выводит: 0
var format = new TextFormat();
trace(format.leftMargin);                                   // Выводит: "null"
```

См. также *TextFormat*.blockIndent, *TextFormat*.rightMargin

TextFormat.rightMargin, свойство

Flash 6

задает размер правого поля абзаца **чтение/запись**

theFormat.rightMargin

Описание

Целочисленное свойство rightMargin задает (в пикселах) минимальное расстояние по горизонтали между правым краем текстового поля и правым краем абзаца. Отрицательные значения rightMargin ActionScript преобразует в 0. Учтите, что независимо от значения rightMargin Flash автоматически добавляет небольшой промежуток между текстом и границей поля. Невозможно выровнять текст точно по краю текстового поля. Свойства leftMargin и rightMargin можно применять к одному абзацу одновременно.

Пример использования этого свойства см. в описании *TextFormat*.leftMargin.

Значение по умолчанию для rightMargin (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно 0. Если в текстовом поле нет текста, то значение rightMargin будет равно null. В только что созданном объекте *TextFormat* оно тоже будет null, поэтому правое поле абзаца не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().rightMargin);  // Выводит: 0
var format = new TextFormat();
trace(format.rightMargin);                                   // Выводит: "null"
```

См. также *TextFormat*.blockIndent, *TextFormat*.leftMargin

TextFormat.size, свойство

Flash 6

задает размер шрифта в пунктах **чтение/запись**

theFormat.size

Описание

Целочисленное поле size задает размер шрифта в пунктах (1/72 дюйма). Увеличивая значение свойства size, мы увеличиваем размер отображаемого на экране текста. Текст, размер шрифта которого превосходит размеры ограничивающего прямоугольника текстового поля, усекается. Для изменения размеров ограничивающего прямоугольника текстового поля предназначены свойства *TextField*._height, *TextField*._width или *TextField*.autoSize.

Для того чтобы задать размер шрифта для текстового поля в целом, надо вызвать *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 40);
theField_txt.border = true;
theField_txt.text = "This text is quite large.";
// Создать объект TextFormat
format = new TextFormat();
format.size = 26;
// Задать для всего поля размер шрифта 26 пунктов
theField_txt.setTextFormat(format);
```

Размер шрифта для одного символа поля можно задать, применив формат с одним аргументом индекса:

```
theField_txt.setTextFormat(5, format);       // Задает размер шестого символа
                                             // ("t") равным 26 пунктам
```

Размер шрифта для группы символов задается путем применения формата с указанием индексов начального и конечного символов:

```
theField_txt.setTextFormat(19, 24, format);  // Задает размер
                                             // символов 19-23 ("large")
```

Значение `size` по умолчанию (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно `12`. Если в текстовом поле нет текста, то значение `size` будет равно `null`. В только что созданном объекте *TextFormat* оно тоже будет `null`, поэтому размер шрифта не будет устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().size);  // Выводит: 12
var format = new TextFormat();
trace(format.size);                                   // Выводит: "null"
```

Обратите внимание, что если для текстового поля включено встраивание шрифтов, текст в этом поле всегда подвергается сглаживанию. Поэтому не рекомендуется использовать шрифты с кеглем меньше 10 пунктов, поскольку в большинстве шрифтов сглаженный текст высотой меньше 10 пунктов становится неразборчивым. Во избежание сглаживания используйте шрифты устройств (то есть системные шрифты), для чего задайте для свойства *TextField*.`embedFonts` значение `false`. Шрифты устройств никогда не сглаживаются. Пиксельные шрифты, отчетливо видимые при малых размерах, можно получить на коммерческой основе с таких сайтов, как *http://www.miniml.com* и *http://www.fontsforflash.com*.

Применение

Формально размер шрифта `size` измеряется в пикселах, а не в пунктах. Однако, поскольку большинство дисплеев осуществляет вывод с разрешением 72 пиксела на дюйм, мы обычно не делаем различия между употреблением терминов «пиксел» (pixel) и «пункт» (point). Однако для устройств с плотностью вывода, отличной от 72 ppi, Flash не выполняет автоматическое преобразование координат: размер 10 всегда означает шрифт высотой 10 пикселов, даже если разрешение экрана составит 200 пикселов на дюйм.

См. также

TextField.`embedFonts`, *TextField*.`_yscale`, *TextFormat*.`font`, ***TextFormat.getTextExtent( )***

TextFormat.tabStops, свойство

Flash 6

задает позиции горизонтальной табуляции в пикселах **чтение/запись**

theFormat.tabStops

Описание

Свойство tabStops представляет собой массив позиций табуляции в абзаце, выраженных в пикселах. Как и в текстовых процессорах, эти позиции определяют выравнивание элементов, разделенных символом табуляции (то есть колонки, на которые переходит точка ввода, при нажатии клавиши Tab). В ActionScript позиции табуляции используются для ввода данных в таблицы и создания простых таблиц, как показано в следующем примере.

Каждый элемент в массиве tabStops задает расстояние от левого края текстового поля, а не от предыдущей позиции. Например, значение tabStops [50, 120] задает две позиции табуляции, одну в 50 пикселов, а другую в 120. Чтобы поместить символ табуляции в строку, нажмите клавишу Tab, воспользуйтесь escape-последовательностью "\t" или выполните *String.fromCharCode(9)*. Для удобства можно присвоить символ табуляции переменной:

```
var tab = String.fromCharCode(9);
```

Символ табуляции всегда вызывает перемещение точки ввода вперед к следующей позиции табуляции, даже если предыдущие были пропущены. Пусть, например, tabStops имеет значение [5, 100] для текстового поля, которое содержит текст:

```
"Colin" + String.fromCharCode(9) + "Moock"
```

Тогда слово «Moock» появится на 100 пикселов вправо от левой границы поля. Слово «Colin» шире 5 пикселов, поэтому первая позиция табуляции игнорируется и используется следующая. Следует помнить, что позиции табуляции задаются в пикселах, а не в символах. Символы шрифта 10 пунктов в среднем имеют ширину 8 пикселов.

Позиции табуляции для текстового поля в целом можно задать, вызвав *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 40);
theField_txt.border = true;
var tab = String.fromCharCode(9);
theField_txt.text = "One" + tab + "Two" + tab + "Three\n"
                  + "Four" + tab + "Five" + tab + "Six";
// Создать объект TextFormat
format = new TextFormat();
format.tabStops = [50, 130];
// Задать позиции табуляции
theField_txt.setTextFormat(format);
```

Для того чтобы задать позиции табуляции для первого абзаца в текстовом поле, надо применить формат к символу с индексом 0:

```
theField_txt.setTextFormat(0, format);
```

Позиции табуляции для следующего абзаца задаются применением формата к символу, следующему сразу за символом перевода строки:

```
theField_txt.setTextFormat(14, format);
```

Позиции табуляции для группы абзацев можно задать, применив формат с указанием индексов символов, входящих в нужные абзацы:

```
theField_txt.setTextFormat(0, 15, format);
```

Если в абзаце есть перенос на следующую строку, то заданный диапазон должен содержать первый символ последующей строки, иначе формат не будет применен к перенесенной строке.

По умолчанию остановка табуляции в текстовом поле происходит каждые (*4 * средняя ширина символа*) пикселов, и массив `tabStops` пуст. Если в текстовом поле нет текста, то значение `tabStops` равно `null`. В только что созданном объекте *TextFormat* оно тоже будет `null`, поэтому позиции табуляции абзаца не будут устанавливаться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().tabStops);  // Выводит: nothing
                                                          // (пустой массив)
var format = new TextFormat();
trace(format.tabStops);                                   // Выводит: "null"
```

Пример

Следующий код добавляет во все объекты *TextField* метод, который программно размещает массив текста в табличной структуре при помощи свойства `tabStops`. Обратите внимание, что в нем применяется пользовательский метод *appendText( )*, листинг которого есть в описании *TextField.replaceSel( )*. Оба метода можно загрузить из сетевого хранилища кода.

```
/*
 *     Метод: TextField.makeTable
 *  Описание: Форматирует массив текста в виде таблицы.
 * Параметры: tabStops     Массив, содержащий позиции колонок,
 *                         начиная с колонки 2, а не 1. Текст в каждой ячейке таблицы
 *                         не переносится, поэтому он не должен превышать размеров табуляции.
 *         data            Массив, который следует форматировать.
 */
TextField.prototype.makeTable = function (tabstops, data) {
  // Записать символ табуляции в переменную
  var tab = "\t";
  // Определить количество колонок
  var numCols = tabstops.length;
  // Определить количество строк
  var numRows = Math.ceil(data.length/numCols);
  // Инициализировать переменную для текста, который в итоге будет присвоен текстовому полю
  var newText = "";
  // Добавить символы табуляции между колонками и символы перевода строк после каждого ряда.
  // Для каждого ряда...
  for (var i = 0; i < numRows; i++) {
    // ...пройти по всем колонкам.
    for (var j = 0; j < numCols; j++) {
      // Поместить в текст каждый элемент массива
      newText += data[i*numCols + j];
      // Если кончились элементы в массиве, завершить цикл
      if (i*numCols + j == data.length - 1) {
        break;
      }
      // Если колонка не последняя...
      if (j < numCols - 1) {
```

```
        // ...добавить символ табуляции.
        newText += tab;
      } else {
        // ...Иначе добавить символ перевода строки.
        newText += "\n"
      }
    }
  }
  // Наконец, надо применить табуляцию с помощью объекта TextFormat. Создать объект TextFormat,
  // но запомнить старое свойство tabStops, чтобы восстановить его в конце работы. В результате
  // добавляемый впоследствии текст будет использовать первоначальные позиции табуляции поля.
  format = this.getNewTextFormat();
  var oldTabStops = format.tabStops;
  format.tabStops = tabstops;
  // Применить позиции табуляции.
  this.setNewTextFormat(format);
  // Вставить таблицу.
  this.appendText("\n" + newText + "\n");
  // Восстановить изначальные позиции табуляции.
  format.tabStops = oldTabStops;
  this.setNewTextFormat(format);
}
// Пример использования. Создадим массив, который станет таблицей.
var tableData = ["one", "two", "three", "four", "five", "six", "seven"];
// Создадим текстовое поле для размещения таблицы.
this.createTextField("theField_txt", 1, 0, 0, 250, 60);
theField_txt.border = true;
// Добавим в таблицу текст.
theField_txt.appendText("Here is the table:");
theField_txt.makeTable([50, 125, 200], tableData);
theField_txt.appendText("Isn't it pretty?");
```

См. также

String.fromCharCode( ), ***TextField.replaceSel( )***, *TextFormat*.blockIndent, *TextFormat*.leftMargin

TextFormat.target, свойство

Flash 6

задает окно или фрейм для гиперссылки **чтение/запись**

theFormat.target

Описание

Строковое свойство target задает имя окна броузера или фрейма, в которые надо загрузить документ с адреса *theFormat*.url. Это может быть специальное имя или одно из предустановленных: "_blank", "_parent", "_self" или "_top". Если свойство theFormat.url не задано, target не оказывает никакого эффекта. Обратите особое внимание, что url, а следовательно и target, действуют только тогда, когда свойство текстового поля *TextField*.html равно true.

Свойство target следует применять совместно со свойством url. Свойство target для текстового поля в целом можно задать, вызвав *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 40);
theField_txt.border = true;
```

```
theField_txt.text = "Visit www.moock.org.";
theField_txt.html = true;   // ОБЯЗАТЕЛЬНО!!
// Создать объект TextFormat
format = new TextFormat();
format.url = "http://www.moock.org/";
format.target = "_blank";  // Открыть ссылку в новом окне
format.underline = true;   // Добавить подчеркивание вручную
// Применить гипертекстовую ссылку
theField_txt.setTextFormat(format);
```

Для того чтобы добавить ссылку в один символ поля, надо применить формат с одним аргументом индекса:

```
theField_txt.setTextFormat(0, format); // Ссылка в первом символе, "V"
```

Ссылку для группы символов можно задать, применив формат с указанием индексов начального и конечного символов:

```
// Сделать ссылкой текст: "www.moock.org"
theField_txt.setTextFormat(6, theField_txt.length, format);
```

Значение `target` по умолчанию (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля, в котором есть текст, равно пустой строке (`""`). Если в текстовом поле нет текста, то значение `target` равно `null`. В только что созданном объекте *TextFormat* оно тоже будет `null`, поэтому ссылка не будет создаваться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().target);  // Ничего не выводит
var format = new TextFormat();
trace(format.target);                                   // Выводит: "null"
```

См. также *TextField*.`html`, *TextFormat*.`url`

TextFormat.underline, свойство

Flash 6

булево значение; задает вывод символов с подчеркиванием **чтение/запись**

theFormat.`underline`

Описание

Булево свойство `underline` определяет, с подчеркиванием (`true`) или без подчеркивания (`false`) должны отображаться символы.

Подчеркивание текста всего текстового поля можно задать, вызвав *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 200, 20);
theField_txt.border = true;
theField_txt.text = "Please enter your name."
// Создать объект TextFormat
format = new TextFormat();
format.underline = true;
// Подчеркнуть все поле
theField_txt.setTextFormat(format);
```

Для того чтобы подчеркнуть один символ поля, надо применить формат с одним аргументом индекса:

```
theField_txt.setTextFormat(0, format); // Подчеркнуть первый символ ("P")
```

Подчеркивание для группы символов можно задать, применив формат с указанием индексов начального и конечного символов:

```
theField_txt.setTextFormat(0, 6, format);  // Подчеркнуть символы с индексами
                                           // 0, 1, 2, 3, 4 и 5 ("Please")
```

Значение underline по умолчанию (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat( )*) для текстового поля равно false. Однако в только что созданном объекте *TextFormat* оно равно null, поэтому подчеркивание не будет изменяться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().underline);  // Выводит: false
var format = new TextFormat();
trace(format.underline);                                   // Выводит: "null"
```

В отличие от большинства веб-броузеров, Flash не подчеркивает автоматически гиперссылки, поэтому для выделения ссылок надо вручную использовать свойство underline, как показано в *TextFormat*.url.

См. также *TextField*.htmlText, *TextFormat*.url

TextFormat.url, свойство

Flash 6

задает гипертекстовую ссылку **чтение/запись**

theFormat.url

Описание

Строковое свойство url задает гипертекстовую ссылку для некоторого участка текста.

Свойство *TextFormat*.url не оказывает никакого действия, если для свойства текстового поля *TextField*.html не задано значение true.

Для того чтобы загрузить документ в именованное окно или фрейм, укажите имя окна или фрейма в качестве значения *theFormat*.target. Flash не подчеркивает гиперссылки и не выделяет их цветом, это надо делать вручную с помощью свойств underline и color. О том, как задавать ссылки, рассказано в описании глобальной функции *getURL( )* и в приложении E.

Ссылку для текстового поля в целом можно создать, вызвав *TextField.setTextFormat( )* без аргументов, содержащих индексы:

```
this.createTextField("theField_txt", 1, 0, 0, 250, 40);
theField_txt.border = true;
theField_txt.text = "Visit www.moock.org.";
theField_txt.html = true;  // ОБЯЗАТЕЛЬНО!!
// Создать объект TextFormat
format = new TextFormat();
format.url = "http://www.moock.org/";
format.underline = true;  // Добавить подчеркивание вручную
// Применить гипертекстовую ссылку
theField_txt.setTextFormat(format);
```

Для того чтобы добавить ссылку в один символ поля, примените формат с одним аргументом индекса:

```
theField_txt.setTextFormat(0, format); // Ссылка в первом символе, "V"
```

Чтобы задать ссылку для группы символов, примените формат с указанием индексов начального и конечного символов:

```
// Преобразовать в ссылку текст: "www.moock.org"
theField_txt.setTextFormat(6, theField_txt.length, format);
```

Значение `url` по умолчанию (в объекте *TextFormat*, возвращаемом *TextField.getTextFormat()*) для текстового поля, в котором есть текст, равно пустой строке (нет ссылки). Если в текстовом поле нет текста, то значение `url` равно `null`. В только что созданном объекте *TextFormat* оно тоже будет `null`, поэтому `url` не будет изменяться при применении формата к текстовому полю:

```
trace(anyUnformattedField_txt.getTextFormat().url);  // Ничего не выводит
var format = new TextFormat();
trace(format.size);                                  // Выводит: "null"
```

См. также

getURL(), *TextField*.`html`, *TextFormat*.`color`, *TextFormat*.`target`, *TextFormat*.`underline`; приложение E

toggleHighQuality(), глобальная функция

Flash 2; устарела во Flash 5

изменяет качество вывода проигрывателя

`toggleHighQuality()`

Описание

Функция *toggleHighQuality()* осуществляет переключение между высоким и низким качеством вывода. При установке высокого качества Flash Player выводит линии со сглаженными (antialiased) краями. При установке низкого качества линии выводятся с несглаженными (зубчатыми) краями. Функция *toggleHighQuality()* не принимает аргументов, а лишь переключает две возможные установки – «High» и «Low». Это не очень удобно, поскольку качество не устанавливается явно в известное состояние и не допускается иных установок качества, кроме двух заданных.

Во Flash 5 *toggleHighQuality()* устарела в связи с появлением глобального свойства `_quality`, которое поддерживает градации качества вывода Low, AutoLow, Medium, High, AutoHigh и Best.

См. также

`_highquality`, `_quality`

trace(), глобальная функция

Flash 4

выводит значение в окно Output

`trace(`*`value`*`)`

Аргументы

value Выражение, которое должно быть разрешено и затем выведено в окно Output. Если разрешенное значение *value* не является строкой, то оно преобразуется в строку перед отправкой в окно Output согласно правилам, описанным в табл. 3.2 главы 3.

Описание

Функция *trace()* представляет собой инструмент отладки, используемый только в отладочном режиме Test Movie среды разработки Flash. Вообще-то незаметная,

trace( ) фактически является одним из базовых компонентов программирования на ActionScript: она позволяет проверить значение переменной или выражения в любой точке во время воспроизведения фильма. Окно Output недоступно при тестировании в броузере или автономном проигрывателе, поэтому для вывода результатов следует использовать экранное текстовое поле, например:

```
this.createTextField("debug_txt", 99999, 50, 50, 300, 200);
debug_txt.border = true;
debug_txt.text += "Your message here" + newline;
```

Альтернативой может быть вызов в фильме Flash окна предупреждения JavaScript, как описывается на странице

http://www.moock.org/asdg/technotes/traceInBrowser

Применение

Многочисленные команды *trace( )* могут замедлить воспроизведение фильма. Чтобы показывать фильм с его настоящей скоростью, отключите трассировку с помощью опции Omit Trace Actions в File → Publish Settings → Flash.

Пример

```
trace(firstName);              // Вывести значение firstName
trace(myClip);                 // Вывести путь к myClip
trace(myClip._x)               // Вывести x-координату myClip
trace("hello" + " there");     // Разрешить, а затем вывести выражение
```

См. также *Object.watch( )*; глава 17

typeof, оператор

Flash 5

определяет тип данных для выражения или идентификатора

typeof *operand*;

Операнды

operand

Выражение или идентификатор, тип данных которых требуется определить.

Возвращаемое значение

Строка с типом данных, полученным для *operand*, согласно табл. 3.4 главы 3.

Описание

Оператор *typeof* возвращает тип данных *operand*. Например, в случае применения с идентификатором клипа он возвращает "movieclip".

См. также

instanceof; «Оператор typeof» в главе 5; «Определение типа элемента данных» и табл. 3.4 в главе 3

unescape(), глобальная функция

Flash 5

раскодирует строку с escape-последовательностями

unescape(*stringExpression*)

Аргументы

stringExpression

Строка (или выражение, разрешаемое в строку), ранее закодированное с помощью *escape( )*.

Возвращаемое значение

Новая строка, представляющая собой раскодированную версию `stringExpression`.

Описание

Функция *unescape( )* возвращает новую строку, основанную на `stringExpression`. Новая строка содержит эквивалентные символы для каждой шестнадцатеричной escape-последовательности в `stringExpression`, как описано в *escape( )*. Функции *escape( )* и *unescape( )* используются при кодировании и раскодировании строк для надежной передачи по сети. Однако использовать *unescape( )* вручную требуется редко, поскольку Flash автоматически преобразует текст в кодировке URL при импорте его через *loadVariables( )* или класс *LoadVars*.

Пример

```
var msg = "hello!";
// Задать для msgCoded значение "hello%21"
msgCoded = escape(msg);
// Задать для msgDecoded значение "hello!"
var msgDecoded = unescape(msgCoded);
```

См. также *escape( )*; приложение B

unloadMovie(), глобальная функция

Flash 4

удаляет фильм или клип из проигрывателя

`unloadMovie(target)`

Аргументы

target Строка или ссылка, указывающая путь к клипу или уровню документа, которую нужно удалить из проигрывателя (в строковом контексте ссылки преобразуются в пути).

Описание

Функция *unloadMovie( )* чаще всего применяется для удаления фильмов из уровней документа в проигрывателе. Например, если фильм загружен на уровень 1 проигрывателя, удалить его из проигрывателя можно так:

```
unloadMovie("_level1");
```

Функция *unloadMovie( )* может применяться для выгрузки экземпляров клипов, и тогда она удаляет содержимое экземпляра, а не сам экземпляр. Экземпляр остается на сцене в виде пустой оболочки, в которую можно загружать файлы *.swf* или *.jpg*. Поэтому один клип может выступать как контейнер динамического содержания, управляемого последовательными вызовами *loadMovie( )* и *unloadMovie( )*. Обратите внимание на разницу между *unloadMovie( )* и *removeMovieClip( )*: последний удаляет экземпляр клипа из памяти, а первый – содержимое из экземпляра клипа, но оставляет сам экземпляр действующим.

Метод *unloadMovie( )* выполняется асинхронно: фактическая операция выгрузки начинается после того, как выполнятся все сценарии текущего кадра.

Применение

Функция *unloadMovie( )* соответствует действию Unload Movie с целевым маршрутом во Flash 4.

См. также

loadMovie( ), *MovieClip.removeMovieClip( )*, *MovieClip.unloadMovie( )*, *removeMovieClip( )*, *unloadMovieNum( )*; «Удаление экземпляров клипов и главных фильмов» и «Проблемы, связанные с перекрытием методов и глобальных функций», глава 13

unloadMovieNum(), глобальная функция

Flash 3

удаляет фильм из уровня документа

```
unloadMovieNum(level)
```

Аргументы

level Неотрицательное целое число или выражение с неотрицательным целым значением, указывающее уровень документа, который должен быть выгружен.

Описание

Функция *unloadMovieNum( )* почти идентична *unloadMovie( ),* за исключением того, что требует задавать `level` цели как число, а не как строку. Это означает, что *unloadMovieNum( )* может удалять фильмы только на уровнях документа, а не из клипов. Обычно она применяется для того, чтобы динамически назначить уровень удаляемого фильма, например:

```
var x = 3;
unloadMovieNum(x);
```

Тот же результат может быть получен с помощью конкатенации строк и обычной функции *unloadMovie( )*:

```
unloadMovie("_level" + x);
```

Применение

Функция *unloadMovieNum( )* соответствует действию Unload Movie во Flash 3 и Flash 4, которые действовали только с нумерованными уровнями.

См. также

loadMovieNum( ), *MovieClip.removeMovieClip( )*, *Movieclip.unloadMovie( )*, *removeMovieClip( )*, *unloadMovie( )*; «Удаление экземпляров клипов и главных фильмов», глава 13

updateAfterEvent(), глобальная функция

Flash 5

перерисовывает содержимое сцены в промежутке между кадрами

```
updateAfterEvent()
```

Описание

Когда функция выполняется во Flash Player в промежутке между кадрами, любые изменения на экране не показываются до вывода следующего кадра. Однако внутри следующих функций можно принудительно обновить экран путем вызова *updateAfterEvent( )*:

- Функция обратного вызова *setInterval( )*
- *MovieClip.onMouseMove( )*
- *MovieClip.onMouseDown( )*
- *MovieClip.onMouseUp( )*
- *MovieClip.onKeyDown( )*
- *MovieClip.onKeyUp( )*

Следует иметь в виду, что *updateAfterEvent( )* – это не произвольное средство обновления экрана: вне функции *setInterval( )* и обработчиков перечисленных выше событий она не оказывает никакого действия. Во Flash 6 даже обработчики событий объектов *Mouse* и *Key* не поддерживают *updateAfterEvent( )*, однако предполагается включение в них поддержки в будущих версиях Player.

Пример

Следующий сценарий вызывает перемещение клипа `pointer_mc` за указателем мыши и с помощью *updateAfterEvent( )* обновляет экран при каждом перемещении указателя (что может происходить чаще, чем смена кадров):

```
pointer_mc.onMouseMove = function () {
  this._x = _root._xmouse;
  this._y = _root._ymouse;
  updateAfterEvent();
}
```

См. также «Обновление экрана с помощью updateAfterEvent()», глава 10

$version, «глобальное» свойство

Flash 4 Build 11; устарело во Flash 5

Версия Flash Player **только для чтения**

`_root.$version`

Описание

Свойство `$version` содержит ту же строку, которая возвращается глобальной функцией *getVersion( )* и свойством `System.capabilities.version` (операционную систему и информацию о версии проигрывателя). В статье настоящего справочника, посвященной `capabilities.version`, подробно рассказано о том, какова структура информации о версии и о том, как определить версию из броузера перед загрузкой Flash Player.

Свойство `$version` было введено во Flash Player 4, когда тот находился в зените своего жизненного цикла, а во Flash 5 его сменила функция *getVersion( )*, которая, в свою очередь, была заменена свойством `System.capabilities.version` во Flash 6. Строго говоря, это не глобальное свойство, а свойство главной временной диаграммы: из временной диаграммы любого другого клипа оно должно вызываться как `_root.$version`.

Применение

Свойство `$version` следует применять только при экспорте в формате Flash 4, функцию *getVersion( )* – при экспорте в формате Flash 5 и свойство `System.capabilities.version` – при экспорте в формате Flash 6.

См. также `capabilities.version`, *getVersion( )*

XML, класс

Flash 5; во Flash 6 улучшена производительность и добавлены методы getBytesLoaded() и getBytesTotal()

поддержка данных XML, основанная на DOM

Конструктор

`new XML()`
`new XML(`*source*`)`

Аргументы

source Необязательная строка, содержащая корректные данные XML (или HTML), которые должны быть преобразованы в иерархию объектов *XML*.

Свойства

attributes Объект, в свойствах которого хранятся атрибуты элемента.

childNodes Массив ссылок на дочерние узлы.

contentType Тип содержимого MIME, который должен передаваться серверам.

docTypeDecl Тег `DOCTYPE` документа.

firstChild Ссылка на первого потомка узла.

ignoreWhite Определяет, следует ли игнорировать при анализе XML узлы пробельных символов.

lastChild Ссылка на последнего потомка узла.

loaded Статус операции *load( )* или *sendAndLoad( )*.

nextSibling Ссылка на узел, следующий за данным на текущем уровне иерархии объектов.

nodeName Имя текущего узла.

nodeType Тип текущего узла, 1 (узел элемента) или 3 (узел текста).

nodeValue Значение текущего узла.

parentNode Ссылка на ближайшего предка узла.

previousSibling Ссылка на узел, предшествующий данному на текущем уровне иерархии объектов.

status Код ошибки, описывающий результат разбора источника XML в иерархию объектов.

xmlDecl Тег объявления документа XML.

Методы

appendChild( ) Добавить к узлу новый дочерний узел.

cloneNode( ) Создать копию узла.

createElement( )

Создать новый узел элемента.

createTextNode( )

Создать новый текстовый узел.

getBytesLoaded( )

Количество загруженных байтов внешнего файла XML.

getBytesTotal( )

Физический дисковый размер внешнего файла XML в байтах.

hasChildNodes( )

Проверить, есть ли у узла потомки.

insertBefore( ) Вставить узел перед данным узлом на том же уровне.

load( ) Импортировать исходный код XML из внешнего документа.

parseXML( ) Анализировать строку исходного кода XML.

removeNode( ) Удалить узел из иерархии объектов.

send( ) Отправить исходный код XML внешнему сценарию или приложению.

sendAndLoad( ) Отправить исходный код XML внешнему сценарию или приложению и получить обратно исходный код XML.

toString( ) Преобразовать объект *XML* в строку.

Обработчики событий

onData( ) Обработчик, выполняемый при завершении загрузки внешнего источника XML.

onLoad( ) Обработчик, выполняемый, когда внешние данные XML разобраны в иерархию объектов.

Описание

Объекты класса *XML* применяются для обработки содержимого документа XML в объектно-ориентированном стиле и отправки данных в формате XML во Flash и из него. Свойства и методы объекта *XML* позволяют создавать документ со структурой XML (или читать уже существующий) и эффективно читать, изменять или удалять информацию в этом документе.

Класс *XML* в действительности представляет собой подкласс внутреннего надкласса *XMLnode*. Различия между этими двумя классами описаны в статье о классе *XMLnode*. Там же в табл. С.28 перечислены свойства и методы, поддерживаемые классами *XML* и *XMLnode*. Если в синтаксисе свойства или метода указано имя экземпляра `theNode`, поддерживаются экземпляры любого из *XML* и *XMLnode*. Если в синтаксисе свойства или метода указано имя экземпляра `xmlDoc`, то это должен быть экземпляр класса *XML*, а не класса *XMLnode*. Обратите внимание, что хотя класс *XML* – это подкласс *XMLnode*, корневой узел в вершине иерархии документа XML представ-

ляет собой экземпляр *XML* а не экземпляр *XMLnode*. Подузлы являются экземплярами *XMLnode*. По этой причине свойства и методы, относящиеся к документу XML в целом, а не отдельным узлам, такие как *load( )* и *send( )*, действуют только с экземплярами класса *XML*. Есть два типа узлов – узлы элементов и узлы текста, о чем рассказано в статье справочника, посвященной свойству *XML*.nodeType.

Исходный код документа XML состоит в основном из ряда элементов и атрибутов. Например, в следующем фрагменте XML элементы BOOK, TITLE, AUTHOR и PUBLISHER имеют такой же формат, как хорошо известные теги HTML, а элемент AUTHOR поддерживает один атрибут, SALUTATION:

```
<BOOK>
  <TITLE>ActionScript for Flash MX: The Definitive Guide</TITLE>
  <AUTHOR SALUTATION="Mr.">Colin Moock</AUTHOR>
  <PUBLISHER>O'Reilly</PUBLISHER>
</BOOK>
```

С точки зрения объектно-ориентированного программирования содержимое документа XML можно рассматривать как иерархию объектов, в которой каждый элемент и блок текста становится узлом объекта в структуре, похожей на блок-схему.

На рис. C.7 показан этот простой фрагмент XML <BOOK>, мысленно представленный в виде иерархии объектов XML.

Рис. C.7. *Пример иерархии узлов XML*

Рассмотрим слева направо структуру и семантику этого примера иерархии объектов XML. Начнем с главного объекта *XML*, показанного на рис. C.7 как DOCUMENT, который автоматически создается конструктором *XML* и служит контейнером для иерархии объектов XML.

Перемещаясь по иерархии на один ряд вправо, приходим к BOOK, представляющему собой первый элемент в данном фрагменте исходного кода XML и, в данном случае, также первый *узлел объекта* в DOCUMENT. Узел BOOK является *корнем* (*root*) этой структуры данных XML – у каждого корректного документа XML должен быть такой всеохватывающий элемент, как BOOK, в котором содержатся все другие элементы (корень дерева XML никак не связан с _root, глобальным свойством, которое обозначает главную временную диаграмму). Ветви иерархии объектов XML добавляются к дереву в результате анализа исходного кода XML или вызова методов, добавляющих узлы, над объектами иерархии.

Если один узел содержится в другом, то он называется *дочерним* для содержащего его узла, а тот называется *родительским*. В данном примере BOOK – это первый дочерний элемент DOCUMENT, а DOCUMENT – это *родитель* (*parent*) для BOOK (родитель XML никак не связан со свойством *MovieClip*._parent)

Перемещаясь по рисунку вправо, обнаруживаем, что у BOOK семь дочерних элементов, в том числе четыре текстовых узла #text, которых не было видно в исходном документе XML. По умолчанию каждый пробельный символ между элементами в исходном коде XML представляется объектом в иерархии объектов XML. Посмотрев внимательно, можно обнаружить пробельные символы – возврат каретки и символ табуляции – между BOOK и TITLE в предшествующем фрагменте XML. Этот пробел представлен на рис. C.7 в виде узла #text. Аналогичные пробельные узлы следуют за узлами узлов TITLE, AUTHOR и PUBLISHER.

Дочерние элементы BOOK являются один для другого братьями (то есть находятся в иерархии на одном уровне). Например, *следующий одноуровневый узел* (*next sibling*) для AUTHOR – #text, а *предыдущий одноуровневый узлел* (*previous sibling*) для AUTHOR – другой #text. Можно видеть, как узлы #text появляются при переходе от одного элемента к следующему на том же уровне. С этими пустыми пробельными узлами можно обращаться одним из следующих способов:

- Вручную удалить их из иерархии объектов (см. ниже примеры кода, удаляющего пробельные символы)
- Обнаруживать их и обходить стороной в коде (способы перехода через узлы предлагаются свойствами nextSibling и previousSibling)
- Просто удалить форматирующие пробельные символы в исходном тексте XML, что обеспечит отсутствие пробельных узлов
- Задать для свойства объекта *XML* ignoreWhite значение true, прежде чем анализировать источник XML

Наконец, мы перемещаемся к последнему ярусу в иерархии, где обнаруживаем, что у каждого из узлов TITLE, AUTHOR и PUBLISHER есть единственный потомок. Каждый дочерний узел представляет собой текстовый узел, соответствующий тексту, содержащемуся в элементах TITLE, AUTHOR и PUBLISHER. Обратите внимание, что текст, содержащийся в элементе исходного кода XML, располагается в соответствующей иерархии объектов в дочернем для этого элемента узле. Для доступа к тексту, содержащемуся в этом элементе, необходимо ссылаться на дочерний для этого элемента узел посредством firstChild.nodeValue или childNodes[0].nodeValue, которые вскоре будут рассмотрены.

Но что с атрибутами элементов? Где они появляются в иерархии объектов XML? Можно предположить, что атрибут SALUTATION элемента AUTHOR будет отображен как дочерний узел с именем SALUTATION. Но на практике атрибут считается не *дочерним узлом* определяющего его элемента, а свойством этого элемента. Узнать, как обращаться к свойствам атрибутов, можно в описании *XML*.attributes.

Посмотрим, как построить документ XML в виде иерархии объектов узлов. Для создания нового пустого объекта *XML* применяется конструктор *XML( )*:

```
myDocument = new XML();
```

После этого можно добавлять узлы в пустой объект *XML*, применяя к объекту такие методы, как *appendChild( )*, *parseXML( )* и *load( )*. Другой способ состоит в создании в сценарии объекта *XML* из исходного кода XML путем вызова конструктора XML с аргументом *source*:

```
myDocument = new XML(source);
```

Например:

```
myDocument = new XML('<P>hello world!</P>');
```

Когда конструктору *XML( )* передается аргумент *source*, последний подвергается синтаксическому анализу и преобразуется в новую иерархию объектов, которая записывается в объект, возвращаемый конструктором. (В данном случае узел P назначается первым дочерним узлом для myDocument, а текстовый узел с nodeValue "hello world!" назначается первым дочерним узлом P.)

После того как иерархия XML создана и сохранена в объекте, можно обращаться к информации в этой иерархии с помощью методов и свойств класса *XML*. Допустим, например, что требуется получить текст «hello world!» из myDocument. Думая на языке объектов, можно предположить, что текст P извлекается как свойство myDocument, например myDocument.P. На самом деле это не так: для доступа к узлам вместо ссылок по имени применяются встроенные свойства класса *XML*, например firstChild и childNodes. К узлу P можно обратиться так:

```
myDocument.firstChild       // Обращается к P
myDocument.childNodes[0]    // Тоже обращается к P
```

Поскольку firstChild возвращает ссылку на первый дочерний узел для заданного узла в иерархии, myDocument.firstChild возвращает ссылку на узел P. Однако нам нужен текст «hello world!», содержащийся в P, а не сам узел P. Как уже было показано, текст узла элемента хранится в его дочернем узле. Поэтому обратиться к текстовому узлу (то есть первому потомку P) можно так:

```
myDocument.firstChild.firstChild  // Обращается к текстовому узлу под P
```

Для получения значения узла применяется свойство nodeValue. Например, можно вывести текст «hello world!» в окно Output следующим образом:

```
trace(myDocument.firstChild.firstChild.nodeValue);
```

Можно присвоить новое значение текстовому узлу под P с помощью:

```
myDocument.firstChild.firstChild.nodeValue = "goodbye cruel world";
```

Чтобы выполнять с узлами другие операции, например полностью удалить узел P, добавить новый узел или переместить текст «hello world!» в другой узел, вызываются соответствующие методы класса *XML*. Например:

```
// Удалить P
myDocument.firstChild.removeNode();

// Создать новый элемент с именем P
newElement = myDocument.createElement("P");

// Добавить в документ новый элемент
myDocument.appendChild(newElement);

// Создать новый текстовый узел для прикрепления к P
newText = myDocument.createTextNode("XML is fun");

// Прикрепить новый текстовый узел к P
myDocument.firstChild.appendChild(newText);
```

Как видите, действия с данными в формате XML в иерархии объектов осуществляются опосредованно. Данные создаются, разрушаются и обрабатываются с помощью вызова методов и обращения к свойствам объектов. Для ознакомления с различными средствами, имеющимися для работы с данными XML, изучите свойства и методы класса *XML*, перечисленные после этой статьи.

Во Flash Player 6 текст XML должен быть в кодировке Unicode формата UTF-8, UTF-16BE или UTF-16LE, что позволяет включать международные символы. Файл в кодировке UTF-16 должен начинаться с маркера порядка байтов (Byte Order Marker, BOM), указывающего порядок байтов – прямой или обратный. Большинство текстовых редакторов автоматически записывают BOM. Если BOM в файле отсутствует, то предполагается кодировка UTF-8. В случае сомнения следует выбрать кодировку UTF-8, в которой порядок байтов не играет роли. Подробнее о UTF и BOM см. по адресу:

http://www.unicode.org/unicode/faq/utf_bom.html

ActionScript строит обработку данных XML на основе стандарта модели объектов документа (Document Object Model, DOM), опубликованной World Wide Web Consortium (W3C). Полное описание представления в DOM данных XML в виде иерархии объектов можно найти по адресу:

http://www.w3.org/DOM

Подробные сведения о независимых от языка спецификациях базовой DOM можно найти по адресу:

http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

В этом документе особое внимание обратите на подраздел «Interface Node» (Узел интерфейса) раздела 1.2. «Fundamental Interfaces»). О подробностях реализации DOM в ECMA-262 рассказано в документе:

http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/ecma-script-binding. html

Применение

Обратите внимание, что все пользовательские подклассы *XML* должны вызывать *super( )* из своего конструктора. Это обеспечит доступность методов и свойств класса *XML* экземплярам подкласса.

Например:

```
function XMLsub (src) {
  // Обязательно! Передать необработанную строку XML в конструктор XML().
  super(src);
}
```

```
XMLsub.prototype = new XML();

theDoc = new XMLsub();
theDoc.parseXML("<TEST>hello world</TEST>");
```

Пример

Как уже отмечалось, пробельные символы между любыми двумя элементами в исходном коде XML представляются текстовым узлом в соответствующей иерархии объектов *XML*. До выпуска Flash Player 5.0.41.0 нежелательные пробельные узлы приходилось вручную вырезать из иерархии объектов XML. Задача удаления узлов определенного типа часто возникает при обработке XML и служит хорошим примером *обхода дерева* (*traversal*), то есть прохода через каждый узел в иерархии. Рассмотрим два разных способа удаления пробельных узлов из документа.

В первом примере все узлы дерева помещаются в массив для дальнейшей обработки при помощи классического стека FIFO (First In First Out). Функция *stripWhitespaceTraverse( )* порождает массив элементов узлов с помощью *theNode*, передаваемого ей в качестве аргумента. Затем выполняется цикл, в котором первый узел в массиве удаляется, обрабатывается, а если у него есть дочерние узлы, то они добавляются в массив. Если в массиве не осталось элементов, это означает, что обработаны все потомки *theNode*. При обработке каждый узел, у которого нет дочерних узлов, рассматривается как потенциально пробельный (поскольку у текстовых узлов не бывает дочерних). Для каждого из этих узлов проверяется:

- Является ли узел текстовым (что определяется свойством nodeType)
- Содержатся ли в нем символы, код ASCII которых больше 32 и которые не считаются пробельными

Любые текстовые узлы, содержащие только символы, код ASCII которых меньше 32 (то есть только пробельные), удаляются:

```
// Удаление пробельных узлов с помощью стека FIFO
// Функция: stripWhitespaceTraverse()
// Версия: 1.0.1
// Описание: Удаляет пробельные узлы из документа XML путем обхода дерева
function stripWhitespaceTraverse (theNode) {
  // Создать список узлов, подлежащий обработке
  var nodeList = new Array();
  // Породить список из узла, переданного функции
  nodeList[0] = theNode;

  // Создать локальную переменную, которая будет следить, найден ли пустой узел.
  var emptyNode;
  // Локальная переменная счетчика.
  var i = 0;

  // Обрабатывать переданный узел и его потомков, пока больше не останется необработанных.
  while (nodeList.length > 0) {
    // Взять первый узел из списка и удалить его из списка.
    currentNode = nodeList.shift();

    // Предполагаем, что узел не пуст..
    emptyNode = false;

    // Если у этого узла есть дочерние...
    if (currentNode.childNodes.length > 0) {
      // ...добавить узлы, дочерние для данного, в список обрабатываемых узлов.
      nodeList = nodeList.concat(currentNode.childNodes);
    } else {
```

```
        // ...в противном случае этот узел является концевым в ветви,
        // поэтому проверить, не текстовый ли он…
        if (currentNode.nodeType == 3) {
          // Да, текстовый, поэтому проверим, есть ли в нем что-нибудь кроме пробелов.
          // Предполагаем, что только пробелы, и пытаемся это опровергнуть.
          emptyNode = true;
          for (i = 0; i < currentNode.nodeValue.length; i++) {
            // Полезным символом является любой, ASCII-код которого превосходит 32
            // (пробел, табуляция, перевод строки и т. д. находятся ниже).
            if (currentNode.nodeValue.charCodeAt(i) > 32) {
              emptyNode = false;
              break;
            }
          }
        }

        // Если полезные символы не найдены, удалить узел
        if (emptyNode) {
          currentNode.removeNode();
        }
      }
    }
  }
```

Прием, продемонстрированный в предшествующем примере, как правило, весьма эффективен. Однако во Flash Player 5 метод *Array.concat( )* выполняется довольно медленно. Ускорить удаление пробелов позволяет прием, показанный в следующем примере. Внимательно изучите комментарии в коде:

```
// Удаление пробельных символов с помощью рекурсивной функции
// Удаляет пробельные узлы из документа XML,
// дважды проходя каждый уровень дерева
function stripWhitespaceDoublePass(theNode) {
  // Обойти в цикле все дочерние узлы theNode
  for (var i = 0; i < theNode.childNodes.length; i++) {
    // Если текущий узел является текстовым...
    if (theNode.childNodes[i].nodeType == 3) {

      // ...проверить, есть ли в узле полезные символы
      var j = 0;
      var emptyNode = true;
      for (j = 0;j < theNode.childNodes[i].nodeValue.length; j++) {
        // Полезный символ имеет код больше 32 (пробел, табуляция,
        // перевод строки и т. д. находятся ниже)
        if (theNode.childNodes[i].nodeValue.charCodeAt(j) > 32) {
          emptyNode = false;
          break;
        }
      }

      // Если полезные символы не найдены, удалить узел
      if (emptyNode) {
        theNode.childNodes[i].removeNode();
      }
    }
  }

  // После удаления пробельных узлов из theNode рекурсивно вызвать
  // stripWhitespaceDoublePass() для остальных дочерних узлов.
  for (var k = 0; k < theNode.childNodes.length; k++) {
```

```
        stripWhitespaceDoublePass(theNode.childNodes[k]);
      }
    }
```

См. также *XMLnode*, класс; *XMLSocket*, класс; «Поддержка HTML», глава 18

XML.appendChild(), метод

Flash 5

добавляет к узлу новый дочерний узел или перемещает существующий

theNode.appendChild(*childNode*)

Аргументы

childNode Существующий объект узла XML.

Описание

Метод *appendChild( )* добавляет заданный *childNode* к *theNode* в качестве последнего дочернего для *theNode*, где *theNode* может быть экземпляром *XML* или *XMLnode*. С помощью *appendChild( )* можно добавить новый узел к существующему узлу, переместить узел внутри документа или переместить узел между документами. В каждом из этих случаев *childNode* должен быть ссылкой на существующий объект узла.

Для того чтобы добавить новый дочерний узел к уже существующему, необходимо предварительно создать новый дочерний узел с помощью *createElement( )*, *createTextNode( )* или *cloneNode( )*, либо он должен быть создан в результате синтаксического разбора исходного кода XML в объект *XML*. Например, в следующем коде создается новый узел P и новый текстовый узел. Текстовый узел добавляется к узлу P, а затем узел P и его дочерний текстовый узел добавляются к верхнему узлу документа:

```
// Создать документ
myDoc = new XML('<P>paragraph 1</P>');

// Создать узел P и текстовый узел
newP = myDoc.createElement("P");
newText = myDoc.createTextNode("paragraph 2");

// Добавить текстовый узел к узлу P
newP.appendChild(newText);

// Добавить узел P (вместе с дочерним текстовым) к myDoc
myDoc.appendChild(newP);

trace(myDoc);  // Выводит: "<P>paragraph 1</P><P>paragraph 2</P>"
```

Для того чтобы переместить узел внутри документа, задайте *childNode* как ссылку на существующий в документе узел. В этом случае *childNode* указывает прежнее положение узла, а *theNode* указывает нового родителя для узла. В процессе добавления к *theNode* узел *childNode* удаляется из прежнего родительского узла. В следующем примере узел B перемещается из родительского узла P в корневой узел документа:

```
// Создать новый документ
myDoc = new XML('<P>paragraph 1<B>bold text</B></P>');
// Сохранить ссылку на узел B
boldText = myDoc.firstChild.childNodes[1];

// Добавить узел B в корень документа, одновременно удалив его из P
myDoc.appendChild(boldText);

trace(myDoc);  // Выводит: "<P>paragraph 1</P><B>bold text</B>"
```

Можно также обойтись без запоминания ссылки и непосредственно переместить узел:

```
myDoc.appendChild(myDoc.firstChild.childNodes[1]);
```

Для того чтобы узел можно было переместить из одного документа в другой, *childNode* должен быть ссылкой на узел в первом (исходном) документе, а *theNode* – ссылкой на узел во втором (целевом) документе. В следующем примере узел B перемещается из myDoc в myOtherDoc:

```
myDoc = new XML('<P>paragraph 1<B>bold text</B></P>');
myOtherDoc = new XML();

myOtherDoc.appendChild(myDoc.firstChild.childNodes[1]);

trace(myDoc);       // Выводит: "<P>paragraph 1</P>"
trace(myOtherDoc);  // Выводит: "<B>bold text</B>"
```

См. также

XML.cloneNode( ), XML.createElement( ), XML.createTextNode( ), XML.insertBefore( ), XML. removeNode( )

XML.attributes, свойство[1]

Flash 5

объект, в свойствах которого хранятся атрибуты элемента **чтение/запись**

theNode.attributes.*attributeIdentifier*
theNode.attributes[*attributeNameInQuotes*]

Описание

Свойство attributes хранит имена и значения атрибутов, определенных для *theNode*, где *theNode* может быть экземпляром *XML* или *XMLnode*. Например, доступ к атрибуту ALIGN следующего тега P:

```
<P ALIGN="CENTER">this is a paragraph</P>
```

осуществляется через *theNode*.attributes.ALIGN или *theNode*.attributes["ALIGN"]. Если бы тег P был единственным в исходном коде XML, то обратиться к атрибуту ALIGN можно было бы так:

```
// Создать иерархию объекта XML
myDoc = new XML('<P ALIGN="CENTER">this is a paragraph</P>');

// Обращение к атрибуту ALIGN. Выводит: "CENTER"
trace(myDoc.firstChild.attributes.ALIGN);

// Задать атрибут ALIGN
myDoc.firstChild.attributes.ALIGN = "LEFT";
```

Свойство attributes представляет собой объект. К объекту attributes можно добавлять новые свойства, тем самым добавляя новые атрибуты в *theNode*, например:

```
// Добавить атрибут CLASS к тегу P
```

[1] Здесь есть ошибка реализации, которая может встретиться при разработке объектно-ориентированных приложений. Объект *attributes* не наследует никаких свойств и методов класса *Object* и вообще не имеет таких свойств, как __proto__ и constructor. Аналогичная ошибка встречается в объектах, возвращаемых функциями *TextField.getTextExtent( )*, *Color.getTransform( )* и *Sound.getTransform( )*. – *Примеч. науч. ред.*

```
myDoc.firstChild.attributes.CLASS = "INTRO";

// Теперь firstChild представляет такой исходный код XML:
// <P ALIGN="CENTER" CLASS="INTRO">this is a paragraph</P>
```

Так как attributes не является массивом, у него нет свойства length (длина), указывающего количество свойств/атрибутов, которые в нем содержатся. Обращаться ко всем атрибутам, определенным в элементе, можно с помощью цикла *for-in*:

```
var count = 0;
for (var prop in theNode.attributes) {
  trace("attribute " + prop + " has the value " + theNode.attributes[prop]);
  count++;
}
trace("The node has " + count + " attributes.");
```

Если у элемента XML, представляемого *theNode*, нет атрибутов, то attributes является пустым объектом, не имеющим свойств, и предыдущий пример покажет ноль атрибутов.

См. также *XML*.nodeType

XML.childNodes, свойство

Flash 5

массив ссылок на дочерние для данного узлы **только для чтения**

theNode.childNodes[*n*]

Описание

Свойство childNodes представляет собой массив, элементы которого содержат ссылки на непосредственных потомков *theNode*, где *theNode* может быть экземпляром *XML* или *XMLnode*. Оно применяется для доступа к узлам в иерархии XML. Создадим такую иерархию объектов:

```
myDoc = new XML('<STUDENT><NAME>Tim</NAME><MAJOR>BIOLOGY</MAJOR></STUDENT>');
```

После этого можно обратиться к узлу STUDENT следующим образом:

```
myDoc.childNodes[0];
```

К узлам NAME и MAJOR (потомкам STUDENT) можно обращаться так:

```
myDoc.childNodes[0].childNodes[0];     // NAME
myDoc.childNodes[0].childNodes[1];     // MAJOR
```

Если иерархия под *theNode* изменяется, childNodes автоматически обновляется, отражая новую структуру. Например, если удалить узел MAJOR, myDoc.childNodes[0].childNodes[1] возвратит undefined.

На узлы часто приходится ссылаться при обработке информации или реорганизации структуры документа. Например, можно изменить имя студента или добавить нового студента:

```
// Получить имя студента
trace("The student's name is: "
       + myDoc.childNodes[0].childNodes[0].childNodes[0].nodeValue);

// Изменить имя студента
myDoc.childNodes[0].childNodes[0].childNodes[0].nodeValue = "James";
```

```
// Копировать узел STUDENT
newStudent = myDoc.childNodes[0].cloneNode(true);

// Добавить в документ новый узел STUDENT
myDoc.appendChild(newStudent);
```

Обратите внимание, что для удобства можно ссылаться на childNodes[0] с помощью свойства firstChild. Следующие ссылки совпадают:

```
myDoc.childNodes[0];
myDoc.firstChild;
```

Обход всех дочерних узлов данного узла можно организовать посредством инструкции *for*, например:

```
for (var i = 0; i < theNode.childNodes.length; i++) {
  trace("child " + i + " is " + theNode.childNodes[i].nodeName);
}
```

Однако в этом примере осуществляется обход только первого уровня иерархии *theNode*. В примерах, приведенных в статье, посвященной *XML*.nextSibling, показано, как обратиться ко всем узлам под *theNode*. Если у *theNode* нет дочерних узлов, *theNode*.childNodes.length равно 0.

Применение

Помните, что пустые текстовые узлы, которые представляют пробельные символы, применяемые для форматирования исходного кода XML, также присутствуют в списке childNode. Например, для следующего исходного кода XML пустые текстовые узлы будут созданы пробельными символами после начального тега BOOK и конечных тегов TITLE, AUTHOR и PUBLISHER:

```
<BOOK>
  <TITLE>ActionScript for Flash MX: The Definitive Guide</TITLE>
  <AUTHOR SALUTATION="Mr">Colin Moock</AUTHOR>
  <PUBLISHER>O'reilly</PUBLISHER>
</BOOK>
```

Следовательно, первым дочерним узлом BOOK будет пустой текстовый узел, а вторым – узел TITLE. Для удаления пробельных символов применяется *XML*.ignoreWhite.

Ошибки реализации

Во Flash Player 5 у документа XML, содержащего объявление <?xml version="1.0"?>, свойство childNodes[0] имеет значение undefined. Первый дочерний элемент документа ошибочно записывается в childNodes[1]. Во Flash Player 6 эта ошибка исправлена.

См. также

XML.firstChild, ***XML.hasChildNodes( )***, *XML*.lastChild, *XML*.nextSibling, *XML*.previousSibling

XML.cloneNode(), метод

Flash 5

создает независимую копию узла

theNode.cloneNode(*deep*)

Аргументы

deep Булево значение, указывающее, следует ли рекурсивно включать дочерние узлы *theNode* в процесс клонирования (true – клонируется вся иерархия

начиная с *theNode*; false – клонируется только сам *theNode* и его атрибуты, если это узел элемента).

Возвращаемое значение

Дубликат объекта *theNode*, который по выбору может включать его поддерево.

Описание

Метод *cloneNode( )* создает и возвращает копию *theNode*, где theNode может быть экземпляром *XML* или *XMLnode*. Если *theNode* представляет собой узел элемента, то клон включает все атрибуты и значения *theNode*. Если *deep* имеет значение true, то в возвращаемой копии содержится вся иерархия узлов, являющихся потомками *theNode*.

Метод *cloneNode( )* часто применяется для создания нового независимого узла, исходя из существующего шаблона (что избавляет от необходимости вручную создавать структуру нового узла). Клонированный узел обычно модифицируется и помещается в существующий документ XML с помощью *appendChild( )* или *insertBefore( )*. В следующем примере клонируется первый абзац документа и создается абзац с той же структурой на том же уровне:

```
// Создать новый документ
myDoc = new XML('<P>paragraph 1</P>');

// Клонировать первый абзац
newP = myDoc.firstChild.cloneNode(true);

// Модифицировать клон
newP.firstChild.nodeValue = "paragraph 2";

// Добавить клон к документу
myDoc.appendChild(newP);

trace(myDoc);  // Выводит: "<P>paragraph 1</P><P>paragraph 2</P>"
```

Обратите внимание, что текст элемента хранится в отдельном дочернем узле этого элемента, поэтому значение *deep* надо установить равным true, чтобы сохранить при клонировании текстовое содержимое элемента. Помните, что *cloneNode( )* не вставляет возвращаемый им элемент в документ узла: это надо сделать отдельно с помощью *appendChild( )* или *insertBefore( )*.

См. также

XML.appendChild( ), *XML.createElement( )*, *XML.createTextNode( )*, *XML.insertBefore( )*

XML.contentType, свойство

Flash 5.0.41.0

тип содержимого MIME для данных XML, пересылаемых с помощью XML.send() и XML.sendAndLoad() — **чтение/запись**

xmlDoc.contentType

Описание

Свойство contentType содержит тип MIME, отправляемый серверу при вызове *XML.send( )* или *XML. sendAndLoad( )*. По умолчанию имеет значение application/x-www-form-urlencoded, но тип MIME оказывает действие чисто косметическое, способствуя взаимодействию с приложениями сервера: отправляемые данные не подверга-

ются кодировке URL. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*). Если свойство contentType задается для отдельного экземпляра *XML*, то оно действует только на этот экземпляр. Так, следующий код задает MIME лишь для doc:

```
doc.contentType = "application/xml";
```

Для того чтобы воздействовать на все объекты *XML*, задайте значение XML.prototype.contentType. Например, следующий код устанавливает для всех объектов XML фильма тип MIME "application/xml":

```
XML.prototype.contentType = "application/xml";
```

Свойство contentType впервые появилось во Flash Player 5.0.41.0, а до этого возможности задать тип MIME не было. Проверить, действует ли contentType в проигрывателе, можно сравнением его с undefined или с помощью функции *getVersion( )*, определяющей номер версии и сборки проигрывателя.

См. также *XML.send( )*, *XML.sendAndLoad( )*

XML.createElement(), метод

Flash 5

создает новый узел элемента

xmlDoc.createElement(*tagName*)

Аргументы

tagName Строка, задающая имя создаваемого элемента с учетом регистра. Например, в теге <P ALIGN="RIGHT"> именем тега является P.

Возвращаемое значение

Новый объект узла элемента, не имеющий родителя и потомков.

Описание

Метод *createElement( )* – это основной способ создания новых узлов элементов для включения в иерархию объектов документа *XML*. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*). Обратите внимание, что *createElement( )* не вставляет возвращаемый им элемент в *xmlDoc*: это надо сделать самостоятельно с помощью *appendChild( )* или *insertBefore( )*. Например, новый элемент P создается и вставляется в документ так:

```
myDoc = new XML();
newP = myDoc.createElement("P");
myDoc.appendChild(newP);
```

Эти шаги можно объединить следующим образом:

```
myDoc.appendChild(myDoc.createElement("P"));
```

Метод *createElement( )* нельзя применять для создания текстовых узлов, для этого предназначен *createTextNode( )*.

См. также

XML.appendChild( ), *XML.cloneNode( )*, *XML.createTextNode( )*, *XML.insertBefore( )*

XML.createTextNode(), метод

Flash 5

создает новый текстовый узел

xmlDoc.createTextNode(*text*)

Аргументы

text Строка, содержащая текст, который должен стать nodeValue для нового узла.

Возвращаемое значение

Новый объект текстового узла, не имеющий родителя и потомков.

Описание

Метод *createTextNode( )* представляет собой основное средство создания новых текстовых узлов для включения в иерархию объектов документа XML. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*). Обратите внимание, что *createTextNode( )* не включает в *xmlDoc* возвращаемый им элемент: это надо сделать самостоятельно с помощью *appendChild( )* или *insertBefore( )*. Вот, например, как можно создать новый элемент P и вставить его в документ, а затем создать для этого элемента P дочерний текстовый узел:

```
myDoc = new XML();
newP = myDoc.createElement("P");
myDoc.appendChild(newP);

newText = myDoc.createTextNode("This is the first paragraph");
myDoc.firstChild.appendChild(newText);

trace(myDoc);  // Выводит: "<P>This is the first paragraph</P>"
```

Текстовые узлы обычно хранятся как дочерние для узлов элементов, создаваемых с помощью *createElement( )*.

Метод *createTextNode( )* преобразует символы <, >, &, ' и " в соответствии с правилами преобразования сущностей, рассмотренными в статье, посвященной *XML.parseXML( )*.

См. также

XML.appendChild( ), *XML.cloneNode( )*, *XML.createElement( )*, *XML.insertBefore( )*, *XML. parseXML( )*

XML.docTypeDecl, свойство

Flash 5

тег DOCTYPE документа **чтение/запись**

xmlDoc.docTypeDecl

Описание

Строка свойства docTypeDecl задает тег DOCTYPE узла *xmlDoc*, если таковой существует. В противном случае docTypeDecl имеет значение undefined. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов *XML* (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Свойство DOCTYPE документа XML задает имя и местонахождение определения типа документа (DTD), используемого для проверки действительности документа. Action-

Script не проверяет действительность документов XML, а только анализирует их. Тег DOCTYPE применяется для создания документов XML, действительность которых можно проверить извне, или для идентификации типа загруженного документа XML.

Пример

```
var myXML = new XML('<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd">'
                    + '<P>a short document</P>');

trace(myXML.docTypeDecl);  // Выводит: "<!DOCTYPE foo SYSTEM "bar.dtd">"

// Установить новый DOCTYPE
myXML.docTypeDecl = '<!DOCTYPE baz SYSTEM "bam.dtd">';
```

См. также

XML.xmlDecl

XML.firstChild, свойство

Flash 5

ссылка на первого потомка узла **только для чтения**

theNode.firstChild

Описание

Свойство firstChild представляет собой синоним childNodes[0]. Оно возвращает ссылку на первый объект узла, являющийся потомком *theNode*, где *theNode* может быть экземпляром класса *XML* или класса *XMLnode*. Если у *theNode* нет дочерних узлов, firstChild возвращает null.

В следующем фрагменте исходного кода XML свойство firstChild узла MESSAGE является текстовым узлом с nodeValue, равным "hey":

```
<!-- Fragment 1 -->
<MESSAGE>hey</MESSAGE>
```

Ниже свойством firstChild узла HOTEL является узел ROOM:

```
<!-- Fragment 2 -->
<HOTEL><ROOM><SIZE>Double</SIZE></ROOM></HOTEL>
```

Если *theNode* – вершина иерархии узлов (то есть экземпляр класса XML, ссылающийся на объект документа *XML*, а не на экземпляр класса *XMLnode*), то firstChild может и не быть ссылкой на первый полезный элемент в документе. Если документ содержит объявление XML (<?xml version="1.0"?>), а возможно и тег DOCTYPE, то обычно перед фактическим корневым элементом иерархии XML есть пробельные узлы (их можно удалить с помощью *XML*.ignoreWhite). Однако если в фрагменте XML нет объявления XML, DOCTYPE и пробельных узлов, то начать обработку можно с узла документа firstChild, например:

```
// Создать новый фрагмент XML
myDoc = new XML('<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>');

// Сохранить первый узел фрагмента XML в переменной msg
msg = myDoc.firstChild;

// Присвоить текст, содержащийся в теге USER,
// текстовому полю с именем userNameOutput
userNameOutput = msg.firstChild.firstChild.nodeValue;
```

Применение nodeValue для доступа к тексту, содержащемуся в теге USER, считается хорошим стилем, хотя фактически не обязательно. Если объект текстового узла используется в строковом контексте, то для этого узла автоматически вызывается метод *toString( )* и возвращается текст, содержащийся в узле.

См. также *XML*.childNodes, *XML*.lastChild, *XML*.nextSibling, *XML*.previousSibling

XML.getBytesLoaded(), метод

Flash 6

возвращает количество загруженных байт внешнего файла XML

xmlDoc.getBytesLoaded()

Возвращаемое значение

Целое число, равное количеству байт, загрузка которых в проигрыватель завершена. Разделить на 1024 для преобразования в килобайты.

Описание

Метод *getBytesLoaded( )* сообщает о количестве байт внешнего файла XML, которые загрузились во Flash Player в результате предшествующего вызова *load( )* или *sendAndLoad( )*. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Для нового объекта *XML getBytesLoaded( )* возвращает undefined. Сразу после вызова *load( )* или *sendAndLoad( )* метод *getBytesLoaded( )* возвращает 0. По мере поступления данных *getBytesLoaded( )* возвращает количество байт, переданных на данный момент, пока не будет загружен весь файл XML. С этого момента он возвращает суммарное количество байт, переданных в последний раз, пока не поступит следующий вызов *load( )* или *sendAndLoad( )*.

Метод *getBytesLoaded( )* применяется совместно с *getBytesTotal( )*, чтобы определить (и обычно показать), как идет загрузка файла XML. Однако для того чтобы определить, завершилась ли операция загрузки, *getBytesLoaded( )* не требуется: по окончании загрузки файла XML автоматически выполняется обработчик события *onLoad( )*. Успешность попытки загрузить файл должна проверяться в обработчике *onLoad( )*.

Пример

Следующий код, который можно загрузить из сетевого хранилища кода, добавляет во все объекты *XML* два метода: *preload( )* и *sendAndPreload( )*. Они действуют так же, как *load( )* и *sendAndLoad( )*, за исключением того, что с помощью *getBytesLoaded( )* сообщают о ходе загрузки файла XML. Предполагается, что в объекте *XML*, вызвавшем *preload( )* или *sendAndPreload( )*, определена пользовательская функция обратного вызова *onBytesLoaded( )*, которая будет вызываться каждые 200 миллисекунд и получать данные о ходе загрузки (количество загруженных байт, процент выполнения загрузки и т. п.). Обычно код *onBytesLoaded( )* выводит данные о ходе загрузки на экран. В объекте *XML* должен также быть определен обработчик *onLoad( )* для ответа на завершение операции загрузки. При использовании с *preload( )* или *sendAndPreload( )* на функцию *onLoad( )* налагается дополнительная ответственность за прекращение неудачных попыток загрузки, как в следующем примере:

```
/*
 *    Метод: XML.checkLoadProgress()
 * Описание: Проверяет ход операции load() или sendAndLoad().
 *           Вызывается только из preload() или sendAndPreload(). Сообщает о ходе
 *           загрузки, вызывая пользовательский обработчик XML.onBytesLoaded().
```

```
 */
XML.prototype.checkLoadProgress = function () {
  // Узнать, сколько килобайт загружено.
  var kbLoaded = Math.floor(this.getBytesLoaded()/1024);
  // Узнать размер загружающегося файла в килобайтах.
  var kbTotal = Math.floor(this.getBytesTotal()/1024);
  // Вычислить степень выполнения (в процентах).
  var percentDone = isNaN(Math.floor(kbLoaded/kbTotal * 100)) ?
                    0 : Math.floor(kbLoaded/kbTotal * 100);
  // Выполнить onBytesLoaded(), который обычно показывает ход загрузки.
  this.onBytesLoaded(this.getBytesLoaded(),
                     this.getBytesTotal(),
                     kbLoaded,
                     kbTotal,
                     percentDone);
  // Если объем загруженных данных равен размеру файла, загрузка файла завершена.
  // Однако надо убедиться, что getBytesLoaded() возвращает число больше 0, потому что
  // getBytesLoaded() и getBytesTotal() возвращают 0, пока не поступит первый байт данных.
  if (this.getBytesLoaded() > 5
      && this.getBytesLoaded() == this.getBytesTotal()) {
    // ...прекратить вызов checkLoadProgress().
    this.clearLoadCheck();
  }
}

/*
 *    Метод: XML.clearLoadCheck()
 * Описание: Удаляет интервал, который вызывает checkLoadProgress().
 */
XML.prototype.clearLoadCheck = function () {
  // Если существует интервал загрузки для этого объекта XML...
  if (this.loaderID) {
    // ...прекратить его вызов.
    clearInterval(this.loaderID);
  }
}

/*
 *    Метод: XML.preload()
 * Описание: Начинает загрузку файла XML и выполняет
 *           checkLoadProgress() каждые 200 мс, пока файл не будет загружен.
 */
XML.prototype.preload = function (url) {
  // Прекратить выполнение другого предзагрузчика, если он выполняется.
  this.clearLoadCheck();

  // Загрузить файл XML.
  this.load(url);
  // Вызывать checkLoadProgress() каждые 200 мс.
  this.loaderID = setInterval(this, "checkLoadProgress", 200);
}

/*
 *     Метод: XML.sendAndPreload()
 * Описание: Отправляет XML на сервер и загружает ответ. Выполняет
 *           checkLoadProgress() каждые 200 мс, пока файл не будет загружен.
 */
XML.prototype.sendAndPreload = function (url, resultXML) {
  // Прекратить выполнение другого предзагрузчика, если он выполняется.
  this.clearLoadCheck();
```

```
  // Если целевой объект XML действителен...
  if (resultXML instanceof XML) {
    // ...отправить и загрузить файл XML.
    this.sendAndLoad(url, resultXML);
  // Вызывать checkLoadProgress() каждые 200 мс.
    this.loaderID = setInterval(this, "checkLoadProgress", 200);
  }  else {
    // ...иначе выход без выполнения загрузки.
    return false;
  }
}

// Применение:
// Создать объект XML.
doc = new XML();
// Присвоить функцию обратного вызова onLoad().
doc.onLoad = function (success) {
  // Если загрузка успешна...
  if (success) {
    // ...обработать новый документ XML.
    trace("done loading!");
  } else {
    // ...иначе обработка ошибок и удаление
    // интервала предзагрузчика (ОБЯЗАТЕЛЬНО!).
    trace("load failed");
    this.clearLoadCheck();
  }
}
// Присвоить функцию обратного вызова  onBytesLoaded(). В ней каждый отдельный
// объект XML может решить, что делать с информацией о предзагрузке.
doc.onBytesLoaded = function (bytesLoaded, bytesTotal,
                              kbLoaded, kbTotal, percentLoaded) {
  trace("Loading: " + kbLoaded +  " of " + kbTotal);
  trace(percentLoaded + " percent complete.");
}
// Загрузить файл XML.
doc.preload("http://www.yoursite.org/datafiles/someInfo.xml");
```

См. также

setInterval( ), *XML.getBytesTotal( )*, *XML.load( )*, *XML.onLoad( )*, *XML.sendAndLoad( )*

XML.getBytesTotal(), метод

Flash 6

возвращает физический размер на диске внешнего файла XML в байтах

xmlDoc.getBytesTotal()

Возвращаемое значение

Целое число, представляющее размер файла XML в байтах. Для преобразования в килобайты его следует разделить на 1024.

Описание

Метод *getBytesTotal( )* сообщает размер (в байтах) внешнего файла, загружаемого в *xmlDoc* посредством *XML.load( )* или *XML.sendAndLoad( )*. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Для нового объекта *XML* метод *getBytesTotal( )* возвращает `undefined`. Сразу после каждого вызова *loadSound( ) getBytesTotal( )* возвращает 0, пока не будет установлен размер загружаемого файла. С этого момента он начинает возвращать общий размер загружаемого файла, пока не поступит новый вызов *load( )* или *sendAndLoad( )*.

Метод *getBytesLoaded( )* применяется совместно с *getBytesTotal( )*, чтобы определить (и, как правило, показать), как идет загрузка файла XML. Подробности и пример кода см. в описании *getBytesLoaded( )*.

См. также

XML.getBytesLoaded( ), *XML.onLoad( )*, *XML.load( )*, *XML.sendAndLoad( )*

XML.hasChildNodes(), метод

Flash 5

проверяет, есть ли у узла потомки

theNode.hasChildNodes()

Возвращаемое значение

Булево значение `true`, если у *theNode* есть дочерние элементы, и `false`, если их нет.

Описание

Метод *hasChildNodes( )* показывает, исходит ли какая-либо иерархия узлов из *theNode*, где *theNode* может быть экземпляром класса *XML* или класса *XMLnode*. Метод синонимичен следующему выражению сравнения:

```
theNode.childNodes.length > 0
```

Если *theNode* не содержит подузлов, то *hasChildNodes( )* возвращает `false`.

Пример

Посредством *hasChildNodes( )* можно определить, следует ли обрабатывать тот или иной узел. Например, следующий код удаляет узлы под первым дочерним узлом документа, пока у первого дочернего узла не останется дочерних:

```
while (myDoc.firstChild.hasChildNodes()) {
  myDoc.firstChild.firstChild.removeNode();
}
```

См. также

XML.childNodes

XML.ignoreWhite, свойство

Flash 5.0.41.0

определяет, надо ли игнорировать пробельные узлы во время синтаксического анализа XML

чтение/запись

xmlDoc.ignoreWhite

Описание

Булево свойство `ignoreWhite` определяет, надо ли в процессе синтаксического анализа отбрасывать текстовые узлы, содержащие только пробельные символы. По умолчанию оно равно `false` (не отбрасывать пробельные узлы). Это глобальная установка, применяемая к документу XML в целом, а не к определенному узлу. Поэтому указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Пример

Отбрасывание пробельных символов в отдельном документе XML в ходе синтаксического анализа достигается выполнением следующего кода:

```
myXML.ignoreWhite = true;
```

А вот как можно сделать то же самое для всех документов XML:

```
XML.prototype.ignoreWhite = true;
```

Свойство `ignoreWhite` должно быть задано раньше, чем произойдет любая попытка разбора XML (обычно в результате операции *load( )* или *sendAndLoad( )*).

См. также

Примеры кода для удаления пробельных символов вручную в описании класса *XML*

XML.insertBefore(), метод

Flash 5

вставляет перед данным узлом новый на том же уровне

theNode.insertBefore(*newChild, beforeChild*)

Аргументы

newChild Существующий объект узла XML.

beforeChild Дочерний узел *theNode*, перед которым должен быть вставлен *newChild*.

Описание

Метод *insertBefore( )* добавляет *newChild* к списку дочерних узлов *theNode* перед *beforeChild* (theNode может быть экземпляром *XML* или *XMLnode*). Метод *insertBefore( )* аналогичен *appendChild( )*, но позволяет точно установить положение нового узла в существующей иерархии объектов XML.

Пример

```
// Создать документ с узлом P
myDoc = new XML('<P>paragraph 2</P>');

// Создать еще один узел P и текстовый узел
newP = myDoc.createElement("P");
newText = myDoc.createTextNode("paragraph 1");

// Добавить текстовый узел в новый узел P
newP.appendChild(newText);

// Вставить новый узел P (включая его текстового потомка) перед существующим P
myDoc.insertBefore(newP, myDoc.firstChild);

trace(myDoc);  // Выводит: "<P>paragraph 1</P><P>paragraph 2</P>"
```

См. также *XML.appendChild( )*

XML.lastChild, свойство

Flash 5

ссылка на последний дочерний узел для данного **только для чтения**

theNode.lastChild

Описание

Свойство `lastChild` представляет собой синоним выражения `childNodes[childNodes.length - 1]` и возвращает ссылку на объект последнего узла, являющегося потомком для *theNode*, где *theNode* может быть экземпляром *XML* или *XMLnode*. Если у *theNode* нет дочерних узлов, то `lastChild` возвращает `null`.

В следующем фрагменте исходного кода XML свойство `lastChild` узла `MESSAGE` является узлом `CONTENT`:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

Пример

```
// Создать новый документ XML
myDoc = new XML('<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>');

// Присваивает msg значение "hi", потому что firstChild для myDoc
// является MESSAGE, lastChild для MESSAGE является CONTENT,
// а firstChild для CONTENT является текстовый узел со значением "hi"
msg = myDoc.firstChild.lastChild.firstChild.nodeValue
```

См. также *XML*.childNodes, *XML*.firstChild, *XML*.nextSibling, *XML*.previousSibling

XML.load(), метод

Flash 5; во Flash 6 добавлены ограничения безопасности

импортирует исходный код XML из внешнего документа

xmlDoc.load(*url*)

Аргументы

url Строка, указывающая адрес загружаемого документа XML.

Описание

Метод *load( )* импортирует внешний документ XML, анализирует его, преобразует в иерархию объектов XML и помещает эту иерархию в *xmlDoc*. Прежнее содержимое *xmlDoc* заменяется вновь загруженным содержимым XML. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Прежде чем обращаться к содержимому, импортированному *load( )*, необходимо убедиться в завершении операций загрузки и синтаксического анализа. Для этого можно проверить значение свойства `loaded` документа XML или назначить для документа обработчик обратного вызова *onLoad( )*, вызываемый при завершении загрузки. Подробную информацию можно найти в описании *XML*.loaded и *XML.onLoad( )*. Для того чтобы определить, успешно ли завершился синтаксический анализ загруженных данных, надо проверить свойство документа `status`. Для контроля за ходом передачи файла применяются методы *getBytesLoaded( )* и *getBytesTotal( )*.

Во Flash 6 на *XML.load( )* распространяются связанные с безопасностью ограничения по домену, описанные в статье *System.security.allowDomain( )*.

Пример

```
myDoc = new XML();
myDoc.load("myData.xml");
```

См. также *XML*.loaded, *XML.onLoad( )*, *XML.sendAndLoad( )*, *XML*.status

XML.loaded, свойство

Flash 5

статус операций load() или sendAndLoad() **только для чтения**

xmlDoc.loaded

Описание

Свойство loaded возвращает булево значение, указывающее на завершение ранее вызванных операций *load( )* или *sendAndLoad( )*, выполняемых над *xmlDoc*. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Свойство loaded устанавливается равным false в самом начале операций XML *load( )* или *sendAndLoad( )*. Если загрузка была успешной, то loaded присваивается значение true. Если над *xmlDoc* не выполнялась ни одна такая операция, loaded имеет значение undefined.

Если loaded равно false, это означает, что загрузка и разбор данных XML все еще продолжаются, и попытки доступа к иерархии объектов в *xmlDoc* окажутся неудачными. Если loaded равно true, значит, данные XML загружены, проанализированы и записаны в *xmlDoc* в виде иерархии объектов. Обратите, однако, внимание, что синтаксический анализ загруженных данных XML может оказаться неудачным (это можно выяснить, прочитав свойство *xmlDoc*.status).

Пример

В следующем примере показан базовый предзагрузчик XML, который ждет окончания загрузки данных XML, прежде чем вывести их (предзагрузчики XML можно также создавать на основе обработчика события *XML.onLoad( )* или методов *XML.getBytesLoaded( )* и *XML.getBytesTotal( )*):

```
// КОД В КАДРЕ 1
// Создать новый документ XML
myDoc = new XML();
// Загрузить в документ внешний файл XML
myDoc.load("userProfile.xml");

// КОД В КАДРЕ 5
// Проверить, загружены ли данные. Если да, перейти в кадр отображения.
// Если нет, продолжить воспроизведение с кадра 4.
// Выполнять циклически до окончания загрузки данных...
if (myDoc.loaded) {
  if (myDoc.status == 0) {
    gotoAndStop("displayData");
  } else {
    gotoAndStop("loadingError");
  }
} else {
  gotoAndPlay(4);
}
```

См. также *XML.load( )*, *XML.onLoad( )*, *XML.sendAndLoad( )*, *XML*.status

XML.nextSibling, свойство

Flash 5

ссылка на узел, следующий за данным **только для чтения**

theNode.nextSibling

Описание

Свойство nextSibling возвращает объект узла, следующего за *theNode* на текущем уровне иерархии объектов XML (*theNode* может быть экземпляром *XML* или *XMLnode*). Если после *theNode* нет узлов, nextSibling возвращает null. В следующем фрагменте исходного кода XML узел CONTENT является nextSibling для узла USER:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

Пример

Свойство nextSibling обычно применяется для обхода иерархии объектов XML. Например, следующий код позволяет просмотреть все дочерние элементы *theNode* в порядке их появления:

```
for (var child = theNode.firstChild; child != null; child = child.nextSibling) {
  trace("found node: " + child.nodeName);
}
```

Развив этот цикл в функцию, можно рекурсивно обойти все узлы в иерархии объектов XML, например:

```
function showNodes (node) {
  trace(node.nodeName + ": " + node.nodeValue);
  for (var child = node.firstChild; child != null; child = child.nextSibling) {
    showNodes(child);
  }
}

// Вызвать функцию с узлом или документом
showNodes(myDoc);
```

Обратите внимание, что в обоих приведенных примерах обхода текстовые узлы показываются без имен, согласно описанию nodeName.

См. также

XML.childNodes, *XML*.firstChild, *XML*.lastChild, *XML*.nodeName, *XML*.nodeValue, *XML*.previousSibling

XML.nodeName, свойство

Flash 5

имя текущего узла **чтение/запись**

theNode.nodeName

Описание

Строковое свойство nodeName отражает имя *theNode*, где *theNode* может быть экземпляром *XML* или *XMLnode*. ActionScript поддерживает только два типа узлов (узлы элементов и текстовые узлы), поэтому nodeName может иметь только два возможных значения:

- Если *theNode* – это узел элемента, то nodeName представляет собой строку, совпадающую с именем тега этого элемента. Например, если *theNode* представляет элемент <BOOK>, то *theNode.nodeName* имеет значение "BOOK".
- Если *theNode* является текстовым узлом, nodeName имеет значение null. Обратите внимание, что это отход от спецификации DOM, определяющей nodeName для текстового узла как строку "#text". При желании можно обратиться к совместимому с DOM свойству nodeType.

Пример

Свойство nodeName позволяет проверить, является ли текущий узел тем элементом, который разыскивается. Например, в следующем коде извлекается все содержимое тегов H1 на первом уровне документа XML (в этом примере ведется также поиск тегов с именем *h1*):

```
myDoc = new XML('<H1>first heading</H1><P>content</P>' +
                '<H1>second heading</H1><P>content</P>');
for (i = 0; i < myDoc.childNodes.length; i++) {
  if (myDoc.childNodes[i].nodeName.toUppercase() == "H1") {
    trace(myDoc.childNodes[i].firstChild.nodeValue);
  }
}
```

См. также *XML*.nodeType, *XML*.nodeValue

XML.nodeType, свойство

Flash 5

тип текущего узла **только для чтения**

theNode.nodeType

Описание

Свойство nodeType содержит целое число, представляющее тип *theNode* (theNode может быть экземпляром *XML* или *XMLnode*). В ActionScript поддерживаются только два типа узлов – узлы элементов и текстовые узлы, – поэтому nodeName может принимать лишь два значения – 1 для узла элемента и 3 для текстового узла. Может показаться, что они назначены произвольно, но на самом деле они соответствуют спецификации DOM. Для справки в табл. C.26 перечислены прочие типы узлов в DOM.

Таблица C.26. Типы узлов в DOM

Описание узла	Код типа узла
ELEMENT_NODE* (узел элемента)	1
ATTRIBUTE_NODE (узел атрибута)	2
TEXT_NODE* (текстовый узел)	3
CDATA_SECTION_NODE (узел раздела CDATA)	4
ENTITY_REFERENCE_NODE (узел ссылки на сущность)	5
ENTITY_NODE (узел сущности)	6
PROCESSING_INSTRUCTION_NODE (узел инструкции обработки)	7
COMMENT_NODE (узел комментария)	8
DOCUMENT_NODE (узел документа)	9
DOCUMENT_TYPE_NODE (узел типа документа)	10
DOCUMENT_FRAGMENT_NODE (узел фрагмента документа)	11
NOTATION_NODE (узел нотации)	12

* Поддерживается Flash.

Формально ActionScript реализует так называемые узлы *attribute*, *document* и *document_type* помимо узлов *element* и *text*, но прямого доступа к ним как к объектам нет. Например, работать с атрибутами узла можно через свойство attributes, но непосредственного доступа к самим узлам *attribute* нет. Аналогично можно обращаться к тегу документа DOCTYPE через свойство docTypeDecl, но непосредственный доступ к самому *document_type* невозможен.

Узлы элементов соответствуют тегам XML или HTML. Так, во фрагменте XML <P>what is your favorite color?</P> тег P будет представлен в иерархии объектов *XML* как узел элемента (nodeType 1). Текст, содержащийся в теге в исходном коде XML, например «what is your favorite color?», будет представлен как текстовый узел (nodeType 3). Узлы секций CDATA безвозвратно преобразуются при синтаксическом анализе в текстовые узлы, о чем говорится в статье, посвященной методу *XML.parseXML( )*.

Пример

В зависимости от типа узла (свойства nodeType) можно осуществлять над ним условные операции. Приведем пример кода, удаляющего все пустые текстовые узлы, являющиеся потомками *theNode*:

```
// Циклический обход всех дочерних для theNode узлов
for (i = 0; i < theNode.childNodes.length; i++) {
  // Если текущий узел текстовый...
  if (theNode.childNodes[i].nodeType == 3) {
    // Проверить, есть ли в этом узле значимые символы
    var emptyNode = true;
    for (var j=0; j < theNode.childNodes[i].nodeValue.length; j++) {
      // Коды значимых символов лежат выше ASCII 32
      if (theNode.childNodes[i].nodeValue.charCodeAt(j) > 32) {
        emptyNode = false;
        break;
      }
    }
    if (emptyNode) {
      // Значимых символов не найдено, поэтому узел удаляется
      theNode.childNodes[i].removeNode();
    }
  }
}
```

См. также

XML, класс; *XMLnode*.attributes, *XML*.docTypeDecl, *XML*.nodeName, *XML*.nodeValue

XML.nodeValue, свойство

Flash 5

значение текущего узла в виде строки **чтение/запись**

theNode.nodeValue

Описание

Свойство nodeValue отражает строковое значение *theNode*, где *theNode* может быть экземпляром *XML* или *XMLnode*. В ActionScript поддерживается лишь два типа узлов (узлы элементов и текстовые узлы), поэтому у nodeValue есть только два возможных значения:

- Если *theNode* представляет собой узел элемента, то nodeValue равно null.

- Если *theNode* – это текстовый узел, то nodeValue имеет значение текста, содержащегося в узле.

Присвоить новый текст существующему текстовому узлу с помощью nodeValue можно так:

```
// Создать новый документ XML
myDoc = new XML('<H1>first heading</H1><P>content</P>');

// Изменить текст, содержащийся в теге H1
myDoc.firstChild.firstChild.nodeValue = "My Life Story";
```

Для того чтобы получить текст текстового узла, достаточно прочесть его свойство nodeValue:

```
trace(myDoc.firstChild.firstChild.nodeValue);
```

Учтите, что в nodeValue предопределенные сущности <, >, &, " и ' представлены литеральными символами <, >, &, " и '. Для получения исходного кода узла, включая неинтерпретированные сущности, применяется метод *XML.toString( )*. Подробнее об этом рассказано в статье, посвященной *XML.parseXML( )*.

Если узел встречается в строковом контексте, Flash представляет его значение с помощью *XML.toString( )*, а не nodeValue:

```
// Инициировать неявное преобразование типа данных. Вызывается toString().
trace(myDoc.firstChild.firstChild);
// То же самое, что:
trace(myDoc.firstChild.firstChild.toString());
```

См. также *XML*.nodeName, *XML*.nodeType, *XML.parseXML( )*, *XML.toString( )*

XML.onData(), обработчик события

Flash 5

выполняется, когда загрузка внешнего исходного кода XML завершена, но синтаксический анализ еще не произведен

xmlDoc.onData(*src*);

Аргументы

src Строка, содержащая загруженный исходный код XML.

Описание

Обработчик *onData( )* автоматически выполняется, когда завершена загрузка необработанного исходного кода XML в *xmlDoc*, обусловленная предшествующим вызовом *load( )* или *sendAndLoad( )*. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

По умолчанию *onData( )* работает в следующем режиме:

- Если загруженный исходный код имеет значение undefined, вызывается *xmlDoc.onLoad( )* с параметром success, равным false.
- В противном случае исходный код разбирается в *xmlDoc*, свойству *xmlDoc*.loaded присваивается значение true и вызывается *xmlDoc.onLoad( )* с параметром success, равным true.

Обработчику *onData( )* можно присвоить пользовательскую функцию обратного вызова, чтобы перехватить исходный код XML, прежде чем ActionScript сможет начать

его синтаксический анализ. В некоторых случаях обработка исходного кода XML вручную может оказаться эффективнее встроенного анализатора ActionScript, особенно во Flash Player 5. Во Flash Player 6 собственный разбор XML обычно выполняется быстрее, чем при помощи пользовательских подпрограмм на ActionScript.

Пример

В следующем примере показано, как вывести загруженный необработанный исходный код XML и не дать ActionScript его анализировать:

```
myDoc = new XML();
myDoc.onData = function (src) {
  trace("Here's the source: \n" + src);
};
myDoc.load("book.xml");
```

В следующем примере перехватывается и показывается загруженный необработанный исходный код XML, но при этом выполняются и обычные действия *onData( )*:

```
myDoc = new XML();
myDoc.onData = function (src) {
  trace("Here's the source: \n" + src);
  if (src == undefined) {
    this.onLoad(false);
  } else {
    this.parseXML(src);
    this.loaded = true;
    this.onLoad(true);
  }
};
myDoc.onLoad = function (success) {
  trace("Here's the parsed XML: " + this);
  trace("Load succeeded? " + success);
};
myDoc.load("book.xml");
```

См. также *XML.onLoad( )*, *XML.load( )*

XML.onLoad(), обработчик события **Flash 5**

выполняется, когда внешние данные XML загружены и проанализированы

xmlDoc.onLoad(*success*)

Аргументы

success Булево значение, указывающее, была ли загрузка успешной (`true`) или неудачной (`false`).

Описание

Обработчик *onLoad( )* объекта *xmlDoc* автоматически выполняется, когда внешний файл XML загружен в *xmlDoc* с помощью методов *load( )* или *sendAndLoad( )*. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

По умолчанию обработчик *onLoad( )* объекта документа XML имеет значение `undefined`. Для того чтобы можно было работать с *onLoad( )*, ему следует присвоить обработчик обратного вызова, то есть пользовательскую функцию:

```
myDoc = new XML();
myDoc.onLoad = handleLoad;
function handleLoad (success) {
  // Здесь XML обрабатывается как положено...
}
```

События *onLoad( )* сообщают о том, что можно безбоязненно обрабатывать *xmlDoc*. Если сгенерировано событие *onLoad( )*, то известно, что загрузка и синтаксический анализ внешних данных XML завершены, и можно безопасно обращаться к этому загруженному содержимому. Поэтому обработчик *onLoad( )* устраняет необходимость написания кода предварительной загрузки, который должен ждать поступления данных после вызова функции XML *load( )*. Так, в следующем коде загружается документ XML, а затем ожидается автоматическое выполнение пользовательской функции *handleLoad( )* по завершении загрузки. Если загрузка была успешной, содержимое XML выводится с помощью функции *displayProduct( )*. В противном случае с помощью функции *displayProduct( )* выводится сообщение об ошибке. (Функции *displayProduct( )* и *displayError( )* – пользовательские и написаны для вывода информации пользователю, но здесь они не показаны.) Вот этот код:

```
myDoc = new XML();
myDoc.onLoad = handleLoad;
myDoc.load("productInfo.xml");

function handleLoad(success) {
  if (success) {
    output = "Product information received";
    displayProduct(); // Вызвать пользовательскую функцию вывода
  } else {
    output = "Attempt to load XML data failed";
    displayError(); // Вызвать пользовательскую функцию вывода сообщения об ошибке
  }
}
```

Обратите внимание, что присваивание нашей функции обратного вызова *onLoad( )* выполняется раньше, чем вызов *load( )*, что гарантирует наличие обработчика при завершении операции *load( )*.

Интерпретатор автоматически устанавливает значение аргумента *success* функции *handleLoad( )* равным `true` или `false` в зависимости от успеха или неудачи загрузки. Во Flash 6 *success* получает значение `false` в случае ошибки «File Not Found (404)» и нарушении правил безопасности (см. *System.security.allowdomain( )*). Однако во Flash 5 получение с сервера страницы с сообщением об ошибке приводит к разбору этой страницы в целевой объект документа XML. Страница анализируется успешно, поэтому попытка загрузки оказывается «успешной», и *success* получает значение `true`, хотя действительный файл XML может оказаться не найденным или могла возникнуть другая ошибка сервера.

Чтобы иметь уверенность в действительном получении запрошенных данных, необходимо явно проверить наличие в их структуре или содержимом какой-либо идентифицирующей характеристики, например `nodeName`, для определенного дочернего узла. Проверьте *XML*.`status`, чтобы выяснить, был ли успешным синтаксический разбор. См. также обработчик события *XML.onData( )*, позволяющий производить специализированный синтаксический анализ.

См. также *XML.load( )*, *XML.onData( )*, *XML.sendAndLoad( )*, *XML*.`status`

XML.parentNode, свойство

Flash 5

ссылка на ближайшего предка узла **только для чтения**

theNode.parentNode

Описание

Свойство parentNode возвращает ссылку на объект узла в иерархии объектов XML, для которого *theNode* является дочерним. *theNode* может быть экземпляром *XML* или *XMLnode*. Если *theNode* представляет собой верхний (корневой) узел в текущей иерархии, то parentNode возвращает null.

В следующем фрагменте исходного кода XML узел MESSAGE играет роль parentNode для текстового узла "hey":

```
<MESSAGE>hey</MESSAGE>
```

А здесь – узел HOTEL для узла ROOM:

```
<HOTEL><ROOM><SIZE>Double</SIZE></ROOM></HOTEL>
```

См. также *XML*.childNodes, *XML*.firstChild, *XML*.lastChild, *XML*.previousSibling

XML.parseXML(), метод

Flash 5

осуществляет синтаксический анализ строки с исходным кодом XML

xmlDoc.parseXML(*string*)

Аргументы

string Строка с исходным кодом XML.

Описание

Метод *parseXML( )* читает и анализирует исходный код XML, содержащийся в *string*, преобразует его в иерархию объектов и помещает полученную иерархию в *xmlDoc*. Любое прежнее содержимое *xmlDoc* заменяется новой иерархией. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*). После завершения синтаксического анализа узнать о его успешности можно через свойство *xmlDoc*.status.

Для того чтобы включить необработанный исходный код HTML или XML в текстовый узел, не выполняя его анализ, следует включить секцию CDATA:

```
<![CDATA[ source ]]>
```

Так, в следующем коде создается элемент MESSAGE с единственным дочерним текстовым узлом, содержащим текст «<B>Welcome</B> to my site» (тег <B> *не* интерпретируется как тег XML и не становится частью иерархии объектов XML):

```
myDoc = new XML();
myDoc.parseXML("<MESSAGE><![CDATA[<B>Welcome</B> to my site]]></MESSAGE>");
```

Обратите внимание, что секция CDATA интерпретируется как узел, поэтому у следующего элемента <P> фактически три дочерних узла – текстовый узел, узел CDATA и еще один текстовый узел:

```
<P>The following source: <![CDATA[<B>Welcome</B> to my site]]> adds bold.</P>
```

Однако в ActionScript поддерживаются только текстовые узлы и узлы элементов, поэтому данная секция CDATA преобразуется в текстовый узел с `nodeType` 3 (текстовый узел), а не 4 (узел секции CDATA). После завершения синтаксического анализа исходного кода CDATA последний нельзя восстановить из иерархии объектов XML.

Об употреблении символов & и < в символьных данных в спецификации W3C XML 1.0 сказано:

> «Символы амперсанд (&) и левая угловая скобка (<) могут присутствовать в своем буквальном виде только как разделители разметки внутри комментариев, инструкций обработки или секций CDATA. В других местах они должны быть преобразованы в цифровые ссылки на символы или строки '&' и '<' соответственно».[1]

Так, чтобы включить код «`<P>We know that (10<11) is true.</P>`» в документ XML, следует использовать сущность `<`:

```
<P>We know that (10&lt;11) is true.</P>
```

или секцию CDATA:

```
<P>We know that<![CDATA[ (10<11) ]]>is true.</P>
```

Начиная с версии 5.0.41.0 (5.0.42.0 в Internet Explorer) при разборе Flash исходного кода XML, в котором есть текстовый узел с символами >, ", ' или &, последние автоматически преобразуются в предопределенные сущности `>`, `"`, `'` и `&`. Например, в следующем коде символ ' превращается в `'`:

```
// Символ апострофа автоматически преобразуется в сущность
var doc = new XML("<p>don't worry</p>");
// Выводит: "don&apos;t worry"
trace(doc.firstChild.firstChild.toString());
```

Напротив, предопределенная сущность в текстовом режиме не изменяется во время синтаксического анализа. Например, сущность `&` при синтаксическом разборе так и остается сущностью `&`, не преобразуясь в & или `&`. Но если предопределенная сущность появляется в секции CDATA, то начальный амперсанд преобразуется, то есть `&` действительно превращается в `&`. Поэтому в CDATA всегда следует использовать буквальные символы, а не сущности.

Символ «меньше» (<) преобразуется в `<`, только если появляется в секции CDATA. Во всех прочих местах вместо него следует писать `<`, чтобы избежать ошибки синтаксического анализа.

При чтении свойства `nodeValue` текстового узла предопределенные сущности `<`, `>`, `"`, `'` и `&` автоматически преобразуются обратно в соответствующие символы:

```
var doc = new XML("<p>don&apos;t worry</p>");
// Выводит: "don't worry"
trace(doc.firstChild.firstChild.nodeValue);
```

Обратите внимание на разницу: *XML.toString( )* текстового узла возвращает необработанный исходный код XML узла, оставляя сущности неизмененными; *XML*`.nodeValue` возвращает интерпретированный текст, пригодный к выводу, в котором сущности преобразованы в буквальные символы. Например:

---

[1] Из раздела «2.4 Character Data and Markup», *http://www.w3.org/TR/2000/REC-xml-20001-006#syntax*.

```
doc = new XML("<P><![CDATA[what a 'nice' day]]></P>");
// Выводит: "what a &apos;nice&apos; day"
trace(doc.firstChild.firstChild.toString());
// Выводит: "what a 'nice' day"
trace(doc.firstChild.firstChild.nodeValue);
```

Применение

Учтите, что во Flash 6 возврат каретки, встретившийся в значении атрибута, вызывает ошибку синтаксического разбора с кодом –8 (нет конца атрибута). Загрузив XML, можно убрать мешающие возвраты каретки вручную внутри обработчика *onData( )*.

Пример

Метод *parseXML( )* позволяет заменить текущую иерархию объектов в объекте *XML* новой иерархией, основанной на внутренне порожденном исходном коде XML (например, некоторых данных, введенных пользователем). В следующем примере путем объединения разметки с данными ввода из текстовых полей `username_txt` и `content_txt` создается простое сообщение XML:

```
myDoc = new XML();
myXMLsource = "<MESSAGE><USER>" + username_txt + "</USER><CONTENT>"
    + content_txt + "</CONTENT></MESSAGE>";
myDoc.parseXML(myXMLsource);
```

См. также

XML.ignoreWhite, *XML.load( )*, *XML*.nodeValue, *XML.onData( )*, *XML*.status, *XML.toString( )*

XML.previousSibling, свойство

Flash 5

ссылка на узел, предшествующий данному **только для чтения**

theNode.previousSibling

Описание

Свойство `previousSibling` возвращает ссылку на объект узла, предшествующего *theNode* на текущем уровне иерархии объектов XML (*theNode* может быть экземпляром *XML* или *XMLnode*). Если на текущем уровне иерархии перед *theNode* нет узлов, то `previousSibling` возвращает `null`.

В следующем фрагменте исходного кода XML узел `CONTENT` играет роль `previousSibling` для узла `USER`:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

Пример

Свойство `previousSibling` позволяет осуществлять обход иерархии объектов XML, хотя чаще для этого применяется `nextSibling`. Ниже показано, как можно просмотреть все дочерние элементы *theNode* в обратном порядке:

```
for (var i = theNode.lastChild; i != null; i = i.previousSibling) {
  trace("found node: " + i.nodeName);
}
```

См. также

XML.childNodes, *XML*.firstChild, *XML*.lastChild, *XML*.nextSibling, *XML*.nodeName, *XML*.nodeValue, *XML*.parentNode

XML.removeNode(), метод

Flash 5

удаляет узел из иерархии объектов XML

theNode.removeNode()

Описание

Метод *removeNode( )* удаляет *theNode* из документа XML, где *theNode* может быть экземпляром *XML* или *XMLnode*. Все потомки (дети, внуки и т. д.) *theNode* тоже удаляются. Свойство childNodes родительского для *theNode* узла автоматически обновляется, чтобы отразить новую структуру оставшейся иерархии объектов.

Пример

Следующий код удаляет второй дочерний узел, место которого занимает третий дочерний узел:

```
myDoc = new XML("<P>one</P><P>two</P><P>three</P>");
myDoc.childNodes[1].removeNode();
trace(myDoc);  // Выводит: "<P>one</P><P>three</P>"
```

См. также *XML.appendChild( )*, *XML.insertBefore( )*

XML.send(), метод

Flash 5

передает исходный код XML внешнему сценарию или приложению

xmlDoc.send(*url*, *window*)

Аргументы

url Строка, указывающая местонахождение сценария или приложения, которому должен быть передан *xmlDoc*.

window Обязательная строка, указывающая имя окна или фрейма броузера, куда должен быть загружен ответ сценария. Может быть имя, заданное пользователем, или одно из четырех предустановленных имен: "_blank", "_parent", "_self" или "_top". Подробности приведены в описании настроек окна для глобальной функции *getURL( )*.

Описание

Метод *send( )* преобразует *xmlDoc* в строку исходного кода XML и отправляет этот код в запросе HTTP сценарию или приложению по *url*. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Предполагается, что сценарий или приложение некоторым образом обрабатывают XML и, возможно, возвращают ответ – обычно веб-страницу – броузеру, который отображает его в *window*. Обратите внимание, что ответ перехватывается не Flash, а броузером. Для того чтобы ответ перехватывался Flash, следует применять *sendAndLoad( )*.

Когда *send( )* вызывается из Flash Player, выполняющегося в броузере, *xmlDoc* посылается с помощью метода POST. Когда *send( )* вызывается из Flash Player, выполняемого как самостоятельное приложение, *xmlDoc* посылается с помощью метода GET. (Подробное описание поддержки GET и POST см. в приложении F.) Приложение сервера, получающее переданную строку XML, должно иметь возможность непосредст-

венного доступа к необработанным данным POST в запросе HTTP и не должно пытаться обрабатывать их как обычные пары имя/значение. В Perl данные запроса POST доступны из STDIN и могут быть извлечены и сохранены, скажем, в переменной `$buffer` следующим образом:

```
read(STDIN,$buffer,$ENV{'CONTENT_LENGTH'});
```

В ASP доступ к необработанным данным POST может быть осуществлен с помощью метода *Request.BinaryRead*.

В PHP 4.1 и выше, установив следующий параметр в *php.ini*, можно получать необработанные данные POST независимо от типа MIME запроса:

```
always_populate_raw_post_data = On
```

Если этот параметр включен, то для каждого запроса POST данные присваиваются переменной `$HTTP_RAW_POST_DATA`. В некоторых приложениях (например, в Cold Fusion до версии ColdFusion MX) может отсутствовать прямой метод доступа к данным запроса POST. В такой ситуации может оказаться необходимым сначала преобразовать объект XML в строку с помощью *XML.toString( )*, а затем передать эту строку серверу как переменную с помощью *loadVariables( )*.

По умолчанию серверу передается тип содержимого MIME `application/x-www-form-urlencoded`. Однако этот тип – чисто внешний, сам текст не имеет кодировки URL. Во Flash Player 5.0.41.0 и выше тип содержимого MIME может быть модифицирован с помощью свойства `XML.contentType`. Например, можно задать тип MIME равным `application/xml`:

```
myXML = new XML();
myXML.contentType = "application/xml";
```

Тем не менее явное задание свойства `contentType` как `application/x-www-form-urlencoded` не влечет URL-кодировки отправляемого текста.

Обратите внимание, что во Flash Player 5.0.41.0 при анализе исходного кода XML и обнаружении в текстовом узле символов &, ', ", < и > эти символы преобразуются в сущности `&`, `'`, `"`, `>`, `<`. Во Flash сущности преобразуются обратно в исходные символы при обращении к свойству `nodeValue` элемента, но эти сущности появятся в исходном коде XML, отправляемом серверу.

Пример

```
myDoc = new XML("<SEARCH_TERM>tutorials</SEARCH_TERM>");
myDoc.send("http://www.domain.com/cgi-bin/lookup.cgi", "remoteWin");
```

См. также

XML.load( ), `XML.loaded`, *XML.onLoad( )*, *XML.sendAndLoad( )*, `XML.status`

XML.sendAndLoad(), метод

Flash 5

отправляет внешнему сценарию или приложению исходный код XML и получает обратно исходный код XML

xmlDoc.sendAndLoad(*url*, *resultXML*)

Аргументы

url Строка, указывающая местонахождение сценария или приложения, которому должен быть передан *xmlDoc*.

resultXML Ссылка на объект документа *XML*, который получит возвращенный исходный код XML.

Описание

Метод *sendAndLoad( )* сериализует *xmlDoc* в строку исходного кода XML и отправляет этот код сценарию или приложению по указанному *url*. Указанный *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Предполагается, что сценарий или приложение некоторым образом обрабатывают XML и возвращают в качестве ответа документ XML. Возвращенный документ перехватывается Flash, анализируется, преобразуется в иерархию объектов XML и помещается в *resultXML*. Прежнее содержимое *resultXML* заменяется вновь загруженным содержимым XML. Важные сведения об отправке XML серверу есть в описании *XML.send( )*. Обратите внимание, что, в отличие от *send( )*, метод *sendAndLoad( )* посылает xmlDoc с помощью запроса POST как в броузере, так и в автономной версии Flash Player 6. Однако в автономном Flash Player 5 под Windows *sendAndLoad( )* использует метод GET. Поддержка GET и POST подробно рассмотрена в приложении F.

Применение

Перед доступом к содержимому, импортируемому *sendAndLoad( )*, необходимо убедиться, что операции загрузки и анализа завершились. Для этого либо проверяется свойство loaded объекта *resultXML*, либо для *resultXML* назначается обработчик события *onLoad( )*, который должен реагировать на завершение загрузки. Подробности см. в описании *XML*.loaded и *XML.onLoad( )*. Результат синтаксического анализа загруженных данных можно узнать, проверив свойство документа status.

На *XML.sendAndLoad( )* распространяются связанные с безопасностью ограничения по домену, описанные для *System.security.allowDomain( )*.

Пример

```
// Создать документ XML
myDoc = new XML("<P>hello server!</P>");

// Создать пустой документ XML, получающий ответ сервера
serverResponse = new XML();

// Отправить myDoc на сервер и поместить ответ в serverResponse
myDoc.sendAndLoad("http://www.domain.com/cgi-bin/readData.cgi", serverResponse);

// Добавить в serverResponse обработчик onLoad(), выводящий ответ
// сервера в текстовом поле output.
serverResponse.onLoad = function () {
  output = serverResponse.toString();
}
```

См. также

XML.load( ), *XML*.loaded, *XML.onData( )*, *XML.onLoad( )*, *XML.send( )*, *XML*.status

XML.status, свойство

Flash 5

указывает, был ли успешным разбор XML в иерархию объектов **только для чтения**

xmlDoc.status

Описание

Свойство `status` возвращает числовой код состояния, указывающий возможные ошибки при синтаксическом анализе исходного кода XML. Указанный `xmlDoc` должен быть узлом верхнего уровня в иерархии объектов XML (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Синтаксический анализ выполняется тогда, когда исходный код XML:

- Передается в качестве аргумента конструктора *XML( )*
- Передается методу *parseXML( )*
- Загружается в новый объект *XML* через методы *load( )* или *sendAndLoad( )*

Коды `status` приведены в табл. C.27. Если при синтаксическом анализе ошибок не было, возвращается `status` со значением 0. Ошибки обозначаются отрицательными числами. Синтаксический анализ завершается при обнаружении первой ошибки, поэтому даже после исправления ошибок, о которых было сообщено, могут обнаружиться новые.

Таблица C.27. Коды статуса синтаксического анализа XML

Статус	Описание
0	Анализ документа не выявил ошибок (то есть успешен)
−2	Секция CDATA завершена неправильно
−3	Объявление XML завершено неправильно
−4	Объявление `DOCTYPE` завершено неправильно
−5	Комментарий завершен неправильно
−6	Элемент XML сформирован неправильно
−7	Недостаточно памяти для синтаксического анализа XML
−8	Значение атрибута завершено неправильно
−9	У начального тега нет соответствующего конечного тега
−10	У конечного тега нет соответствующего начального тега

Обычно с помощью `status` определяют, можно ли приступить к обработке загруженного внешнего файла XML. Проверьте свойство `loaded`, чтобы убедиться в завершении команд *load( )* или *sendAndLoad( )*, прежде чем проверять `status`. Обратите внимание, что анализатор XML в ActionScript не проверяет действительность документов по DTD, а только определяет их корректность.

Пример

```
myDoc = new XML("<BOOK>Colin Moock</AUTHOR></BOOK>");
trace(myDoc.status);  // Выводит: "-10" (отсутствует начальный тег)
```

См. также

XML.load( ), `XML.loaded`, *XML.onLoad( )*, *XML.parseXML( )*, *XML.sendAndLoad( )*

XML.toString(), метод

Flash 5

исходный код узла XML в виде строки

theNode.toString()

Возвращаемое значение

Строка, представляющая исходный код иерархии объекта XML, начинающаяся в *theNode*.

Описание

Метод *toString( )* преобразует объект узла XML или объект документа XML в аналогичный ему исходный код XML (где *theNode* может быть экземпляром *XML* или *XMLnode)*. Если *theNode* представляет собой объект XML верхнего уровня, в строку включаются имеющиеся теги DOCTYPE и объявления XML. Если свойство документа ignoreWhite имеет значение false, сохраняются пробельные символы, а исходный код документа остается таким, каким он был при анализе.

Обратите внимание, что *toString( )* не преобразует предопределенные сущности <, >, &, " и ' в буквальные символы. Текстовое значение узла с преобразованием сущностей в буквальные символы позволяет получить свойство nodeValue. Подробности см. в статье *parseXML( )*.

Обычно нет необходимости явно вызывать *toString( ): toString( )* автоматически вызывается, как только *theNode* встречается в строковом контексте.

Пример

```
var myDoc = new XML('<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd"><BOOK>
<TITLE>ActionScript for Flash MX: The Definitive Guide</TITLE>'
  + '<AUTHOR SALUTATION="Mr.">Colin Moock  </AUTHOR>    '
  + '<PUBLISHER>O\'Reilly & Associates, Inc</PUBLISHER>    </BOOK>');

trace(myDoc.toString());
// Выводит:
<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd">
<BOOK>       <TITLE>ActionScript for Flash MX:
 The Definitive Guide</TITLE><AUTHOR SALUTATION="Mr.">Colin
Moock  </AUTHOR>    <PUBLISHER>O'Reilly & Associates, Inc
</PUBLISHER>    </BOOK>
```

См. также

Object.toString( ), *XML*.nodeValue

XML.xmlDecl, свойство

Flash 5

тег объявления документа XML **чтение/запись**

xmlDoc.xmlDecl

Описание

Строковое свойство xmlDecl представляет тег объявления XML в *xmlDoc*, если таковое существует. В противном случае xmlDecl имеет значение undefined. Документ *xmlDoc* должен быть узлом верхнего уровня в иерархии объектов *XML* (то есть экземпляром класса *XML*, а не класса *XMLnode*).

Тег объявления XML документа XML применяется для указания версии XML, в которой создан документ. Тег объявления XML служит для построения корректных документов XML, действительность которых может быть проверена извне.

Пример

```
// Корректный документ (но без проверки действительности по DTD)
myXML = new XML('<?xml version="1.0"?><P>this is a short document</P>');
trace(myXML.xmlDecl);  // Displays: "<?xml version="1.0"?>"
// Задать новое объявление XML
myXML.xmlDecl = '<?xml version="1.0" standalone="no"?>';
```

См. также *XML*.docTypeDecl

XMLnode, класс

Flash 5 (не документирован)

внутренний надкласс класса XML

Описание

Класс *XMLnode* определяет базовые свойства и методы узлов в иерархии объектов XML, и хотя это внутренний инструмент ActionScript, он может применяться программистами для расширения существующей по умолчанию функциональности объектов XML.

Каждая иерархия объектов XML формально содержит два типа узлов объектов:

- Один узел *XML*, выступающий как главный контейнер иерархии
- Произвольное количество узлов *XMLnode*, являющихся дочерними для главного узла контейнера

Узел главного контейнера является экземпляром класса *XML*. Если создать myDoc так:

```
myDoc = new XML();
```

то myDoc будет экземпляром класса *XML*. Класс *XML* наследует классу *XMLnode*, поэтому узлы главного контейнера обладают всеми свойствами и методами, определенными *XMLnode*, с добавлением тех, которые определены *XML*. Напротив, дочерние элементы myDoc в действительности будут экземплярами класса *XMLnode*, а не класса *XML*.

Поэтому если создать myParagraph так:

```
myParagraph = myDoc.createElement("P");
```

то myParagraph станет экземпляром класса *XMLnode*. Чаще всего внутреннее различие между классами узлов не оказывает воздействия на применение объектов XML. Однако если требуется добавить наследуемое свойство или метод во все объекты XML, то нужно использовать свойство prototype класса *XMLnode*, а не класса *XML* (см. пример ниже). Методы и свойства, прикрепленные к *XMLnode*.prototype, наследуются всеми узлами XML в фильме.

Для справки в табл. С.28 приведены свойства, методы и обработчики событий, определенные в *XMLnode* и *XML*. Обратите внимание, что все перечисленные элементы доступны в экземплярах класса *XML*, но элементы, определенные в *XML*, не доступны для экземпляров *XMLnode*. Например, метод *load()* может быть вызван для экземпляра класса *XML*, но не для экземпляра класса *XMLnode*. Подробное описание каждого элемента можно найти в соответствующем месте описания класса *XML*.

Таблица C.28. Свойства, методы и обработчики событий для XMLnode и XML

XMLnode и XML	только XML
appendChild( )	contentType
attributes	*createElement( )*
childNodes	*createTextNode( )*
cloneNode( )	docTypeDecl
firstChild	*getBytesTotal( )*
hasChildNodes( )	*getBytesLoaded( )*
insertBefore( )	ignoreWhite
lastChild	*load( )*
nextSibling	loaded
nodeName	*onData( )*
nodeType	*onLoad( )*
nodeValue	*parseXML( )*
parentNode	*send( )*
previousSibling	*sendAndLoad( )*
removeNode( )	status
toString( )	xmlDecl

Пример

В следующем коде пользовательский метод *secondChild( )* добавляется в XMLnode.prototype (после этого метод *secondChild( )* становится доступным для всех узлов XML в фильме):

```
XMLnode.prototype.secondChild = function () {
  return this.childNodes[1];
};

myDoc = new XML("<PRODUCT>Cell Phone</PRODUCT><PRODUCT>Game Console</ PRODUCT>");

trace(myDoc.secondChild());  // Displays: "<PRODUCT>Game Console</PRODUCT>"
```

Расширение класса *XML* через *XML*.prototype вполне допустимо, но такое расширение будет применяться только к узлам главных контейнеров (непосредственным экземплярам класса *XML*).

См. также *XML*, класс; «Надклассы и подклассы» в главе 12

XMLSocket, класс

Flash 5

поддержка непрерывного соединения TCP/IP между сервером и клиентом

Конструктор

new XMLSocket()

Методы

close( ) Завершить открытое соединение с приложением сервера.

connect( ) Установить новое соединение с приложением сервера.

send( ) Отправить приложению сервера иерархию объектов XML в виде строки.

Обработчики событий

onClose( ) Выполняется, когда сервер завершает соединение.

onConnect( ) Выполняется, когда попытка соединения завершена.

onData( ) Выполняется, когда данные получены, но еще не проанализированы как XML.

onXML( ) Выполняется, когда данные получены и разобраны в иерархию объектов XML.

Описание

Большинство соединений между Flash и сервером имеют очень короткий срок жизни. Когда Flash запрашивает внешние данные через функции *loadMovie( )*, *loadVariables( )* или *load( )*, устанавливается временный канал связи. По этому каналу передаются данные, после чего он закрывается. Такого рода кратковременные соединения могут иметь полезное применение, но имеют два важных ограничения:

- После закрытия соединения сервер не может связаться с Flash. Инициировать соединение с сервером всегда должен Flash.
- При каждом получении Flash информации с сервера должно открываться новое соединение. Время и нагрузка на процессор, требуемые для многократного открытия соединений, не позволяют Flash и близко подойти к операциям с сервером в реальном масштабе времени.

Начиная с Flash 5 эти ограничения могут быть преодолены с помощью класса *XMLSocket*, который позволяет открыть постоянный канал связи между приложением сервера и Flash. С помощью *XMLSocket* разрабатываются системы, требующие частых обновлений на сервере, например чаты и сетевые многопользовательские игры.

Для того чтобы связаться с удаленным приложением через *XMLSocket*, необходимо сначала создать и сохранить объект *XMLSocket*, например:

```
mySocket = new XMLSocket();
```

После этого вызывается метод *connect( )*, который просит Flash установить канал связи с удаленным приложением, например:

```
mySocket.connect("http://www.myserver.com", 8000);
```

Когда соединение установлено, объект *XMLsocket* действует как приемопередатчик. Данные в формате XML посылаются удаленному приложению с помощью метода сокета *send( )*, а о получении данных в формате XML можно узнать, когда генерируется событие сокета *onXML( )*.

Приложение сервера, взаимодействующее с объектом *XMLSocket*, должно:

- Обслуживать соединения сокетов TCP/IP на конкретном порту с номером, большим или равным 1024.

- Передавать данные в формате XML сегментами, разделяемыми *нулевым байтом* (то есть символом ASCII null).

Приложения сервера обычно создаются не программистами Flash, а программистами сервера. Некоторые из имеющихся Flash-совместимых серверов сокетов перечислены в табл. C.29.

Таблица C.29. Flash-совместимые серверы сокетов

Название	Описание	Вид лицензии	Адрес
moockComm	Облегченная контрольная реализация на Java	Бесплатный	*http://www.moock.org/chat/moockComm.zip*
aquaServer	Облегченная контрольная реализация на Java	Бесплатный	*http://www.figleaf.com/development/flash5/aquaserver. zip*
Unity	Полнофункциональный rooms-based-сервер на Java	Платный	*http://www.moock.org/unity*
Fortress	Развитый мощный сервер сокетов	Платный	*http://www.xadra.com/products/main.html*
ElectroServer	Полнофункциональный прошедший испытание практикой сервер	Платный	*http://www.electrotank.com/ElectroServer*

Соединение *XMLSocket* остается открытым, пока не произойдет одно из следующих событий:

- Вызван метод *close( )* объекта *XMLSocket*.
- Ссылки на объект *XMLSocket* перестали существовать.
- Сервер закрывает соединение (при этом генерируется событие *onClose( )*).
- Фильм выгружен или Flash Player завершил работу.

Класс *XMLSocket* предоставляет также четыре обработчика событий – *onClose( )*, *onConnect( )*, *onData( )* и *onXML( )*, позволяющие следить за состоянием открытого соединения.

Если не определить обработчики обратного вызова для свойств `onClose` и `onConnect`, то нельзя будет осуществлять проверку ошибок или обеспечить какую-либо обратную связь при установлении соединения. Если не определить обработчик обратного вызова для свойства `onXML`, то нельзя будет получить уведомление при получении сокетом данных от приложения на стороне сервера, а также извлечь эти данные.

Обширная информация о разработке *XMLSocket* есть по адресам:

http://www.moock.org/unity

http://hotwired.lycos.com/webmonkey/multimedia/shockwave_flash/tutorials/tutorial9. html

Следует учесть, что начиная с Flash 6 связь в реальном времени может также быть реализована с помощью Macromedia Flash Communication Server MX (Comm Server). Comm Server использует не XML, а особый протокол RMTP для удобного выполнения

удаленного кода ActionScript. Кроме того, Comm Server поддерживает потоковые видео и аудио. Подробнее об этом сказано по адресу:

http://www.macromedia.com/software/flashcom/

Пример

В примере C.10 определен подкласс *XMLSocket* с именем *SocketController*, создающий для работы с классом *XMLSocket* ряд удобных методов и обработчиков событий. Класс записан в файле *SocketController.as* и может быть загружен из сетевого хранилища кода. В примере C.11 показано, как с помощью *SocketController* реализовать простое чат-приложение.

Пример C.10. Класс SocketController

```
/*
 * Класс SocketController. Расширяет XMLSocket.
 *   Версия: 1.0.0
 * Описание: Предоставляет средства для связи с сервером сокетов.
 *
 * Методы
 *  doConnect()             -соединиться с сервером
 *  setServer()             -задать имя хоста и порт сервера
 *  killConnect()           -отсоединиться от сервера
 *  send()                  -послать серверу XML
 *
 * Обработчики событий
 *  onConnectFailure()      -когда попытка соединения удачна
 *  onConnectSuccess()      -когда попытка соединения не удачна
 *  onServerKillConnect()   -когда сервер закрывает соединение
 *  onClientKillConnect()   -когда Flash закрывает соединение
 *  onXML()                 -когда поступает новый XML
 */

// Сделать XMLSocket надклассом.
SocketController.prototype = new XMLSocket();

// Конструктор класса
function SocketController (host, port) {
  // Свойства экземпляра
  this.host = null;
  this.port = 0;

  // Инициализация: присвоить переданные значения свойствам host и port.
  if (arguments.length > 1) {
    this.setServer(host, port);
  }
}

// Методы экземпляра
/*
 * Попытка соединиться с сервером.
 */
SocketController.prototype.doConnect = function () {
  // Если не указан допустимый сервер, то отказ
  if (this.host == null || this.port < 1024) {
    trace("Connection attempt aborted: invalid host or port.");
    this.onConnectFailure();
    return false;
```

```
  } else {
    // Иначе попытка соединиться (connect возвращает true, если начало соединения успешное).
    var connectSuccess = this.connect(this.host, this.port);
  }

  // Если начальный этап соединения успешен...
  if (connectSuccess) {
    // ...ждать возникновения события onConnect().
    trace("Initial connection succeeded. Awaiting server response...");
  } else {
    // ...иначе connect() возвратил false, и соединение не было установлено.
    trace("Initial connection failed");
    // Вызвать пользовательскую функцию обратного вызова для обработки неудачи соединения.
    this.onConnectFailure();
  }
};

/*
 * Задать имя хоста и порт сервера.
 */
SocketController.prototype.setServer = function (host, port) {
  // Обработка ошибок. Host должен быть строкой, а port - числом, большим 1023.
  if (typeof host != "string"
      || typeof port != "number"
      || (typeof port == "number" && port < 1024)) {
    trace("Server assignment failed: invalid host or port.");
    // Сбросить host и port.
    this.host = null;
    this.port = 0;
    return false;
  }

  // Host и port допустимы, поэтому присвоить их.
  trace("Server set to: " + host + " " + port);
  this.host = host;
  this.port = port;
};

/*
 * Функция обратного вызова, отвечающая на завершение попытки соединения.
 */
SocketController.prototype.onConnect = function (success) {
  // Если аргумент success функции onConnect() равен true, соединение установлено.
  if (success) {
    trace("Connection to: " + this.host + " " + this.port + " established");
    // Вызвать пользовательскую функцию обратного вызова для обработки успешного соединения.
    this.onConnectSuccess();
  } else {
    // success равен false, поэтому соединение не было установлено.
    trace("Connection to: " + this.host + " "
                    + this.port + " could not be established");
    // Вызвать пользовательскую функцию обратного вызова для обработки неудачи соединения.
    this.onConnectFailure();
  }
};

/*
 * Функция обратного вызова, вызываемая при закрытии сервером соединения.
 */
```

```
SocketController.prototype.onClose = function () {
  trace("The server has terminated the connection.");
  // Вызвать пользовательскую функцию обратного вызова для обработки
  // закрытия сервером соединения
  this.onServerKillConnect();
};

/*
 * Закрывает соединение с сервером вручную.
 */
SocketController.prototype.killConnect = function () {
  trace("Closing current connection.");
  this.close();
  // Вызвать пользовательскую функцию обратного вызова для обработки
  // закрытия соединения клиентом
  this.onClientKillConnect();
};
```

Пример С.11 демонстрирует применение *SocketController* в простом приложении чата.

Пример С.11. Простое приложение чата на базе SocketController

```
// Простое приложение чата
// Загрузить класс SocketController
#include "SocketController.as"
// Начало шоу
main();

// Приложение чата
function main () {
  // Запомнить переносимую ссылку на текущую временную диаграмму
  _global.chatMain = this;

  // Создать текстовое поле для посылаемого текста
  chatMain.createTextField("send_txt", 1, 100, 100, 300, 20);
  chatMain.send_txt.border = true;
  chatMain.send_txt.type = "input";

  // Создать текстовое поле output
  chatMain.createTextField("output_txt", 2, 100, 130, 300, 200);
  chatMain.output_txt.border = true;
  chatMain.output_txt.type = "dynamic";

  // Создать экземпляр SocketController и соединиться с сервером
  chatController = new SocketController();
  chatController.setServer("moock.org", 8050);
  chatController.doConnect();

  // Вызывается при получении сообщения
  chatController.onXML = function (src) {
    trace("Received XML: " + src);
    var message = src.firstChild.firstChild.nodeValue;
    chatMain.output_txt.text +=  message + "\n";
  };
  // Вызывается при неудаче соединения
  chatController.onConnectFailure = function () {
    chatMain.output_txt.text +=  "Connection attempt failed." + "\n";
  };

  // Вызывается при успешном соединении
  chatController.onConnectSuccess = function () {
```

```
    chatMain.output_txt.text +=  "Connection established." + "\n";
  };

  // Вызывается при закрытии соединения сервером
  chatController.onServerKillConnect = function () {
    chatMain.output_txt.text +=  "Connection terminated by server." + "\n";
  };

  // Вызывается при закрытии соединения Flash
  chatController.onClientKillConnect = function () {
    chatMain.output_txt.text +=  "Connection closed." + "\n";
  };

  // Добавить приемник для перехвата нажатий клавиши Enter
  chatController.onKeyDown = function () {
    // Если клавиша Enter нажата, когда курсор был в поле send...
    if (Key.getCode() == Key.ENTER
        && eval(Selection.getFocus()) == chatMain.send_txt) {
      // ...отправить содержимое поля send в теге <MESSAGE>
      chatController.send("<MESSAGE>" + chatMain.send_txt.text + "</MESSAGE>");

      // Очистить поле ввода
      chatMain.send_txt.text = "";
    }
  };
  // Зарегистрировать приемник
  Key.addListener(chatController);
}
```

См. также *loadVariables( )*, класс *LoadVars*, класс *XML*

XMLSocket.close(), метод

Flash 5

завершает открытое соединение с приложением сервера

socket.close()

Описание

Метод *close( )* разрывает канал связи между *socket* и приложением сервера. После выполнения *close( )* над *socket* попытки вызова *send( )* с *socket* оказываются безуспешными. Аналогично приложение сервера более не сможет отправлять данные Flash через *socket*.

Обратите внимание, что *close( )* никак не действует, если *socket* уже закрыт или ранее не был соединен. Кроме того, *close( )* не запускает обработчик *onClose( )* объекта сокета: *onClose( )* генерируется только закрытием соединения со стороны сервера.

См. также *XMLSocket.connect( )*, *XMLSocket.onClose( )*, *XMLSocket.send( )*

XMLSocket.connect(), метод

Flash 5

открывает соединение с приложением сервера

socket.connect(*host*, *port*)

Аргументы

host Строка, задающая имя хоста в виде "www.myserver.com" или стандартного IP-адреса (четырех 8-битовых разделенных точками десятичных чисел, например 111.222.3.123). Если указан null или пустая строка, то значением по умолчанию считается адрес сервера, с которого был получен фильм.

port Целое число, задающее номер порта TCP, больший или равный 1024. Стандартного или общепринятого номера порта для связи через *XMLSocket* нет.

Возвращаемое значение

Булево значение, указывающее на начальный успех (true) или неудачу (false) попытки соединения.

Описание

Метод *connect( )* пытается установить соединение между Flash и приложением сервера, выполняющимся на порту *port* машины *host*.

Если *connect( )* возвращает true, то начальная фаза соединения успешно завершена и позднее будет запущен обработчик *onConnect( )* для *socket*. В обработчике *onConnect( )* можно определить, полностью ли установлено соединение. Учтите, что попытки соединения могут иметь различную продолжительность, особенно когда попытка соединения оказывается неудачной. При вызове *connect( )* следует показывать пользователю, что идет процесс установления соединения.

Если *connect( )* возвращает false, значит, начальная фаза установки соединения не была успешно завершена. В этом случае обработчик *onConnect( )* для *socket* не вызывается.

Важно проверять как значение, возвращаемое методом *connect( )*, так и значение параметра *success* обработчика *onConnect( )*, если *connect( )* возвращает true.

Применение

По соображениям безопасности не разрешается подключение с помощью *connect( )* к произвольным хостам в Интернете. Подключение возможно только к хостам в домене, из которого был загружен фильм. Правила доступа *connect( )* к другим доменам описаны в статье *System.security.allowDomain( )* данного справочника. Метод *connect( )* возвращает false при попытках соединений, нарушающих ограничения безопасности. Обратите внимание, что ограничения, связанные с безопасностью, не действуют в автономном проигрывателе.

Пример

```
// Создать новый объект сокета
mySocket = new XMLSocket();
// Присвоить onConnect функцию обработчика обратного вызова
mySocket.onConnect = handleConnect;
// Попытка соединиться с приложением, выполняемым на myserver.com на порту 10000
if (mySocket.connect("myserver.com", 10000) == false) {
  // Переход в кадр, выводящий сообщение об ошибке
  this.gotoAndStop("failureDialog");
} else {
  // Переход в кадр, где будет ожидаться генерирование onConnect
  this.gotoAndPlay("connecting");
}
```

См. также *XMLSocket.close( )*, *XMLSocket.onConnect( )*

XMLSocket.onClose(), обработчик события

Flash 5

вызывается при закрытии соединения сервером

socket.onClose*()*

Описание

Обработчик события *onClose( )* выполняется при закрытии сервером (а не Flash) соединения с *socket*. Инициируемое сервером закрытие обычно происходит при завершении работы приложения сервера или преднамеренном «выкидывании» клиента.

Пример

Для того чтобы обеспечить реакцию на событие *onClose( )*, следует присвоить свою функцию (то есть собственный обработчик обратного вызова) свойству *onClose* объекта *XMLSocket*. Этот обработчик обратного вызова применяется для обнаружения внешнего рассоединения с сокетом. В следующем коде функция *handleClose( )* присваивается свойству *onClose* объекта *mySocket*. Функция *handleClose( )* просто извещает пользователя о том, что произошло закрытие соединения, обновляя значение текстового поля status_txt:

```
mySocket = new XMLSocket();
mySocket.onClose = handleClose;

function handleClose () {
  status_txt.text += ("\nThe server has terminated the connection.\n");
}
```

См. также *XMLSocket.close( )*; глава 10

XMLSocket.onConnect(), обработчик события

Flash 5

вызывается по завершении попытки соединения (успешной или нет)

socket.onConnect*(success)*

Аргументы

success Булево значение, указывающее, успешной (true) или неудачной (false) была попытка соединения.

Описание

Обработчик события *onConnect( )* вызывается при завершении ранее вызванной операции *connect( )*. Выполнение *onConnect( )* не обязательно означает, что соединение было успешно установлено: обработчик обратного вызова выполняется тогда, когда завершена *попытка* соединения, вне зависимости от ее успеха. Обработчику обратного вызова, заданному в *onConnect( )*, передается аргумент *success*, указывающий на успешность попытки (то есть на то, что соединение установлено). Если это так, *success* получает значение true. Если попытка была неудачной (то есть произошел тайм-аут соединения, отказ в соединении или было невозможно установить соединение по другим причинам), то аргумент *success* получает значение false. Обратите внимание, что ActionScript не делает различия между тайм-аутом сети, неизвестным хостом, отказом в доступе или другими обычными ошибками соединения. Поэтому выполнение обработчика обратного вызова может происходить с задержкой до мину-

ты после подачи команды *connect( )* в зависимости от настроек сервера, попытка соединения с которым осуществляется, скорости соединения, сетевого трафика и т. д.

Пример

С помощью *onConnect( )* можно определить успех или неудачу попытки соединения. На практике он может устанавливать флаг, указывающий на необходимость начать передачу, если соединение оказалось успешным. Обработчик обратного вызова позволяет также выполнять код, нейтрализующий ошибку неудачного соединения, например, сообщающий пользователю о сущности проблемы.

Для того чтобы ответить на событие *onConnect( )*, надо присвоить свою функцию (то есть обработчик обратного вызова) свойству `onConnect` объекта *XMLSocket*. В следующем коде функция *handleConnect( )* присваивается свойству `onConnect` объекта `mySocket`. Путем обновления текстового поля `status_txt` функция *handleConnect( )* извещает пользователя об успехе или неудаче соединения:

```
mySocket = new XMLSocket();
mySocket.onConnect = handleConnect;

function handleConnect (succeeded) {
  if (succeeded) {
    status_txt.text += ("Successfully connected.\n");
  } else {
    status_txt.text += ("Connection attempt failed.\n");
  }
}
```

Код, иллюстрирующий применение обработчика *onConnect( )* в более полной системе, можно найти в примере C.11 для класса *XMLSocket*.

См. также *XMLSocket.connect( )*, *XMLSocket.onClose( )*; глава 10

XMLSocket.onData(), обработчик события

Flash 5

выполняется, когда внешние данные получены, но еще не разобраны как XML

socket.onData(*src*)

Аргументы

src Строка, содержащая загруженные данные, обычно являющиеся исходным кодом XML.

Описание

Обработчик *onData( )* выполняется, когда во Flash через *socket* передан нулевой байт (символ ASCII null). По умолчанию *onData( )* просто создает из *src* новую иерархию объектов XML и передает эту иерархию в *socket.onXML( )*. Однако обработчику *onData( )* может быт присвоена пользовательская функция обратного вызова, перехватывающая *src* до того, как ActionScript может проанализировать его как XML. В некоторых случаях (например, видеоиграх реального времени) обработка вручную данных в *src* может повысить производительность по сравнению со встроенным в ActionScript синтаксическим анализом.

Пример

Следующий код показывает, как присвоить *onData( )* пользовательскую функцию обратного вызова. Эта функция обратного вызова просто выводит любые данные, полученные *mySocket*, и не дает ActionScript анализировать данные как XML:

```
mySocket = new XMLSocket();

mySocket.onData = function (src) {
  trace("Received data: \n" + src);
};
```

См. также *XMLSocket.onXML( )*

XMLSocket.onXML(), обработчик события

Flash 5

вызывается, когда данные получены объектом XMLSocket и проанализированы как XML

socket.onXML*(XMLobject)*

Аргументы

XMLobject

Объект XML, который будет хранить поступившие в формате XML данные.

Описание

Обработчик события *onXML( )* выполняется тогда, когда Flash получил данные и выполнил их синтаксический анализ. Когда *socket* получает от сервера законченный блок данных (то есть строку, за которой следует символ ASCII `null`), автоматически вызывается обработчик обратного вызова, заданный в *socket*.onXML. Сервер может посылать данные с любой частотой, но обработчик обратного вызова выполняется только тогда, когда *socket* в конце получает символ `null` (то есть нулевой байт). В Java нулевой байт задается как `'\0'`.

При получении нулевого байта ActionScript анализирует все данные, полученные *socket* с момента отправки последнего нулевого байта (или начального установления связи, если это первый нулевой байт). Проанализированные данные преобразуются в иерархию объектов XML, которая передается как аргумент *XMLobject* функции обратного вызова.

Если как Flash-программист вы отвечаете только за клиентскую часть приложения клиент/сервер, просто отметьте себе, что обработчик обратного вызова, заданный `onXML`, получает все новые данные XML по их прибытии. Доступ к новым данным XML осуществляется через *XMLobject*.

Для доступа к необработанным данным, пересылаемым через сокет, переопределите действие по умолчанию обработчика события сокета *onData( )* (см. *XMLSocket.onData( )*).

Пример

Для того чтобы ответить на событие *onXML( )*, надо присвоить свою функцию обратного вызова свойству `onXML` объекта *XMLSocket*. В следующем коде функция *handleIncoming( )* присваивается свойству `onXML` объекта `mySocket`. Функция *handleIncoming( )* обращается к одному из узлов иерархии объектов XML, хранящейся в `messageObj`, и добавляет его значение к текстовому полю `messages_txt`:

```
mySocket = new XMLSocket();
mySocket.onXML = handleIncoming;
```

```
function handleIncoming (messageObj) {
  trace("Got some new data!");
  // messageObj будет содержать фрагмент: <MESSAGE>text</MESSAGE>
  var message = messageObj.firstChild.firstChild;
  messages_txt.text += (message.nodeValue + "\n");
}
```

Код, иллюстрирующий применение обработчика *onXML( )* в более полной системе, можно найти в примере C.11 для класса *XMLSocket*.

См. также *XMLSocket.send*, *XMLSocket.onData( )*; глава 10

XMLSocket.send(), метод

Flash 5

передает данные в формате XML приложению сервера

socket.send(*XMLobject*)

Аргументы

XMLobject

Объект *XML*, который должен быть преобразован в строку и передан приложению сервера, или любая строка, содержащая текст в формате XML.

Описание

Метод *send( )* передает сообщение из Flash приложению сервера через *socket*. Передаваемое сообщение должно быть экземпляром класса *XML*, но может также являться строкой в формате XML. При вызове *send( )* объект *XMLobject* преобразуется в строку и передается удаленному приложению, завершаясь нулевым байтом (символом ASCII null). Удаленное приложение не обязано дать ответ; однако если ответ прислан, он запускает обработчик события *onXML( )* сокета *socket*.

Пример

В следующем примере удаленному приложению посылается очень простое сообщение в формате XML через сокет *mySocket*, являющийся действующим объектом *XMLSocket*, для которого уже установлено соединение (обратите внимание, что message является объектом *XML*, а не объектом *XMLSocket*; см. в примере C.11 в статье о классе *XMLSocket* образец законченного приложения *XMLSocket*):

```
var message = new XML('<MESSAGE>testing...testing...</MESSAGE>');
mySocket.send(message);
```

Допускается также передавать строку, содержащую текст в формате XML, не оборачивая ее в объект *XML*. Для простых сообщений XML этого часто бывает достаточно:

```
mySocket.send('<MESSAGE>testing...testing...</MESSAGE>');
```

См. также *XMLSocket.onXML*, *XML.send( )*; класс *XMLSocket*

III

Приложения

- Приложение A «Ресурсы»
- Приложение B «Набор символов Latin 1 и коды клавиш»
- Приложение C «Обратная совместимость и обновление версий проигрывателя»
- Приложение D «Отличия от ECMA-262 и JavaScript»
- Приложение E «Поддержка HTML в текстовых полях»
- Приложение F «Поддержка GET и POST»
- Приложение G «Обзор компонентов Flash UI»
- Приложение H «Встраивание фильмов Flash в веб-страницы»

A

Ресурсы

Ниже приводятся некоторые небезынтересные ресурсы. См. также URL, перечисленные в табл. B.3 введения.

Книги по ActionScript

Среди сотен имеющихся в продаже книг по Flash и ActionScript, на мой взгляд, можно выделить очень немногие:

- «Object-Oriented Programming with ActionScript» (Объектно-ориентированное программирование на ActionScript) Брэндена Холла (Branden Hall) и Сэмюэла Уэна (Samuel Wan), New Riders, 2002 г.
- «Flash MX Application Design and Development» (Проектирование и разработка приложений во Flash MX) Джессики Шпигель (Jessica Speigel), New Riders, 2002.
- «Robert Penner's Programming Macromedia Flash MX» (Программирование в Macromedia Flash MX) Роберта Пеннера (Robert Penner), Osborne McGraw-Hill, 2002.

ActionScript и веб-сайты для программистов

Сайт книги «ActionScript for Flash MX: The Definitive Guide»

http://www.moock.org/asdg

Отсюда можно загрузить все примеры из этой книги (хранилище кода Code Depot), а также десятки других полезных примеров ActionScript. Кроме того, там есть обновления к книге, статьи, некоторые главы, обнаруженные ошибки и новости. Поддерживается автором.

moockmarks – авторские закладки

http://www.moock.org/moockmarks

Перечислены сотни сайтов – источников технической информации и вдохновения разработчика для Flash и веб-дизайна в целом.

Центр Macromedia по поддержке Flash

http://www.macromedia.com/support/flash

Быстро растущее собрание информации по ActionScript и Flash. Содержит полный текст Macromedia's ActionScript Dictionary и регулярно обновляемую базу данных технических заметок для разработчиков Flash. Начинать поиск ответов на возникающие вопросы можно здесь. Демонстрационную версию среды разработки Flash можно загрузить с *http://www.macromedia.com/ software/flash/trial*.

Macromedia Exchange for Flash

http://www.macromedia.com/exchange/flash

Собрание компонентов профессионального качества, файлов проектов *.fla* и примеров ActionScript. Размещается на сервере Macromedia и поддерживается ею, но представляет труды сообщества Flash-разработчиков. Доступ к примерам требует наличия утилиты установки под названием Extension Manager, которую можно взять там же.

Flash Kit

http://www.flashkit.com

Flash Kit, поддерживаемый *internet.com*, представляет собой крупнейший сторонний информационный сайт по Flash. Там есть все, что может иметь отношение к Flash: файлы примеров, доски объявлений, учебники, новости, статьи, интервью, чат, галерея и многое другое. К сожалению, в огромном объеме бывает трудно сориентироваться. Особый интерес для программистов Flash представляют *.fla*-файлы примеров ActionScript, которые можно загрузить с *http:// www.flashkit.com/movies/Scripting*.

Ultrashock

http://www.ultrashock.com

Общий портал для Flash-разработчиков. Ultrashock активно поддерживает собрание *.fla*-файлов и учебников, посвященных ActionScript, а также интенсивно действующий форум разработчиков.

We're Here

http://www.were-here.com

Большая и активная группа досок объявлений во Всемирной паутине для обсуждения Flash и ActionScript. Имеются также статьи, примеры *.fla*-файлов, учебники и новости Flash.

Почтовый список рассылки кодировщиков Flash

http://chattyfig.figleaf.com

Руководимый Брэнденом Холлом (Branden Hall), этот почтовый список рассылки Flash-разработчиков особое внимание уделяет ActionScript средней и повышенной сложности. Это хорошее место для изучения практических технологий и общения с коллегами, но *не* предназначенное для общих вопросов по Flash или проверок сайтов. Возможен поиск в архи-

вах, которые пополняются ежемесячно. Прежде чем посылать вопросы, поищите ответы в архивах.

Brandon Williams's and Ethan Kennedy's Flash Experiments

http://www20.brinkster.com/ahab/flash5

Большая коллекция сценариев ActionScript, демонстрирующих сложные математические методы. Для изучения доступны десятки *.fla*-файлов с примерами, в которых применяется 3D-графика, фракталы и многие зрительные физические эффекты. Содержит глубокие статьи, посвященные использованию математики при программировании во Flash.

Система компонентов Flash MX 3D

http://www.benchun.net/mx3d

Компоненты 3D-ядра для Flash Player 6. Автор – Бен Чан (Ben Chun), выпускник Массачусетского технологического и инженер Macromedia.

Flash API

http://www.yestoall.com/flashAPI

Набор библиотек для разработки на Flash. Содержит визуальные инструменты (наклон и вращение), дополнения объектов (управление панорамированием звука) и утилиты (таймеры, загрузчики, вспомогательные уравнения и средства расчета расстояний).

[PROTO]type

http://www.layer51.com/proto

Общественный веб-сайт для размещения расширений встроенных классов ActionScript (иными словами, новые методы и свойства для таких объектов, как *String*, *Array* и *XML*). Представлены сотни расширений, многие из которых очень полезны.

FAQ comp.graphics.algorithms

http://www.faqs.org/faqs/graphics/algorithms-faq

Описывает математические решения широкого круга задач программирования графики. Вопросы могут быть как достаточно простыми (например, как найти расстояние между точкой и прямой), так и сложными, относящимися к 3D-графике и обычно выходящими за рамки программирования на ActionScript.

Archived Macromedia Flash Players

http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm

Загрузка для тестирования старых версий Macromedia Flash Player.

Общие ресурсы для программирования игр

На перечисленных ниже сайтах представлена полезная информация для проектирования, планирования и программирования игр (эти сайты не посвящены конкретно ActionScript):

http://www.gamedev.net
http://www.flipcode.com
http://www.javascript-games.org

Flash и Active Server Pages (ASP)

Следующие статьи из ASP 101 дают хорошее введение в интеграцию Flash с ASP для создания простых форм Flash:

http://www.asp101.com/articles/flash
http://www.asp101.com/articles/flash2

Macromedia Flash Communication Server MX (Comm Server)

Comm Server API (например, *Remote SharedObject*, *Camera*, *Microphone*, *NetConnection* и *NetStream*) используется для создания многопользовательских веб-приложений. Сведения о передаче в реальном времени данных, аудио и видео с Comm Server во Flash можно найти здесь:

http://www.macromedia.com/software/flashcom
http://www.flashcom.info

Macromedia Flash Remoting MX (Flash Remoting)

Flash Remoting позволяет использовать Flash как клиента удаленных веб-приложений и веб-сервисов, включая созданные с помощью ColdFusion MX. Дополнительные сведения о Flash Remoting можно найти здесь:

http://www.macromedia.com/software/flash/flashremoting

Ресурсы ECMA-262

Как ActionScript, так и JavaScript основаны на спецификации языков сценариев ECMA-262, поэтому их базовый синтаксис одинаков. И поэтому следующие ресурсы, хотя они не посвящены конкретно ActionScript, будут полезны тем, кто на нем программирует:

Спецификация ECMA-262

http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

Официальная спецификация языка ECMA-262, разработанная ECMA, организацией, поддерживающей язык, на котором основан ActionScript. Другими реализациями ECMA-262 являются JavaScript, разработанный Netscape, и Jscript, разработанный Microsoft. Этот насыщенный техническими подробностями документ может пригодиться в таких областях, как создание интерпретаторов, удовлетворяющих ECMA.

Руководство Netscape по базовому JavaScript

http://developer.netscape.com/docs/manuals/js/core/jsguide/contents.htm

В этой очень понятной документации по JavaScript от Netscape подробно описаны базовые функции языка JavaScript.

Объектно-ориентированное программирование

Описание объектной модели Netscape

http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm

Статья, излагающая реализацию ООП на основе прототипов, наследование и классы в JavaScript. Теория иллюстрируется многочисленными примерами кода. Netscape старается описывать ООП в JavaScript в терминах, понятных тем, кто программирует на C и Java.

Основные понятия объектно-ориентированного программирования от Sun (из The JavaTM Tutorial)

http://java.sun.com/docs/books/tutorial/java/concepts

Этот сайт предназначен для начинающих программистов на Java, но рассказывает об основах ООП вообще и полезен для получения подготовки по ООП при программировании на любом языке.

Формат файла SWF

Код Flash Player не является открытым, но сам формат файла *.swf* общедоступен, что послужило одной из причин получения им широкой поддержки сторонних разработчиков. Вот некоторые источники дополнительной информации:

Macromedia Flash file format (SWF)

http://www.macromedia.com/software/flash/open/licensing/fileformat

Здесь можно скачать SWF SDK Macromedia – «набор инструментов для разработчиков, позволяющий писать файлы Macromedia Flash (SWF), документация о формате файлов Macromedia Flash (SWF) и код для записи файлов SWF».

OpenSWF

http://www.openswf.org

Центр общей информации о реализациях SWF. Содержит учебники, доски объявлений, ресурсы и исходный код.

Ming

http://ming.sourceforge.net

Библиотека C с открытым исходным кодом для создания фильмов в формате SWF, а также набор оболочек для использования этой библиотеки в C++ и популярных языках сценариев, таких как PHP, Python и Ruby. Информацию о программном обеспечении с открытым исходным кодом можно найти на *http://www.opensource.org*.

Perl :: Flash

http://www.2shortplanks.com/flash

Библиотека Perl для «динамичного программируемого создания фильмов Flash/SWF». Разработчик – Саймон Уистоу (Simon Wistow).

Кодировки символов и Unicode

Диапазоны символов Unicode и шрифты Unicode, которые их поддерживают

http://www.alanwood.net/unicode/fontsbyrange.html

Отличное место, где можно найти шрифт для отображения конкретного диапазона символов Unicode.

Поддержка Unicode и многоязычности в HTML, шрифтах, броузерах и других приложениях

http://www.alanwood.net/unicode

Неплохая подборка сведений об использовании Unicode во Всемирной паутине.

Краткое введение в кодовые страницы и Unicode

http://www-106.ibm.com/developerworks/library/codepages.html

Краткий, но информативный обзор развития кодировок символов.

Учебник по вопросам кодировки символов

http://www.cs.tut.fi/~jkorpela/chars.html

Весьма информативное собрание статей по основам представления символов в компьютерах. Автор – Юкка Корпела (Jukka Korpela).

B

Набор символов Latin 1 и коды клавиш

В табл. B.1 перечислены символы набора Latin 1 с их эквивалентами в Unicode в диапазоне от U+0000 до U+00FF (то есть C0 Controls, Basic Latin, C1 Controls и Latin 1 Supplemental). Первая колонка таблицы (*Дес.*) показывает кодовую позицию каждого символа в десятичном виде (стандартное значение ASCII или Latin 1), во второй колонке приводится escape-последовательность Unicode для этого символа, а в третьей колонке показан или описан сам символ. Дополнительные сведения о кодировке символов в Flash см. в главе 4. Таблицы B.2 и B.3 перечисляют коды клавиш, используемые объектом *Key*. Несмотря на некоторое перекрытие, коды клавиш представляют физические клавиши на клавиатуре и не имеют прямого отношения к символам Latin 1 из табл. B.1. Коды клавиш для букв и цифр, перечисленные в табл. B.3, совпадают с кодовыми позициями в Latin 1 (и значениями ASCII) для букв A–Z и цифр 0–9. Дополнительные сведения о кодах клавиш можно найти в описаниях метода *Key.getCode( )* и объекта *Key* в «Справочнике по языку».

Дополнительные материалы, посвященные кодировкам символов, можно найти в следующих ресурсах:

Учебное руководство по проблемам кодировки символов

http://www.cs.tut.fi/~jkorpela/chars.html

Весьма информативное собрание статей по основам представления символов в компьютерах. Автор — Jukka Korpela.

FAQ по Unicode

http://www.unicode.org/unicode/faq

Хороший обзор по Unicode – международному стандарту кодировки символов – в виде вопросов и ответов.

Найти кодовую позицию Unicode символа, отсутствующего в Latin 1, можно здесь:

Vendor Mappings

http://www.unicode.org/Public/MAPPINGS/VENDORS

Показывает кодовые страницы конкретных производителей с соответствующими кодовыми позициями Unicode. Если вы знаете, какая кодовая

страница используется для ввода текста в вашем компьютере, вы узнаете, какие символы можно вводить с клавиатуры.

Where is my character?

http://www.unicode.org/unicode/standard/where

Полезный документ, предлагающий способы найти символ в наборе Unicode.

Unicode Code Charts

http://www.unicode.org/charts

Ряд файлов *.pdf*, в которых описаны и показаны символы Unicode.

Unihan Database

http://www.unicode.org/charts/unihan.html

Часть Unicode Character Database, в которой есть кодовые позиции многих символов азиатских языков, в том числе в кодировке Shift-JIS, которую поддерживает Flash 5 для символов японского языка.

Таблица В.1. Символы ISO 8859-1 (Latin 1) и их эквиваленты в Unicode

Дес.	Unicode	Описание	Дес.	Unicode	Описание
0	\u0000	[null]	20	\u0014	[управление устройством 4]
1	\u0001	[начало заголовка]	21	\u0015	[отрицание]
2	\u0002	[начало текста]	22	\u0016	[синхронизация]
3	\u0003	[конец текста]	23	\u0017	[конец передачи]
4	\u0004	[конец передачи]	24	\u0018	[аннулирование]
5	\u0005	[запрос]	25	\u0019	[конец носителя]
6	\u0006	[подтверждение]	26	\u001a	[замена]
7	\u0007	[звонок]	27	\u001b	[переключение кода]
8	\u0008	[забой]	28	\u001c	[разделитель файлов]
9	\u0009	[горизонтальная табуляция]	29	\u001d	[разделитель групп]
10	\u000a	[перевод строки]	30	\u001e	[разделитель записей]
11	\u000b	[вертикальная табуляция]	31	\u001f	[разделитель элементов]
12	\u000c	[подача листа]	32	\u0020	[пробел]
13	\u000d	[возврат каретки]	33	\u0021	!
14	\u000e	[расширение кода]	34	\u0022	"(прямые кавычки)
15	\u000f	[восстановление кода]	35	\u0023	#
16	\u0010	[переключение]	36	\u0024	$
17	\u0011	[управление устройством 1]	37	\u0025	%
18	\u0012	[управление устройством 2]	38	\u0026	&
19	\u0013	[управление устройством 3]	39	\u0027	' (апостроф)

Дес.	Unicode	Описание	Дес.	Unicode	Описание
40	\u0028	(	74	\u004a	J
41	\u0029	)	75	\u004b	K
42	\u002a	*	76	\u004c	L
43	\u002b	+	77	\u004d	M
44	\u002c	, (запятая)	78	\u004e	N
45	\u002d	– (минус)	79	\u004f	O
46	\u002e	. (точка)	80	\u0050	P
47	\u002f	/	81	\u0051	Q
48	\u0030	0	82	\u0052	R
49	\u0031	1	83	\u0053	S
50	\u0032	2	84	\u0054	T
51	\u0033	3	85	\u0055	U
52	\u0034	4	86	\u0056	V
53	\u0035	5	87	\u0057	W
54	\u0036	6	88	\u0058	X
55	\u0037	7	89	\u0059	Y
56	\u0038	8	90	\u005a	Z
57	\u0039	9	91	\u005b	[
58	\u003a	:	92	\u005c	\
59	\u003b	;	93	\u005d	]
60	\u003c	<	94	\u005e	^
61	\u003d	=	95	\u005f	_ (подчеркивание)
62	\u003e	>	96	\u0060	` (гравис)
63	\u003f	?	97	\u0061	a
64	\u0040	@	98	\u0062	b
65	\u0041	A	99	\u0063	c
66	\u0042	B	100	\u0064	d
67	\u0043	C	101	\u0065	e
68	\u0044	D	102	\u0066	f
69	\u0045	E	103	\u0067	g
70	\u0046	F	104	\u0068	h
71	\u0047	G	105	\u0069	i
72	\u0048	H	106	\u006a	j
73	\u0049	I	107	\u006b	k

Дес.	Unicode	Описание	Дес.	Unicode	Описание
108	\u006c	l	142	\u008e	управляющий символ
109	\u006d	m	143	\u008f	управляющий символ
110	\u006e	n	144	\u0090	управляющий символ
111	\u006f	o	145	\u0091	управляющий символ
112	\u0070	p	146	\u0092	управляющий символ
113	\u0071	q	147	\u0093	управляющий символ
114	\u0072	r	148	\u0094	управляющий символ
115	\u0073	s	149	\u0095	управляющий символ
116	\u0074	t	150	\u0096	управляющий символ
117	\u0075	u	151	\u0097	управляющий символ
118	\u0076	v	152	\u0098	управляющий символ
119	\u0077	w	153	\u0099	управляющий символ
120	\u0078	x	154	\u009a	управляющий символ
121	\u0079	y	155	\u009b	управляющий символ
122	\u007a	z	156	\u009c	управляющий символ
123	\u007b	{	157	\u009d	управляющий символ
124	\u007c	|	158	\u009e	управляющий символ
125	\u007d	}	159	\u009f	управляющий символ
126	\u007e	~	160	\u00a0	[неразрывный пробел]
127	\u007f	[удаление]	161	\u00a1	Ў
128	\u0080	управляющий символ	162	\u00a2	ў
129	\u0081	управляющий символ	163	\u00a3	J
130	\u0082	управляющий символ	164	\u00a4	¤
131	\u0083	управляющий символ	165	\u00a5	Ґ
132	\u0084	управляющий символ	166	\u00a6	¦
133	\u0085	управляющий символ	167	\u00a7	§
134	\u0086	управляющий символ	168	\u00a8	Ё
135	\u0087	управляющий символ	169	\u00a9	©
136	\u0088	управляющий символ	170	\u00aa	Є
137	\u0089	управляющий символ	171	\u00ab	«
138	\u008a	управляющий символ	172	\u00ac	¬
139	\u008b	управляющий символ	173	\u00ad	-
140	\u008c	управляющий символ	174	\u00ae	®
141	\u008d	управляющий символ	175	\u00af	Ї

Дес.	Unicode	Описание	Дес.	Unicode	Описание
176	\u00b0	°	210	\u00d2	Т
177	\u00b1	±	211	\u00d3	У
178	\u00b2	I	212	\u00d4	Ф
179	\u00b3	i	213	\u00d5	Х
180	\u00b4	г	214	\u00d6	Ц
181	\u00b5	µ	215	\u00d7	Ч
182	\u00b6	¶	216	\u00d8	Ш
183	\u00b7	·	217	\u00d9	Щ
184	\u00b8	ё	218	\u00da	Ъ
185	\u00b9	№	219	\u00db	Ы
186	\u00ba	є	220	\u00dc	Ь
187	\u00bb	»	221	\u00dd	Э
188	\u00bc	j	222	\u00de	Ю
189	\u00bd	S	223	\u00df	Я
190	\u00be	s	224	\u00e0	а
191	\u00bf	ї	225	\u00e1	б
192	\u00c0	А	226	\u00e2	в
193	\u00c1	Б	227	\u00e3	г
194	\u00c2	В	228	\u00e4	д
195	\u00c3	Г	229	\u00e5	е
196	\u00c4	Д	230	\u00e6	ж
197	\u00c5	Е	231	\u00e7	з
198	\u00c6	Ж	232	\u00e8	и
199	\u00c7	З	233	\u00e9	й
200	\u00c8	И	234	\u00ea	к
201	\u00c9	Й	235	\u00eb	л
202	\u00ca	К	236	\u00ec	м
203	\u00cb	Л	237	\u00ed	н
204	\u00cc	М	238	\u00ee	о
205	\u00cd	Н	239	\u00ef	п
206	\u00ce	О	240	\u00f0	р
207	\u00cf	П	241	\u00f1	с
208	\u00d0	Р	242	\u00f2	т
209	\u00d1	С	243	\u00f3	у

Дес.	Unicode	Описание	Дес.	Unicode	Описание
244	\u00f4	ф	250	\u00fa	ъ
245	\u00f5	х	251	\u00fb	ы
246	\u00f6	ц	252	\u00fc	ь
247	\u00f7	ч	253	\u00fd	э
248	\u00f8	ш	254	\u00fe	ю
249	\u00f9	щ	255	\u00ff	я

В табл. В.2 перечислены коды избранных специальных клавиш на клавиатуре. Эти коды клавиш используются только с объектом *Key*. См. алфавитный список его особых свойств в табл. С.8 (в статье «*Key*, объект» «Справочника по языку»).

Таблица В.2. Коды специальных клавиш объекта Key

Клавиша	Код клавиши	Свойство	Клавиша	Код клавиши
Backspace	8	Key.BACKSPACE	Num Pad 0	96
Tab	9	Key.TAB	Num Pad 1	97
Enter	13	Key.ENTER	Num Pad 2	98
Shift	16	Key.SHIFT	Num Pad 3	99
Control	17	Key.CONTROL	Num Pad 4	100
Alt	18 (не перехватывается)		Num Pad 5	101
Pause/ Break	19		Num Pad 6	102
Caps Lock	20	Key.CAPSLOCK	Num Pad 7	103
Escape	27	Key.ESCAPE	Num Pad 8	104
Space	32	Key.SPACE	Num Pad 9	105
Page Up	33	Key.PGUP	Num Pad *	106
Page Down	34	Key.PGDN	Num Pad +	107
End	35	Key.END	Num Pad Enter	13 (не перехватывается как 108)
Home	36	Key.HOME	Num Pad (	109
Left arrow	37	Key.LEFT	Num Pad .	110
Up Arrow	38	Key.UP	Num Pad /	111
Right Arrow	39	Key.RIGHT	F1	112
Down arrow	40	Key.DOWN	F2	113
Print Screen	44		F3	114
Insert	45	Key.INSERT	F4	115
Delete	46	Key. DELETEKEY	F5	116

Клавиша	Код клавиши	Клавиша	Код клавиши
F6	117	; :	186
F7	118	+ =	187
F8	119	– _	189
F9	120	/ ?	191
F10	121 (не перехватывается)	` ~	192
F11	122	[{	219
F12	123	\ |	220
Num Lock	144	] }	221
Scroll Lock	145	" '	222

В табл. В.3 перечислены коды буквенных и цифровых клавиш. Эти коды клавиш используются только с объектом *Key* (см. *Key.getCode( )* и *Key.isDown( )* в «Справочнике по языку»).

Таблица В.3. Коды клавиш букв и цифр объекта Key

Клавиша	Код клавиши	Клавиша	Код клавиши	Клавиша	Код клавиши
A	65	M	77	Y	89
B	66	N	78	Z	90
C	67	O	79	0	48
D	68	P	80	1	49
E	69	Q	81	2	50
F	70	R	82	3	51
G	71	S	83	4	52
H	72	T	84	5	53
I	73	U	85	6	54
J	74	V	86	7	55
K	75	W	87	8	56
L	76	X	88	9	57

C

Обратная совместимость и обновление версий проигрывателя

Flash позволяет публиковать файлы *.swf* в формате, совместимом с прежними версиями проигрывателя (с помощью File → Publish Settings → Flash → Version). Когда вы будете это читать, у большинства пользователей уже будет установлен Flash Player версии не ниже 6, но может потребоваться создать фильм для Flash Player 5 или 4, например, если он предназначен для портативных устройств, в которых еще нет поддержки Flash Player 6. Даже тем, кто создает документы исключительно для Flash 6, это приложение поможет обойтись без устаревших элементов ActionScript и начать осваивать более быстрые методы, предпочтительные для Flash 6. Кроме того, им будет проще понять или обновить старый код, который может им встретиться.

Об определении версии Flash Player на этапе исполнения см. `capabilities.version`, *getVersion( )* и `$version` в «Справочнике по языку». Сценарий для броузера, определяющий версию, можно скачать на странице *http://www.moock.org/webdesign/flash/ detection/moockfpi*.

При разработке для Flash 4 или 5 применяйте *вышедшие из употребления* (*deprecated*) средства (то есть те, которые поддерживаются только для обратной совместимости), перечисленные в табл. C.1 (из раздела «Flash 4 в сравнении с Flash 5 и последующими версиями» данного приложения). См. также «Using Flash MX ActionScript with older versions of Flash» («Использование Flash MX ActionScript с более старыми версиями Flash») в разделе «Understanding the ActionScript Language» (Сведения о языке ActionScript) файлов электронной подсказки, вызываемой через Help → Using Flash. Вышедшие из употребления функции ActionScript перечислены в окне Toolbox панели Actions в папке Deprecated.

Сведения о распространенности различных версий Flash Player можно найти по адресу:

http://www.macromedia.com/software/flash/survey/whitepaper

Загрузить архивные версии Flash Player можно со следующих страниц:

http://www.macromedia.com/support/flash/downloads.html
http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm

Информацию о разработке Flash-содержимого для устройств см. в описании объекта *Capabilities* в «Справочнике по языку».

Для того чтобы сценарии выполнялись во Flash Player 4, необходимо на вкладке Flash в пункте File → Publish Settings задать Flash 4 в качестве Version для файла *.swf*. Иначе при попытке выполнения во Flash Player 4 кода из файла *.swf*, созданного во Flash 5 или более позднем, весь код будет отключен. Для того чтобы сценарии выполнялись во Flash Player 5, необходимо на вкладке Flash в пункте File → Publish Settings задать Flash 5 в качестве Version для файла *.swf*. Иначе при попытке выполнения во Flash Player 5 кода из файла *.swf*, созданного во Flash 6 или более позднем, весь код будет отключен.

Лучшие приемы программирования во Flash 6 ActionScript

Лишь небольшая часть Flash 5 ActionScript была формально объявлена устаревшей во Flash 6. Во Flash 6 появились следующие формальные изменения языка:

- Функцию *eval( )* запрещено использовать в левой части операции присваивания.
- Область видимости функции определяется в момент ее объявления, а не вызова, после чего остается постоянной. См. раздел «Область видимости функций» главы 9.

В следующем списке перечислены новые функции, которые следует предпочесть альтернативам, предоставляемым Flash 5:

- Глобальные переменные должны определяться в `_global`, а не в `_root` или *Object*.
- Конструкторы надклассов должны выполняться с помощью оператора *super*, а не ручной ссылки на функцию конструктора надкласса.
- Весь текст должен импортироваться в кодировке Unicode, а не в Latin 1 или Shift-JIS.
- Обработчики событий клипов и кнопок должны назначаться как свойства объектов, а не с помощью синтаксиса *onClipEvent(*`event`*)* или *on(*`event`*)*. См. в «Справочнике по языку» исключения для отдельных событий.
- Доступ к содержимому текстового поля должен осуществляться через свойство *`TextField`*`.text`, а не имя переменной текстового поля.
- Файлы JPEG следует загружать непосредственно с помощью *loadMovie( )*, а не заключенными в файлы *.swf*.
- Часто циклы временной диаграммы и *onEnterFrame( )* можно заменить *setInterval( )*.

- Вместо cookie JavaScript следует использовать локальный *SharedObject*.
- `System.capabilities.version` предпочтительнее *getVersion( )* (хотя в фильмах для Flash Player 5 необходимо применять *getVersion( )*).
- События объектов *Key* и *Mouse* – *onKeyDown( )*, *onKeyUp( )*, *onMouseDown( )* и *onMouseUp( )* – следует применять вместо аналогичных событий *MovieClip* за исключением случаев, когда требуется *updateAfterEvent( )*.
- Класс *LoadVars* предпочтительнее функции *loadVariables( )*.
- Свойства `Stage.scaleMode` и `Stage.showMenu` заменили глобальными вызовами *fscommand("allowscale")* и *fscommand("showmenu")*.

О других предпочтительных способах программирования во Flash 6 ActionScript читайте в следующем документе:

http://www.macromedia.com/desdev/mx/flash/whitepapers/actionscript_standards.pdf

Как правило, новые объекты, классы, команды и операторы, введенные во Flash 6 (и перечисленные в предисловии), нельзя применять при экспорте в формат Flash 5. Три исключения составляют инструкция *switch* и операторы строгого равенства и неравенства (`===` и `!==`), которые можно применять при экспорте из Flash MX в формат Flash 5. Если для фильма задан формат экспорта версии 5, то в окно Flash MX Output выводятся сообщения об обнаруженных при экспорте несовместимостях.

Обновление версий проигрывателя

Далее описываются изменения и исправления, введенные в младших версиях Flash Player.

Обновления Flash Player 6, Build 61

Во время подготовки этой книги к печати (ноябрь 2002 г.) Macromedia выпустила Build 61 проигрывателя Flash Player 6 (Flash Player 6.0.61.0). Ниже перечислены основные особенности новой сборки:

Безоконный режим

Реализован для Netscape под Windows и Mac OS X в дополнение к Internet Explorer под Windows. См. атрибут `WMODE` в приложении Н.

Улучшенная производительность ActionScript

Для ускорения доступа к переменным теперь используются регистры, благодаря чему ActionScript в целом стал выполняться быстрее. Повышенная производительность особенно заметна в компонентах FUI (Flash User Interface).

Настройка заголовков HTTP

Заголовки HTTP для операций POST можно настраивать с помощью новых методов *XML.addRequestHeader( )* и *LoadVars.addRequestHeader( )*.

Зависимые библиотеки совместного доступа этапа исполнения

Библиотекам совместного доступа этапа исполнения разрешено работать с объектами из других библиотек совместного доступа этапа исполнения.

Динамические свойства Accessibility (возможностей доступа) (только Windows, ActiveX)

Разрешается задавать свойства объектов доступа (доступные раньше только на этапе разработки в окне Flash MX Accessibility) на этапе исполнения с помощью нового метода *Accessibility.updateProperties( )* и объекта `Accessibility._accProps` (со свойствами `silent`, `forceSimple`, `name`, `description` и `shortcut`).

Поддержка Linux

Поддержка Mozilla 1.1 и Standalone Player под RedHat Linux 7.3.

Разные исправления

Исправлен ряд ошибок, особенно во Flash Player 6 для Internet Explorer под Windows.

Значительно подробнее об изменениях сказано в описании релиза по адресу:

http://www.macromedia.com/support/flash/releasenotes/player/rn_6.html

Обновления Flash Player 6, Build 47

В следующем списке перечислены основные изменения, внесенные в проигрыватель Flash Player 6 между сборками Build 40 и Build 47 (Flash Player 6.0.47.0), выпущенный в августе 2002:

- Минимизированный или скрытый Netscape под Windows больше не расходует процессорное время.
- В текстовых полях больше не появляется несколько курсоров.
- Закрыта уязвимость, найденная во Flash Player 6 – «Macromedia Flash URL Modification Issue». См.:

  http://www.macromedia.com/v1/handlers/index.cfm?ID=23294

Обновления Flash Player 6, Build 40

Ниже перечислены основные изменения, внесенные в проигрыватель Flash Player 6 между сборками Build 29 и Build 40 (Flash Player 6.0.40.0), выпущенный в июле 2002. Мелкие улучшения и исправления ошибок описаны в *http://www.macromedia.com/support/flash/ts/documents/flashplayer_r40.htm*.

- Исправлен метод POST в Netscape (все платформы) и Internet Explorer для Макинтошей.
- Подключаемый модуль для Netscape 6.2 и выше стал поддерживать связь JavaScript и *fscommand( )* через XPConnect (которым Netscape заменила свою технологию LiveConnect).
- Исправлена ошибка переноса на следующую строку, вызывавшая дрожание текста.

- Метод *Sound.start( )* стал поддерживать потоковый звук. Подробности см. в *Sound.start( )*.
- Добавлена возможность маскирования шрифтов устройств. Подробности см. в описании метода *MovieClip.setMask( )*.
- В объекте *Sound* стали доступны свойства MP3 ID3, описываемые в `Sound.id3` в «Справочнике по языку».
- Исправлен ввод с немецких клавиатур под Windows (теперь поддерживается ввод @ и других специальных символов).
- Исправлено отображение верхней части ASCII в Unicode на Макинтошах. Верхняя часть ASCII теперь должна правильно отображаться в версиях Flash Player для Макинтошей.
- В тег `OBJECT/EMBED` добавлен атрибут `allowScriptAccess`, который может запрещать выполнение сценариев других доменов со страницы HTML. См. *http://www.macromedia.com/support/flash/ts/documents/allow_script_access.htm*.

Обновления Flash Player 6, Build 29

В сборке Build 29 проигрывателя Flash Player 6 (Flash Player 6.0.29.0), выпущенной в мае 2002, исправлены следующие ошибки исходной общедоступной версии Flash Player 6 (Flash Player 6.0.25.0, выпущена в марте 2002):

- Прекращение загрузки файлов (*.swf*, *.jpg* или *.mp3*) стало проходить корректно.
- Прекращается воспроизведение звука события, прикрепленного к объекту *Sound*, когда не остается ссылок на объект *Sound*.
- Большие файлы (видео или MP3) перестали блокировать проигрыватель.

Обновления Flash Player 5, Build 41 и 42

В следующем списке собраны основные изменения в Build 41 (Netscape) и 42 (Internet Explorer) Flash Player 5 (предыдущей общедоступной сборкой была 30, первоначально выпущенная со средой разработки Flash 5):

- Курсор в текстовом поле принимает цвет текста.
- Фильмы в ячейке таблицы не приводят к аварийному завершению Internet Explorer 5.5.
- Положение прокрутки текстового поля не сбрасывается при модификации содержимого поля.
- Текстовые поля со встроенными шрифтами не выходят за границы области видимости при прокрутке.
- Добавлено свойство `XML.contentType`.
- Добавлено свойство `XML.ignoreWhite`.
- При синтаксическом анализе исходного кода XML и появлении в текстовом узле символов &, ', ", < и > последние преобразуются в сущности `&`, `'`, `"`, `<` и `>`. Это преобразование прозрачно во Flash, поскольку сущности преобразуются обратно в символы при преобразовании

объекта XML в строку; однако сущности *будут* присутствовать в исходном коде XML, посылаемом серверу.

- *Math.random( )* больше не возвращает значение 1. Максимальным возвращаемым значением является 0,999...
- Общая производительность улучшена, особенно в Windows 98.

Flash 4 в сравнении с Flash 5 (и последующими)

Основные вопросы обратной совместимости и различий между ActionScript во Flash 4, Flash 5 и Flash 6 приведены в табл. C.1.

Таблица C.1. Вопросы обратной совместимости с Flash 4

Вопрос	Описание
Создание переменных	Функция Flash 4 *set* заменена инструкцией *var*
Ссылки на переменные и временные диаграммы	Конструкции в стиле Flash 4 из косой черты и двоеточия (`/square:area`) заменены точечной нотацией (`square.area`)
Операторы сравнения строк	Операторы сравнения строк Flash 4 `eq`, `ne`, `ge`, `gt`, `le`, `lt` заменены во Flash версии 5 и более поздних следующими операторами: `==`, `!=`, `>=`, `>`, `<=`, `<`
Оператор конкатенации строк	При создании содержимого Flash 4 во Flash версии 5 и более поздних следует вместо оператора Flash 4 `&` применять оператор *add*. При создании документов для Flash версии 5 и выше используйте для конкатенации строк оператор `+`
Длина строки	Функция Flash 4 *length( )* (например, *length(`myString`)*) заменена свойством *`String`*`.length` (например, *`myString`*`.length`)
Извлечение подстроки	Функция Flash 4 *substring( )* (например, *substring(`myString,1,3`)*) заменена методами *substring( )*, *substr( )* и *slice( )*. Обратите внимание, что *substring( )*во Flash 4 и Flash 5 имеет различия
Функции кодовой позиции символа	Функции Flash 4 *chr( )* и *mbchr( )* (применяемые для создания символа по кодовой позиции) заменены *String.fromCharCode( )*. Функции Flash 4 *ord( )* и *mbord( )* (позволяющие определить кодовую позицию символа) заменены методом *String.charCodeAt( )*
Преобразование типа данных	При импортировании файлов Flash 4, Flash 5 и Flash MX автоматически вызывают функцию *Number( )* для любых числовых данных, выступающих в качестве операндов следующих потенциально неоднозначных операторов: `+`, `==`, `!=`, `< >`, `<`, `>`, `> =`, `<=`
Инструкция *ifFrameLoaded*	Инструкция Flash 3 *ifFrameLoaded* вышла из употребления. Для создания кода предварительной загрузки следует применять свойства `_totalframes` и `_framesloaded` *MovieClip*
Бесконечные циклы	Flash 4 допускал максимум 200 000 итераций цикла. Flash 5 и Flash 6 разрешают 15 секунд на цикл, после чего извещают пользователя об отсутствии отклика от фильма. См. раздел «Максимальное количество итераций» главы 8

Таблица С.1 (продолжение)

Вопрос	Описание
Сравнение подпрограмм и функций	Во Flash 4 можно было создать подпрограмму, прикрепив блок кода к кадру с меткой и выполняя его с помощью утверждения *call( )*. Во Flash 5 и выше подпрограммы Flash 4 заменяются функциями
События клипов	Flash 4 поддерживал только события кнопок (то есть функции, начинающиеся с *on( )*). О современной технологии обработки событий рассказано в главе 10
Перехват нажатий клавиш	Во Flash 4 единственным средством перехвата нажатий клавиш было событие кнопки *keyPress*. Во Flash 5 и выше объект *Key* в сочетании с событиями клипов *onClipEvent(keyDown)* и *onClipEvent(keyUp) keyDown* предоставляет значительно больший контроль над взаимодействием с клавиатурой
Tell Target (устарела)	*Tell Target* из Flash 4 (обеспечивающая управление удаленными клипами фильмов) заменена свойствами и методами, доступ к которым осуществляется с помощью точечной нотации и инструкции *with*. См. «Использовать ли Tell Target?» далее в этом приложении
Get Property (устарела)	Команда Flash 4 *Get Property* более не требуется для доступа к свойствам клипа фильма. Ее место занял оператор «точка». См. главу 13
int (устарела)	Функция Flash 4 *int( )* (применялась для усечения чисел с плавающей запятой до целых) заменена методами *Math.floor( )*, *Math.ceil( )* и *Math.round( )*
Генерирование случайных чисел	Функция Flash 4 *random( )* (применявшаяся для генерирования псевдослучайного числа) заменена методом *Math.random( )*
toggleHighQuality (устарела)	Функция Flash 4 *toggleHighQuality* (позволявшая установить качество вывода проигрывателя) заменена глобальным свойством `_quality`
`_highquality` (устарело)	Свойство Flash 4 `_highquality` заменено глобальным свойством `_quality`
Поддержка объекта *Math* во Flash 4	Функции и свойства объекта *Math* (например, *Math.cos( )*, `Math.PI`) не имеют встроенной поддержки во Flash Player 4. Однако значения аппроксимируются при экспорте в формате Flash 4
loadMovie и *loadMovieNum*	Функция Flash 3 *loadMovie( )*, загружавшая фильм на уровень с заданным номером, во Flash 5 заменена функцией *loadMovieNum( )* (принимающей целочисленный аргумент уровня). Однако функция *loadMovie( )* Flash 4, выполняющая загрузку в целевой клип, по-прежнему доступна во Flash 5 и выше как *loadMovie( )*
Печать	Flash 5 и выше поддерживает встроенные функции печати, которые во Flash 4.0.20.0 и выше были доступны только как модифицированное действие *Get URL*
Не поддерживаемые объекты и классы	Flash 4 не поддерживает объекты и классы, встроенные во Flash 5 и Flash 6 (исключая *Math*, о котором говорилось выше)

Управление клипами

В главе 13 было рассказано об управлении клипами с помощью технологий Flash 5 (и более поздних). Сейчас будет рассмотрена эквивалентная технология Flash 4.

До появления Flash 5 для управления клипом требовалось исполнить специальные *Movie Clip Actions*. Надо было сказать: «Пусть проигрывается клип с именем eyes», для чего выполнялся следующий код:

```
Begin Tell Target ("eyes")
 Play
End Tell Target
```

Однако во Flash 5 клипами фильмов можно управлять более непосредственным способом – с помощью встроенных методов, например:

```
eyes.play();
```

Аналогично для доступа к встроенным свойствам фильма до Flash 5 приходилось прибегать к командам непосредственного чтения и записи свойств, например:

```
GetProperty ("ball", _width)
Set Property ("ball", X Scale) = 90
```

Во Flash 5 стало возможным извлекать и задавать свойства клипа с помощью оператора точки, так же, как происходит обращение к свойствам любого объекта:

```
ball._width;
ball._xscale = 90;
```

До Flash 5 обращение к переменным внутри клипа осуществлялось с помощью двоеточия, отделяющего имя клипа от имени переменной:

```
Set Variable: "x" = myClip:myVariable
```

Во Flash 5 переменная клипа является просто свойством объекта этого клипа, поэтому теперь для установки и получения значений переменных применяется оператор «точка»:

```
x = myClip.myVariable;
```

Наконец, до Flash 5 обращение к вложенным уровням клипов напоминало обращение к элементам дерева каталогов с применением символов косой черты и точек:

```
clipA/clipB/clipC
../../../clipC
```

Поскольку во Flash 5 клипы являются данными сходного с объектами вида, можно обращаться к одному клипу как к свойству другого. Поэтому для доступа к так называемым *вложенным* (*nested*) клипам применяется оператор «точка», а зарезервированное свойство клипа _parent позволяет ссылаться на содержащий его клип:

```
clipA.clipB.clipC;
_parent._parent._parent.clipC;
```

Сравнение синтаксиса обращения к переменным во Flash 4 и Flash 5

Конструкции в стиле Flash 4, содержащие косую черту и двоеточие, например `/square:area`, заменены во Flash 5 (и более поздних) синтаксисом с точкой – значительно более удобным способом ссылки на переменные и временные диаграммы. Прежний синтаксис устарел и не рекомендуется больше к применению. В табл. С.2 показаны эквивалентные формы обращения к переменным во Flash версий 4 и 5 (и более поздних). Другие синтаксические различия приведены в табл. С.1.

Таблица С.2. Синтаксис обращения к переменным во Flash 4 и 5

Синтаксис Flash 4	Синтаксис Flash 5	Ссылается на...
`/`	`_root`	Главную временную диаграмму фильма
`/:x`	`_root.x`	Переменную `x` на главной временной диаграмме фильма
`/clip1:x`	`_root.clip1.x`	Переменную `x` в экземпляре `clip1` на главной временной диаграмме фильма
`/clip1/clip2:x`	`_root.clip1.clip2.x`	Переменную `x` в экземпляре `clip2` в экземпляре `clip1` на главной временной диаграмме фильма
`../`	`_parent`	Временную диаграмму, на которой расположен текущий клип (на один уровень выше временной диаграммы текущего клипа*)
`../:x`	`_parent.x`	Переменную `x` временной диаграммы, на которой расположен текущий клип (на один уровень выше временной диаграммы текущего клипа)
`../../:x`	`_parent._parent.x`	Переменную `x` временной диаграммы, содержащей клип, содержащий текущий (на два уровня выше временной диаграммы текущего клипа)
`clip1:x`	`clip1.x`	Переменную `x` в экземпляре `clip1`, причем `clip1` расположен в текущей временной диаграмме
`clip1/clip2:x`	`clip1.clip2.x`	Переменную `x` в экземпляре `clip2`, причем `clip2` расположен в `clip1`, который находится в текущей временной диаграмме
`_level1:x`	`_level1.x`	Переменную `x` главной временной диаграммы фильма, загруженного в уровень 1
`_level2:x`	`_level2.x`	Переменную `x` главной временной диаграммы, загруженной в уровень 2

* «Временная диаграмма текущего клипа» – это та временная диаграмма, которая содержит ссылку на переменную.

Локальные переменные в подпрограммах, написанных в стиле Flash 4

Функции представляют собой предпочтительный механизм создания переносимых модулей кода, но во Flash версий 5 и 6 по-прежнему поддерживаются *подпрограммы* в стиле Flash 4. Во Flash 4 подпрограмму можно было создать, прикрепив блок кода к кадру с меткой. После этого подпрограмму можно было вызывать удаленно с помощью действия *Call*. Но во Flash 4 переменные, объявленные в подпрограмме, не были локальными и сохранялись, пока существовала временная диаграмма, в которой она была определена. Во Flash версий 5 и выше можно создавать локальные переменные в подпрограммах тем же способом, каким мы создавали их в функциях, – с помощью инструкции *var*. Однако переменные, определенные в подпрограмме через *var*, создаются как локальные переменные только тогда, когда подпрограмма выполняется через функцию *Call*. Если сценарий в кадре подпрограммы выполняется в результате перемещения в кадр головки воспроизведения, то инструкция *var* объявляет обычную нелокальную переменную временной диаграммы. Тем не менее вместо подпрограмм следует использовать более современные функции и локальные переменные функций.

Преобразование типов данных Flash 4 во Flash 5

Во Flash 4 строковые и числовые операторы были совершенно различными: одна группа операторов работала только с числами, а другая – только со строками. Например, оператором конкатенации строк во Flash 4 был `&`, тогда как сложение чисел выполнял оператор `+`. Аналогично сравнение строк осуществлялось посредством операторов `eq` и `ne`, а сравнение чисел – операторов `=` и `<>`. В табл. C.3 представлен синтаксис Flash 5 и более поздних для аналогичных операторов Flash 4.

Таблица C.3. Операторы Flash 4 в сравнении с операторами Flash 5

Операция	Синтаксис Flash 4	Синтаксис Flash 5
Конкатенация строк	`&`	`+` или *add*
Равенство строк	`eq`	`==`
Неравенство строк	`ne`	`!=`
Сравнение строк	`ge, gt, le, lt`	`>=, >, <=, <`
Сложение чисел	`+`	`+`
Равенство чисел	`=`	`==`
Неравенство чисел	`<>`	`!=`
Сравнение чисел	`>=, >, <=, <`	`>=, >, <=, <`

Операндами некоторых операторов Flash 5 (и более поздних версий) могут быть как строки, так и числа. Например, если это строки, то оператор `+` конкатенирует операнды, образуя новую строку. А если числа, то оператор `+`

складывает два операнда математически. Аналогично оператор равенства (==) и оператор неравенства (!=) применяются во Flash 5 для сравнения строк, чисел и других типов данных.

Поскольку во Flash 5 многие операторы работают с несколькими типами данных, а во Flash 4 – нет, то при импорте файлов Flash 4 во Flash 5 возникает неоднозначность. Поэтому при импорте файлов Flash 4 во Flash 5 и Flash MX автоматически вставляется функция *Number( )*, аргументами которой являются любые числовые данные, выступающие в качестве операндов следующих потенциально неоднозначных операторов (если только операнд не является числовым операндом): +, ==, !=, <>, <, >, >=, <=.

В файлах Flash 4, конвертированных в формат Flash 5 или Flash MX, будет также заменен оператор конкатенации строк (&) на новый оператор *add*. В табл. С.4 представлены примеры перевода операторов Flash 4 в операторы Flash 5.

Таблица С.4. Примеры трансляции операторов Flash 4 в операторы Flash 5

Синтаксис Flash 4	Синтаксис Flash 5 (и последующих)
`Loop While (count <= numRecords)`	`while (Number(count)<=Number(numRecords))`
`If (x = 15)`	`if(Number(x) == 15)`
`If (y <> 20)`	`if(Number(y) != 20)`
`Set Variable: "lastName" = "kavanagh"`	`lastName = "kavanagh"`
`Set Variable: "name" = "molly" & lastName`	`name = "molly" add lastName`

События Flash 4

Большинство событий ActionScript впервые появилось во Flash 5. Экспортируя фильм в формат Flash 4, используйте только обработчики событий кнопок (во Flash 4 поддерживались только события кнопок) и тщательно протестируйте результат во Flash Player 4. Чтобы узнать, поддерживается ли событие конкретной версией Flash Player, обращайтесь в статье, посвященной этому событию в «Справочнике по языку».

Строковые операторы Flash 4 и Flash 5

Оператор конкатенации строк Flash 4 (&) выполняет во Flash 5 (и последующих версиях) другую операцию, а именно «поразрядное И». Если вы осуществляете экспорт файла *.swf* из более новой версии Flash в формат Flash 4, то должны конкатенировать строки оператором *add*. Обратите внимание, что *add* поддерживается только для обратной совместимости: во Flash версии 5 и выше предпочтительнее оператор +.

Экспортируя файл *.swf* в формат Flash 4, следует пользоваться более старыми операторами `eq` и `ne` для сравнения строк на равенство (==) и неравенство (!=). Во Flash 5 и выше следует применять операторы == и !=.

В файлах *.swf*, экспортируемых в формат Flash 4, сравнение строк должно выполняться более старыми операторами `gt`, `ge`, `lt` и `le`. Во Flash 5 и выше следует применять эквивалентные операторы `>`, `>=`, `<` и `<=`.

В файлах *.swf*, экспортируемых в формат Flash 4, длину строки следует определять более старой функцией *length( )*, поддерживаемой для обратной совместимости. А во Flash 5 и выше лучше предпочесть свойство *String*.`length`. Вот как функция Flash 4 *length( )* применяется для вывода количества символов в слове «obsolete»:

```
trace (length("obsolete"));  // Выводит: 8
```

Во Flash 5 метод *String.substr( )* предназначен для извлечения подстрок и ближе всего походит на функцию Flash 4 *substring( )*, которой для извлечения подстроки также передаются начальный индекс и длина. Функция Flash 5 *String.substring( )* принимает начальный и конечный индексы.

Экспортируя файл *.swf* в формат Flash 4, для создания символов по их числовому коду необходимо применять более старые функции Flash 4 *chr( )* и *mbchr( )*. Во Flash 5 и выше следует предпочесть метод *String.fromCharCode( )*.

В файле *.swf*, экспортируемом в формат Flash 4, кодовую позицию символа надо определять посредством более старых функций Flash 4 *ord( )* и *mbord( )*. Во Flash 5 и выше следует предпочесть метод *String.charCodeAt( )*.

Во Flash 5 все строковые операции и функции работают с многобайтовыми символами, поэтому во Flash 5 нельзя принудительно выполнять однобайтовые операции, как во Flash 4. Например, функция Flash 5 *fromCharCode( )* ближе всего к *chr( )*. То же относится к *mblength( )* и `length`, *mbsubstring( )* и *substr( )*, *mbord( )* и *charCodeAt( )*.

Использовать ли Tell Target?

Во Flash 4 основным инструментом для ссылки на клипы был *Tell Target*, который, будь он благословен, был довольно громоздким и вышел из употребления в связи с появлением во Flash 5 значительно более элегантной объектной модели. Функция *Tell Target* устарела (то есть применять ее больше не рекомендуется). И хотя ее по-прежнему можно использовать для программирования в стиле Flash 4, в будущем она, вероятно, исчезнет.

Рассмотрим такой код, в котором *Tell Target* применяется для воспроизведения экземпляра с именем `closingSequence`:

```
Begin Tell Target ("closingSequence")
  Play
End Tell Target
```

Во Flash 5 мы вызываем гораздо более удобный и легче читаемый метод *play( )* с экземпляром `closingSequence`:

```
closingSequence.play();
```

Tell Target может также выполнять несколько операций над экземпляром внутри блока кода, например:

```
Begin Tell Target ("ball")
  (Set Property: ("ball", x Scale) = "5")
  Play
End Tell Target
```

Во Flash 5 для получения аналогичных результатов следует предпочесть инструкцию *with( )*, описанную в главе 6:

```
with (ball) {
  _xscale = 5;
  play();
}
```

D

Отличия от ECMA–262 и JavaScript

Естественно, что эта книга посвящена сложностям ActionScript, но те, у кого есть отличная книга издательства O'Reilly «JavaScript: The Definitive Guide»,[1] обнаружат большое сходство между справочными разделами обеих книг.

Хотя ActionScript, как и JavaScript, основан на стандарте ECMA-262, его разработчики были вынуждены пойти на некоторые отклонения из-за ограничений на размер проигрывателя и для обеспечения обратной совместимости. Тот, кому потребуется перенести код из JavaScript, Jscript или другого основанного на ECMA-262 языка, сможет извлечь большую пользу из табл. D.1. В ней собраны запроектированные отличия ActionScript (для Flash 5 и выше) от ECMA-262 и JavaScript. Аналогично, тому, кто решит перенести приложение ActionScript на другой язык, будет легче избежать ловушек, обусловленных отклонениями ActionScript от стандарта ECMA-262.

В табл. D.1 собраны отличия ActionScript для Flash 5 от стандарта ECMA-262, представляющие собой результат *умысла*. В ней не приведены ошибки, которые могли быть сделаны при попытке реализации стандарта. Кроме того, в табл. D.1 отражены особенности ActionScript, проявляющиеся в отсутствие прагмы `#strict`. (Недокументированная и не поддерживаемая прагма `#strict` имела целью повысить совместимость ActionScript со стандартом ECMA-262. Однако применять ее *не следует*, потому что она приводит к аварийному завершению Internet Explorer для Windows. См. подробности в статье, посвященной прагме `#strict` в «Справочнике по языку».)

Таблица D.1. Отличия ActionScript от ECMA-262 и JavaScript

Вопрос	Описание
Обработка исключительных ситуаций	ActionScript не поддерживает обработку исключительных ситуаций *try/catch/finally* или *throw*.

[1] Дэвид Флэнаган «JavaScript. Подробное руководство», 4-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2004 г.

Вопрос	Описание
escape( ) кодировка (только Flash 6)	Во Flash 6 функция *escape( )* ActionScript кодирует символы между кодовыми позициями 127 и 255 (верхняя часть таблицы ASCII) в последовательности UTF-8 с использованием `%xx`, а не метода, предлагаемого спецификацией ECMA-262. См. статью, посвященную функции *escape( )* в «Справочнике по языку».
Метки инструкций	ActionScript не поддерживает инструкции с метками, применяемые в командах *break* и *continue* для управления выполнением программы.
Преобразование строк в булевы значения	В ECMA-262 любая непустая строка преобразуется в `true`. В ActionScript в `true` преобразуются только строки, которые могут быть преобразованы в допустимые числа, отличные от нуля.
Чувствительность к регистру	Спецификация ECMA-262 требует полной чувствительности к регистру, а в ActionScript к регистру чувствительны ключевые слова, но не идентификаторы. См. главу 15, особенно табл. 15.1.
Область видимости функций	Во Flash 5 (но не во Flash 6), когда функция из одной временной диаграммы присваивается переменной во временной диаграмме другого клипа, цепочкой областей видимости присвоенной функции становится временная диаграмма этой переменной. В ECMA-262 и ActionScript для Flash 6 нельзя изменить область видимости функции путем присваивания: она устанавливается постоянной и зависит от местонахождения объявления функции.
Регулярные выражения	ActionScript не поддерживает регулярные выражения. Однако разработчик Павилс Юрьянс (Pavils Jurjans) создал свой объект *RegExp*, который можно загрузить с *http://www. jurjans.lv/flash/RegExp.html*.
Свойства обработчиков событий (только Flash 5)	Во Flash 5 (но не Flash 6) обработчики событий клипов должны определяться с помощью *onClipEvent (`eventName`)*, а обработчики событий кнопок определяются с помощью *on (`eventName`)*. Во Flash 6 обработчики событий клипов и кнопок должны присваиваться как свойства объектов, как в JavaScript. См. «Синтаксис обработчиков событий» в главе 10.
Глобальные переменные	ActionScript Flash 5 не поддерживает переменные, глобальные для всего документа. Глобальные переменные могут моделироваться путем прикрепления свойств к `Object.prototype`, как описано в разделе «Конец цепочки наследования» главы 12. Во Flash 6 глобальные переменные создаются в объекте `_global` (отсутствующем в спецификации ECMA-262).
Функция *eval( )*	В ActionScript функция *eval( )* поддерживает лишь небольшую часть возможностей, предусматриваемых ECMA-262: она работает, только когда ей в качестве аргумента передается идентификатор, и применяется только для того, чтобы динамически генерировать ссылки на идентификаторы. Во Flash 6 она не может находиться в левой части инструкции присваивания.

Вопрос	Описание
Преобразование данных типа `defined`	В ActionScript особое значение `undefined` в строковом контексте преобразуется в пустую строку (`""`) и в число 0 – в числовом контексте. В ECMA-262 `undefined` преобразуется в строку «undefined» в строковом контексте и в числовое значение `NaN` в числовом контексте.
Конструктор *Function*	ActionScript не поддерживает конструктор *Function*, применяемый в JavaScript для создания функций с помощью синтаксиса *new Function ( );*.
Создание объекта *Date*	При создании нового объекта *Date* ActionScript не примет (то есть не сможет разобрать) строку даты, читаемую человеком, например такую, как «January 9, 2001».
Инструкция *switch* (только Flash 5)	ActionScript для Flash 5 не поддерживает инструкции *switch/case/default* (применяемые для построения сложных условных операторов). Инструкция *switch* поддерживается во Flash 6 и может использоваться при экспорте из среды разработки Flash MX в файл *.swf* формата Flash 5 или Flash 4.
Поддержка языков (только Flash 5)	ECMA-262 требует поддержки стандарта кодировки символов Unicode, а ActionScript для Flash 5 этого требования не выполняет. Flash 5 поддерживает наборы символов Latin 1 и Shift-JIS и реализует некоторое подмножество функций и соглашений в стиле Unicode (например, escape-последовательности `\u`). Flash 6 полностью поддерживает Unicode.
Объектная модель	Естественно, что JavaScript содержит встроенные классы и объекты, относящиеся к веб-броузерам, а Flash содержит те, которые относятся к фильмам Flash. Для программистов на JavaScript, привыкших к работе с DHTML, может оказаться полезным представлять себе главный фильм документа Flash как аналог объекта документа HTML, а клипы – как аналоги объектов слоев.
Выполнение кода по времени	Методы *setTimeout( )* и *setInterval( )* объекта окна в JavaScript отсутствуют в ActionScript для Flash 5, но могут быть смоделированы с помощью циклов событий временных диаграмм и клипов, как показано в главе 8. Flash 6 поддерживает *setInterval( )*, но не *setTimeout( )*.
Конструктор *Object*	Во Flash 5 конструктор ActionScript для класса *Object* не принимает параметров. В ECMA-262 *Object* принимает параметр `value`, который может быть булевым значением, строкой или элементарным числовым типом.
Слэш-нотация и Tell Target	ActionScript поддерживает старую слэш-нотацию для ссылок на переменные и управления фильмами через *Tell Target* (например, «/ball:speed»). Ни *Tell Target*, ни слэш-нотация не определены в ECMA-262.

Вопрос	Описание
Нестандартные обработчики событий *on( )* и *onClipEvent( )*	ActionScript поддерживает отличный от EMCA-262 формат обработчиков событий для кнопок (*on( )*) и клипов (*onClipEvent( )*), применяемый для непосредственного прикрепления обработчиков событий к объектам на этапе разработки. См. главу 10.
`arguments.caller`	Свойство `arguments.caller` поддерживается ActionScript, но вышло из употребления в Java Script версии 1.3. В Netscape 4 свойство JavaScript `arguments.caller` относится к объекту *Arguments* вызывающей функции. В ActionScript свойство `arguments.caller` служит ссылкой на саму вызывающую функцию, а не объект *Arguments*. Подробности см. в «Справочнике по языку» в статье, посвященной `arguments.caller`.
Необъявленные переменные	В JavaScript ссылка на необъявленную переменную вызывает ошибку времени выполнения. ActionScript терпимо относится к ссылкам на необъявленные переменные.
Свойство `__proto__`	Свойство *`Object`*`.__proto__` формально документировано в ActionScript, чего не делают ECMA или JavaScript. См. *`Object`*`.__proto__` в «Справочнике по языку».
Вызов необъявленных функций	В JavaScript вызов необъявленной функции вызывает ошибку времени выполнения. В ActionScript ошибка не генерируется.
constructor, свойство	В JavaScript (в соответствии с ECMA-262) свойство `constructor` всегда читается из свойства `prototype` конструктора объекта (*`someObject`*`.__proto__.constructor`). Во Flash Player 6 и 5 свойство `constructor` записывается непосредственно в *`someObject`*. Это тонкое различие подробно рассмотрено в разделе «Более сложные проблемы стандартного назначения надкласса» главы 12.
Object.*toString( )*	ECMA-262 постулирует, что для встроенных классов *toString( )* длжен возвращать: [object *`class`*], где *`class`* – имя класса объекта. В ActionScript *toString( )* возвращает [object Object], если класс не реализует специальный *toString( )* (например, объект *Date* возвращает дату в читаемом формате).
super «оператор»	«Оператор» ActionScript *super* не является реализацией оператора ECMAScript v4 *super*. См. раздел «'Оператор' super» главы 5.

E

Поддержка HTML в текстовых полях

В данном приложении излагаются специальные замечания по использованию текстовых полей с поддержкой HTML и перечисляются теги HTML, которые в них действуют.

Поддержка HTML в текстовых полях

В число тегов HTML, поддерживаемых в текстовых полях Flash 5, входят `<B>`, `<I>`, `<U>`, `<FONT>`, `<P>`, `<BR>` и `<A>`. Flash 6 добавляет поддержку `<LI>` и `<TEXTFORMAT>`. Хотя тег `<TABLE>` не поддерживается, таблицы HTML можно моделировать с помощью позиций табуляции в текстовых полях (см. описание *TextFormat*.tabStops в «Справочнике по языку») и компонента Macromedia FDataGrid, который можно получить на Flash Exchange (*http://www.macromedia.com/exchange/flash/*).

Чтобы включить поддержку текста в формате HTML для объекта *TextField*, присвойте его свойству *TextField*.html значение true, а затем задайте в качестве значения свойства *TextField*.htmlText нужный текст в разметке HTML. Теги `<HTML>` и `<BODY>` не поддерживаются и не требуются. Неподдерживаемые теги игнорируются, хотя их текстовое содержание сохраняется. Flash требует заключать в кавычки значения, присваиваемые атрибутам тегов. Подробнее об этом и других близких темах см. в разделах «Использование HTML для ввода и вывода», «Нераспознаваемые теги и атрибуты» и «Заключение значений атрибутов в кавычки» далее в этом приложении.

Подробнее о применении HTML в текстовых полях говорится в статье «Справочника по языку», посвященной *TextField*.htmlText.

<B> (Bold)

Тег `<B>` показывает текст полужирным шрифтом, при условии существования полужирного начертания для выбранного шрифта. Например,

```
<B>This is bold text.</B>
```

будет показано как **This is bold text.**

См. обсуждение встраивания полужирного шрифта в статьях «Справочника по языку», посвященных *TextField*.embedFonts и *TextField*.htmlText.

<I> (Italics)

Тег <I> выводит текст курсивом, при условии существования курсивного начертания для выбранного шрифта. Например,

```
<I>This is italic text.</I>
```

будет показано как *This is italic text*.

См. обсуждение встраивания курсивного шрифта в статьях *TextField*.embedFonts и *TextField*.htmlText «Справочника по языку».

<U> (Underline)

Тег <U> выводит текст с подчеркиванием. Например,

```
<U>This is underlined text.</U>
```

будет показано как This is underlined text.

Поскольку Flash не подчеркивает гиперссылки автоматически, для их выделения следует использовать тег <U>. Например,

```
<A HREF="http://www.flight404.com"><U>Click here</U>
</A> to visit a neat site.
```

будет выведено как Click here to visit a neat site.

<LI> (List Item)

Тег <LI> создает маркер перед текстом, который в нем заключен. В отличие от HTML, <LI> не требует тега <UL>. Например,

```
While you work: <LI>Stretch</LI><LI>Drink water</LI><LI>Rest your eyes</LI>
```

будет выведен так:

While you work:

- Stretch
- Drink water
- Rest your eyes

Тег <LI> не поддерживает многоуровневые маркеры или тег <OL> (нумерованный список). Строки в формате *TextFormat*.bullet представляются в *TextField*.htmlText тегами <LI>.

<FONT> (Font Control)

Тег <FONT> поддерживает следующие три атрибута:

FACE

Атрибут FACE задает название шрифта. Во Flash 6, как и в HTML, можно задать список из нескольких названий шрифтов, хотя эта возможность не

документирована. Flash пытается выводить текст первым найденным шрифтом из перечисленных в атрибуте FACE. Например, в коде <FONT FACE="Arial,Helvetica">my text</FONT> Flash выведет «my text» шрифтом Helvetica, если отсутствуетArial, но есть Helvetica. Подробности см. в описании *TextFormat*.font. Функция, выбирающая подходящий шрифт на машине пользователя, описана в *TextField.getFontList( )*.

SIZE

Атрибут SIZE задает размер содержащегося в теге текста в виде фиксированного количества пунктов (например, <FONT SIZE="18">) или как относительный размер. Относительному размеру предшествует знак + или –, и он указывается относительно размера текста, заданного для текстового поля в инспекторе свойств. Например, если в инспекторе свойств задан размер в 14 пунктов, то <FONT SIZE="- 2"> выводит в теге текст высотой в 12 пунктов. См. также *TextFormat*.size.

COLOR

Атрибут COLOR задает цвет текста, помещенного в тег, в виде шестнадцатеричного числа, которому предшествует знак (#) – например, <FONT COLOR="#FF0000">this is red text</FONT>. Шестнадцатеричное число представляет собой RGB-значение цвета как три двузначных числа от 00 до FF. Атрибут COLOR реализован во Flash более строго, чем позволяет HTML, – необходим знак (#) и в качестве значений не могут выступать названия цветов, например "green" или "blue". См. также класс *Color* и *TextFormat*.color.

Вот несколько примеров использования тега <FONT>:

```
<FONT FACE="Arial">this is Arial</FONT>
<FONT FACE="Arial" SIZE="12">this is 12pt Arial</FONT>
<FONT FACE="Lucida Console" SIZE="+4" COLOR="#FF0000">this is red,
+4pt Lucida Console</FONT>
```

<P> (Paragraph Break)

Тег <P> разделяет абзацы, но во Flash он ведет себя совсем иначе, чем в HTML. Так, если для <P> нет закрывающего тега, то во Flash, в отличие от обычного HTML, строка не переводится. Обратите внимание на разницу между Flash и веб-броузером при выводе следующего текста HTML:

```
I hate filling out forms. <P>So sometimes I don't.
```

Вывод во Flash:

I hate filling out forms. So sometimes I don't.

Вывод в веб-броузере:

I hate filling out forms.
So sometimes I don't.

Чтобы добавить перевод строки, Flash нужен замыкающий тег </P>. Например:

```
<P> I hate filling out forms.</P> So sometimes I don't.
```

Кроме того, во Flash <P> вызывает появление одного перевода строки, как для <BR>, тогда как в веб-броузерах <P> приводит к появлению двух переводов строки. Возьмем такой текст:

```
<P>This is line one.</P><P>This is line two.</P>
```

Во Flash этот код выводится без промежуточной пустой строки:

This is line one.
This is line two.

В веб-броузере этот код выводится с промежутком между строками:

This is line one.

This is line two.

Поскольку во Flash поведение тега <P> не такое, как в веб-броузерах, мы часто используем вместо него тег <BR>. Однако атрибут ALIGN тега <P> удобен для выравнивания текста по центру, правому или левому краю, например:

```
<P ALIGN="CENTER">Centered text</P>
<P ALIGN="RIGHT">Right-justified text</P>
<P ALIGN="LEFT">Left-justified text</P>
```


 (Line Break)

Тег
 приводит к переводу строки в массиве текста и функционально эквивалентен escape-последовательности \n или ключевому слову newline. Рассмотрим такой код:

```
This is line one. <BR>This is line two.
This is line one. \nThis is line two.
```

Обе строки выводятся во Flash одинаково:

This is line one.
This is line two.

<A> (якорь или гипертекстовая ссылка)

Тег <A> создает гипертекстовую ссылку. При щелчке пользователя по тексту, помеченному тегом <A>, в броузер загружается документ, указанный в атрибуте HREF этого тега. Если фильм проигрывается в автономном проигрывателе, запускается веб-броузер по умолчанию для данной системы и документ загружается в этот броузер.

Общий синтаксис тега <A>:

```
<A HREF="documentToLoad.html">linked text</A>
```

Например, подключиться к хорошей игровой консоли можно с помощью

```
<A HREF="http://www.gamecube.com/">nice system</A>
```

Как и в HTML, URL может быть абсолютным или заданным относительно текущей страницы. Поскольку Flash не подчеркивает гиперссылки и не вы-

деляет их цветом автоматически, можно дополнить тег ссылки следующим образом:

```
<FONT COLOR="#0000FF">
<U><A HREF="http://www.gamecube.com/">nice console</A></U></FONT>
```

Обычно переход по ссылке влечет замену текущего фильма документом, указанным в атрибуте HREF тега якоря. Однако тег якоря может также открыть новое окно броузера. Атрибут TARGET позволяет задать имя окна, в которое должен быть загружен указанный в ссылке документ:

```
<A HREF="documentName" TARGET="windowName">linked text</A>
```

Если окна с именем windowName не существует, броузер откроет новое окно и присвоит ему имя windowName. Чтобы открыть новый документ в его собственном анонимном окне, можно воспользоваться ключевым словом "_blank":

```
<A HREF="mypage.html" TARGET="_blank">linked text</A>
```

Обратите внимание, что когда мы открываем окна с помощью атрибута TARGET, то не можем управлять размером или панелями инструментов нового окна. Чтобы открыть по ссылке окно конкретного размера, следует обратиться к JavaScript. Технология запуска вторичных окон с заданными параметрами с помощью JavaScript описывается на странице:

http://www.moock.org/webdesign/flash

Дополнительные сведения о том, как связаться с JavaScript из ActionScript, есть в описаниях глобальных функций *fscommand( )* и *getURL( )* в «Справочнике по языку» и в разделе «Выполнение JavaScript из ссылок HTML» далее в этой главе.

Помимо "_blank" атрибут TARGET может также иметь значение "_parent", "_self", либо "_top", либо имя окна или фрейма. Например, атрибут TARGET может загрузить во фрейм документ с помощью ссылки:

```
<A HREF="documentName" TARGET="frameName">linked text</A>
```

Во Flash теги якоря не всегда ведут себя в точности, как в HTML. Например, нельзя использовать атрибут NAME, поэтому невозможны внутренние ссылки в теле текста.

Во Flash теги якоря не добавляются в порядок обхода по табуляции в фильмах и потому недоступны с клавиатуры. Если требуется, чтобы содержимое было доступно с клавиатуры и других устройств ввода, следует пользоваться для ссылок кнопками, а не тегами якоря.

<TEXTFORMAT> (Поддержка форматирования)

Тег <TEXTFORMAT> – особое расширение Flash для HTML, с помощью которого функции форматирования абзацев класса *TextFormat* реализуются в синтаксисе HTML. Его атрибуты соответствуют свойствам leftMargin, rightMargin, blockIndent, indent, leading и tabStops класса *TextFormat*:

LEFTMARGIN

Задает (в пикселах) размер левого поля содержащегося текста. См. *TextFormat*.leftMargin.

RIGHTMARGIN

Задает (в пикселах) размер правого поля содержащегося текста. См. *TextFormat*.rightMargin.

BLOCKINDENT

Задает (в пикселах) размер отступа слева содержащегося текста. См. *TextFormat*.blockIndent.

INDENT

Задает (в пикселах) размер отступа для первой строки каждого абзаца содержащегося текста. См. *TextFormat*.indent.

LEADING

Задает (в пикселах) размер межстрочного промежутка для содержащегося текста. См. *TextFormat*.leading.

TABSTOPS

Задает позиции горизонтальной табуляции содержащегося текста в виде разделенных запятыми неотрицательных чисел. См. *TextFormat*.tabStops.

Следующий код показывает, как создать текстовое поле, в котором промежуток между строками по вертикали составляет 10 пикселов и отступ первых строк параграфов составляет 20 пикселов:

```
this.createTextField("output_txt", 1, 0, 0, 100, 200);
output_txt.html = true;
output_txt.border = true;
output_txt.wordWrap = true;
output_txt.type = "input";
output_txt.htmlText = "<TEXTFORMAT LEADING='10' INDENT='20'>"
                + "<P>This is the first paragraph.</P>"
                + "<P>This is the second paragraph.</P>"
                + "<P>And finally, this is the third paragraph.</P>"
                + "</TEXTFORMAT>";
```

<TAB> (Символ табуляции)

Тег <TAB> (не документирован) эквивалентен вставке в строку символа табуляции (ASCII 9). Применяется для перемещения текста в следующую позицию табуляции, когда свойство текстового поля *TextField*.condenseWhite имеет значение true (литеральные символы табуляции в этом случае подвергаются сжатию вместе с остальными пробельными символами).

Поддержка сущностей

Текстовые поля Flash с HTML поддерживают также специальные символьные сущности, перечисленные в табл. E.1. Когда такая сущность появляется

в свойстве *TextField*.htmlText текстового поля, Flash показывает на экране соответствующий символ. Однако другие стандартные сущности, например ©, и числовые сущности, например ™ (символ торговой марки), не поддерживаются. Для включения таких символов, как «копирайт», «торговая марка» или «евро», используйте текстовые файлы в формате Unicode, escape-последовательности Unicode или метод *String.fromCharCode( )* (см. «Кодировка символов» в главе 4). В следующей главе показано, как включить символ евро в HTML текстового поля:

```
this.createTextField("price_txt", 0, 100, 100, 200, 30);
price_txt.html = true;
// Евро представляется escape-полседовательностью \u20AC
price_txt.htmlText = "<B>FINAL COST:</B> \u20AC 200";
```

Таблица Е.1. Поддерживаемые сущности

Сущность	Представляемый символ
<	<
>	>
&	&
"	"
'	'
	неразрывный пробел

Заключение значений атрибутов в кавычки

В HTML, но не в его варианте для Flash, значения атрибутов HTML можно заключать в одинарные кавычки, двойные кавычки или оставлять вообще без кавычек. Например, в большинстве броузеров допустимы следующие теги:

```
<P ALIGN=RIGHT>
<P ALIGN='RIGHT'>
<P ALIGN="RIGHT">
```

Во Flash значения атрибутов без кавычек не допускаются. Например, синтаксис <P ALIGN=RIGHT> недопустим во Flash. Тем не менее в качестве ограничителей значений атрибутов могут использоваться как одинарные, так и двойные кавычки. При составлении значений текстовых полей, в которые входят атрибуты HTML, необходимо следить за правильным заключением атрибутов в кавычки, используя один тип кавычек для самой строки, а другой – для значений атрибутов. Например:

```
// Оба примера допустимы.
myText = "<P ALIGN='RIGHT'>hi there</P>";
myText = '<P ALIGN="RIGHT">hi there</P>';
// Этот пример приведет к ошибке, так как двойные кавычки
// используются для ограничения как строки, так и атрибута.
myText = "<P ALIGN="RIGHT">hi there</P>";
```

Дополнительные сведения о применении кавычек в строках см. в разделе «Строковые литералы» главы 4.

Нераспознаваемые теги и атрибуты

Как и веб-броузеры, Flash игнорирует теги и атрибуты, которые ему не известны. Например, если присвоить текстовому полю Flash текст HTML следующего вида:

```
<P>Please fill in and print this form</P>
<FORM><INPUT TYPE="TEXT"></FORM>
<P>Thank you!</P>
```

результат будет таким:

Please fill in and print this form
Thank you!

Элементы `FORM` и `INPUT` не поддерживаются Flash, поэтому оба они игнорируются. Аналогично, если использовать элементы контейнера типа `<TD>`, то содержимое сохраняется, а разметка игнорируется. Например,

```
myTextField = "<TABLE><TR><TD>table cell text</TD></TR></TABLE>";
```

выводит следующую строку без форматирования в табличном виде:

table cell text

Однако во Flash 5 и Flash 6 для тега, у которого отсутствует замыкающий, весь последующий текст считается частью тега и не будет выведен на экран. Например, если выполнить присваивание:

```
theField_txt.htmlText = "We all know that 5 < 20. That's obvious.";
```

Flash выведет:

We all know that 5

Чтобы включить символ < в текстовое поле HTML, воспользуйтесь сущностью `<`:

```
theField_txt.htmlText = "We all know that 5 &lt; 20. That's obvious.";
```

Теперь Flash покажет:

We all know that 5 < 20. That's obvious.

Дополнительную информацию о включении исходного кода HTML в текстовые поля HTML можно получить по адресу:

http://moock.org/asdg/technotes/sourceInHtmlField/

Использование HTML для ввода и вывода

Текст HTML, введенный в текстовое поле вручную с помощью инструмента Text, в среде разработки Flash не будет показан как HTML. Чтобы отобра-

зить HTML-форматирование текста на экране, нужно присвоить текст HTML свойству `htmlText` объекта *TextField* или, во Flash 5, переменной, ассоциированной с текстовым полем. Например:

```
// Синтаксис Flash 6
theField_txt.htmlText = "<P><B>Error!</B> <I>No email address!</I></P>";

// Синтаксис Flash 5
theField_txt = "<P><B>Error!</B> <I>No email address!</I></P>";
```

Во Flash 6 синтаксис Flash 5 не только не решит задачу, но и заменит объект *TextField* присвоенной строкой! Во Flash 6 необходимо применять свойство `htmlText`.

Во Flash Player можно ввести HTML в фильм с помощью обычного или с включенной поддержкой HTML текстового поля для ввода.

Когда обычный текст вводится в текстовое поле для ввода с включенной поддержкой HTML, в свойстве `htmlText` поля появляются автоматически добавляемые теги разметки. Например, введенный текст «Hi there» преобразуется в такое значение свойства `htmlText`:

```
'<P ALIGN="LEFT"><FONT FACE="Arial" SIZE="10" COLOR="#000000">Hi there</FONT></P>'
```

Когда символы HTML <, >, &, " и ' вводятся в текстовое поле для ввода с включенной поддержкой HTML, они представляются в свойстве `htmlText` сущностями `>`, `<`, `&`, `"` и `'`. Например, текст «<B>hi there</B>» преобразуется в такое значение свойства `htmlText`:

```
'<P ALIGN="LEFT"><FONT FACE="Arial" SIZE="10"
 COLOR="#000000">&lt;B&gt;hi there&lt;/B&gt;</FONT></P>'
```

В обоих случаях первоначальное (без разметки) значение доступно через свойство `text`.

Когда обычный или содержащий HTML текст вводится в обычное (с выключенной поддержкой HTML) текстовое поле ввода, модификации введенного текста не происходит. Обычные текстовые поля ввода позволяют ввести исходный код HTML без искажений. Это удобно, если требуется показать или принять теги HTML дословно, не игнорируя их и не интерпретируя.

Пример, демонстрирующий ввод данных в обычное или с включенной поддержкой HTML текстовое поле для ввода, есть в сетевом хранилище кода.

Выполнение JavaScript из ссылок HTML

В большинстве броузеров, поддерживающих JavaScript, можно выполнять команды JavaScript из якорных тегов, если указать протокол `javascript:` в значении атрибута `HREF`. Например:

```
<A HREF="javascript:square(5);">find the square of 5</A>
```

В ActionScript также можно выполнять команды JavaScript из тега <A>, например:

```
theField_txt.htmlText = "<A HREF='javascript:alert(5);'>display the number 5</A>";
```

Однако при задании строк в командах JavaScript необходимо использовать вместо кавычек сущность HTML ", например:

```
theField_txt.htmlText = "<A HREF=
                         'javascript:alert(&quot;hello world&quot;);'>"
                         + "display hello world</A>";
```

Вызов функций ActionScript из ссылок HTML

Хотя из тегов Flash <A> можно выполнять не любые предложения кода ActionScript, *функции* ActionScript из них вызывать можно. Для вызова функции ActionScript из якорного тега используется следующий синтаксис:

```
<A HREF="asfunction:myFunctionName">invoke the function</A>
```

Оператор вызова функции () недопустим и не должен проименяться при вызове функции ActionScript из тега якоря. Имя функции должно следовать непосредственно за ключевым словом asfunction:. Если между ними окажется пробел, функция не будет выполнена.

Из якорного тега можно не только вызвать функцию ActionScript, но и передать ей один параметр:

```
<A HREF="asfunction:myFunctionName,myParameter">invoke the function</A>
```

Здесь *myParameter* является значением передаваемого параметра. В вызванной функции *myParameter* всегда окажется строкой. Чтобы передать функции несколько фрагментов информации, нужно поместить в значение myParameter разделитель и самостоятельно разделить строку на части в функции. Например, вот вызов функции *roleCall( )*, которой передаются два значения, разделенных символом | :

```
<A HREF="asfunction:roleCall,megan|murray">invoke the function</A>
```

А вот сама функция *roleCall( )*. Обратите внимание на разделение значений с помощью метода *split( )*:

```
function roleCall (name) {
  var bothNames = name.split("|");
  trace("first name: " + bothNames[0]);  // Выводит: megan
  trace("last name: " + bothNames[1]);   // Выводит: murray
}
```

F

Поддержка GET и POST

В табл. F.1 перечислены двенадцать функций и методов, с помощью которых можно передать содержимое на сервер HTTP в запросе HTTP посредством метода GET или метода POST. По причине имеющихся ограничений и ошибок реализации, о которых рассказывается в этом приложении, лучших результатов можно добиться, применяя по возможности метод HTTP GET, особенно при разработке фильмов в формате Flash 5. Информацию Macromedia о специфических трудностях, которые применение POST вызывает во Flash Player, можно получить на следующих страницах:

http://www.macromedia.com/support/flash/ts/documents/flash_get_post.htm

http://www.macromedia.com/support/flash/ts/documents/flashplayer_r40.htm

Поддержка GET и POST через броузеры

Как на PC под Windows, так и на Макинтошах версии Flash Player 6 в виде подключаемого модуля или элемента ActiveX полностью поддерживают методы HTTP GET и POST. Однако несмотря на эту поддержку небольшие ошибки в некоторых сочетаниях броузер/проигрыватель приводят к эпизодическому отказу функций, использующих POST (см. приведенные выше URL). Кроме того, до версии Flash Player 6.0.40.0 применение метода POST приводило к проблемам с сервлетами Java, о чем говорится в следующем документе:

http://www.macromedia.com/support/flash/ts/documents/java_get.htm

Поддержка GET и POST в автономном проигрывателе

В табл. F.1 приводятся сведения об официальной поддержке методов HTTP GET и POST под Windows и на Макинтошах в Standalone Player 5 и Standalone Player 6 для каждой функции. Сделаем следующие общие замечания:

- Standalone Player 5 под Windows и на Макинтошах не поддерживает POST.
- Standalone Player 6 под Windows не поддерживает POST для функций, которые получают результат в веб-броузер, а не в проигрыватель.
- Даже если на вашей системе POST работает, следует при любой возможности применять GET, обеспечивающий наилучшие результаты на большинстве платформ.

Таблица F.1. Поддержка GET и POST как методов HTTP во Flash Standalone Player 5 и Standalone Player 6 для систем Windows и на Макинтошах

Функция	**Win GET F5**	**Mac GET F5**	**Win POST F5**	**Mac POST F5**	**Win GET F6**	**Mac GET F6**	**Win POST F6**	**Mac POST F6**
getURL( )	✓	✓	✗	✗[a]	✓	✓	✗[b]	✓
loadMovie( )	✓	✓	✗	✗[a]	✓	✓	✓	✓
loadMovieNum( )	✓	✓	✗	✗[a]	✓	✓	✓	✓
loadVariables( )	✓	✗[c]	✗	✗[a c]	✓	✓	✓	✓
loadVariablesNum( )	✓	✗[c]	✗	✗[a c]	✓	✓	✓	✓
LoadVars.send( )	N/A	N/A	N/A	N/A	✓	✓	✗[b]	✓
LoadVars.sendAndLoad( )	N/A	N/A	N/A	N/A	✓	✓	✓	✓
MovieClip.getURL( )	✓	✓	✗	✗[a]	✓	✓	✓	✓
MovieClip.loadMovie( )	✓	✓	✗	✗[a]	✓	✓	✓	✓
MovieClip.loadVariables( )	✓	✓	✗	✗[a]	✓	✓	✓	✓
XML.send( )	✓	✓	✗	✗[a]	✓	✓	✗[b]	✓
XML.sendAndLoad( )	✓	✓	✗	✗[a]	✓	✓	✓	✓

[a] Запросы POST преобразуются в запросы GET, но переменные не посылаются (то есть строка запроса отсутствует). По моим проверкам, POST работает в Standalone Player на Макинтошах, но Macromedia и сообщество разработчиков согласны с тем, что POST работает в этом проигрывателе ненадежно.

[b] Запросы POST преобразуются в запросы GET, а данные передаются в строке запроса GET.

[c] Метод *loadVariables( )* вообще не работает ни в Standalone Player ни в исполняемых проекторах (фильмах, упакованных вместе со Standalone Player в один исполняемый файл).

G

Обзор компонентов Flash UI

В данном приложении перечислены и кратко описаны методы и свойства, поддерживаемые компонентами Flash User Interface (FUI), которые устанавливаются вместе с Flash MX. Компоненты FUI предоставляют стандартные элементы интерфейса пользователя, такие как переключатели, флажки и окна списков. Мы выделили эту сводку их методов и свойств в отдельное приложение по следующим причинам:

- «Справочник по языку» по своему замыслу представляет полное и подробное описание всех свойств и методов класса. Здесь же мы приводим лишь краткую сводку методов и свойств, а не полную документацию по ним. Соответствующее полное описание заняло бы гораздо больше места (возможно, сотни страниц), что не укладывается в рамки этой книги.
- Данная книга посвящена основам языка ActionScript. Компоненты FUI фактически есть не что иное, как расширения Flash и ActionScript, и сами написаны на ActionScript. Они представляют собой важные примеры из бесчисленного количества расширений, о которых невозможно рассказать в одной книге. Возьмите, к примеру, компонент FDataGrid от Macromedia, который можно получить с Flash Exchange (*http://www.macromedia.com/ exchange/flash/*).
- Это приложение – краткий справочник по всем методам и свойствам компонентов FUI. Его можно держать открытым на столе во время работы, чтобы не прибегать к электронной подсказке Flash или выпадающим меню, чтобы узнать о существовании того или иного метода или свойства. Компоненты организованы не совсем так, как в электронной подсказке от Macromedia, кроме того, здесь приведен более подробный синтаксис компонентов (см. табл. G.1 и G.2).
- Мы решили, что лучше сгруппировать компоненты FUI по идейному принципу, а не по алфавиту, как в «Справочнике по языку», где их разъединили бы другие статьи, такие как `_focusrect` (глобальное свойство) и *fscommand( )* (функция).

Полную документацию по компонентам Flash UI можно найти в ресурсах, приведенных в конце главы 14.

FCheckBox

поддержка стандартного элемента UI «флажок» (checkbox)

Методы

getEnabled( )

Булево значение, указывающее на то, включен ли флажок (то есть реагирует ли на действия мыши).

getLabel( )

Возвращает строку – метку флажка.

getValue( )

Булево значение, указывающее на то, выставлен ли флажок (то есть выбран или не выбран).

registerSkinElement( )

Регистрирует скин для изменения внешнего вида флажка.

setChangeHandler( )

Задает функцию обратного вызова, которая вызывается, когда изменяется состояние флажка.

setEnabled( )

Включает и выключает флажок (не то же самое, что выбирает и сбрасывает выбор).

setLabel( )

Задает строку – метку флажка.

setLabelPlacement( )

Задает расположение метки относительно флажка («left» или «right»).

setSize( )

Задает ширину флажка в пикселах.

setStyleProperty( )

Задает для флажка определенное свойство стиля (список поддерживаемых свойств см. в объекте *FStyleFormat*).

setValue( )

Устанавливает флажок (`true`) или сбрасывает его (`false`).

FComboBox и FListBox

поддержка стандартных элементов UI «комбинированное окно списка», то есть выпадающий список (combo box) и «окно списка» (list box)

В табл. G.1 сопоставляются методы объектов *FComboBox* и *FListBox*. Обратите внимание на общность большинства методов и поддержку *FListBox* выделения нескольких элементов одновременно.

Таблица G.1. Методы FComboBox и FListBox

Метод	Описание	FComboBox	FListBox
addItem( )	Добавляет элемент к списку элементов окна	✓	✓
addItemAt( )	Добавляет элемент к списку в указанном месте	✓	✓
getEnabled( )	Булево значение, указывающее на то, активен элемент или затенен	✓	✓
getItemAt( )	Возвращает элемент списка в заданной позиции	✓	✓
getLength( )	Возвращает количество элементов в списке	✓	✓
getRowCount( )	Возвращает количество строк, показываемых в прокручиваемом списке	✓	✓
getScrollPosition( )	Возвращает номер верхней отображаемой строки	✓	✓
getSelectedIndex( )	Возвращает индекс элемента, выделенного в данный момент	✓	✓
getSelectedIndices( )	Возвращает массив индексов выделенных элементов	×	✓
getSelectedItem( )	Возвращает выделенный элемент списка как объект	✓	✓
getSelectedItems( )	Возвращает массив выделенных элементов списка как объектов	×	✓
getSelectMultiple( )	Булево значение, указывающее на возможность выделения в окне списка нескольких элементов	×	✓
getValue( )	Возвращает значение выделенного элемента	✓	✓
registerSkinElement( )	Регистрирует элемент скина для изменения внешнего вида компонента	✓	✓
removeAll( )	Удаляет из списка все элементы	✓	✓
removeItemAt( )	Удаляет элемент в заданной позиции списка	✓	✓
replaceItemAt( )	Удаляет элемент в заданной позиции списка	✓	✓
setAutoHideScrollBar( )	Булево значение, определяющее, нужно ли убрать полосу прокрутки, если она не нужна	×	✓
setChangeHandler( )	Задает функцию обратного вызова, которая вызывается, когда изменяется состояние списка	✓	✓
setDataProvider( )	Задает массив элементов для объекта DataProvider, который заполняет список	✓	✓

Таблица G.1 (продолжение)

Метод	Описание	FComboBox	FListBox
setEditable( )	Булево значение, определяющее возможность редактирования содержимого комбинированного списка	✓	×
setEnabled( )	Включает или выключает (то есть деактивирует) окно списка	✓	✓
setItemSymbol( )	Задает ID символа библиотеки графического элемента для показа содержимого списка	✓	✓
setRowCount( )	Задает максимальное количество строк в списке	✓	✓
setScrollPosition( )	Вызывает прокрутку окна списка до нужного элемента	×	✓
setSelectedIndex( )	Выделяет элемент в списке по номеру его строки (отсчитываемому от 0)	✓	✓
setSelectedIndices( )	Задает массив индексов элементов, которые нужно выделить	×	✓
setSelectMultiple( )	Булево значение, указывающее на возможность выделения в окне списка нескольких элементов	×	✓
setSize( )	Задает ширину комбинированного списка или окна списка в пикселах (для окна списка может задавать ширину)	✓	✓
setStyleProperty( )	Задает для комбинированного списка или окна списка определенное свойство стиля (список поддерживаемых свойств см. в объекте FStyleFormat)	✓	✓
setValue( )	Задает текущее значение комбинированного окна списка	✓	×
setWidth( )	Задает ширину окна списка в пикселах	×	✓
sortItemsBy( )	Сортирует элементы списка по их меткам или содержащимся данным	✓	✓

FPushButton

поддержка стандартного элемента UI «кнопка» (pushbutton)

Методы

getEnabled( )

Булево значение, указывающее на то, включена ли кнопка (то есть реагирует на действия мыши).

getLabel( )

Возвращает строку – метку кнопки.

registerSkinElement( )

Регистрирует скин для изменения внешнего вида кнопки.

setClickHandler( )

Задает функцию обратного вызова, которая вызывается при щелчке по кнопке.

setEnabled( )

Включает и выключает кнопку (то есть определяет ее активность или затенение).

setLabel( )

Задает строку – метку кнопки.

setSize( )

Задает высоту и ширину кнопки в пикселах.

setStyleProperty( )

Задает для кнопки определенное свойство стиля (список поддерживаемых свойств см. в объекте *FStyleFormat*).

FRadioButton и FRadioButtonGroup

поддержка стандартных элементов UI «переключатель» (radio button) и «группа переключателей»

В табл. G.2 сопоставляются методы объектов *FRadioButton* и *FRadioButtonGroup*. Обратите внимание на общность некоторых методов, хотя для *FRadioButtonGroup* метод относится к группе переключателей в целом, а не к отдельной кнопке.

Таблица G.2. Методы FRadioButton и FRadioButtonGroup

Метод	Описание	FRadioButton	FRadioButtonGroup
getData( )	Возвращает данные, ассоциированные с переключателем	✓	×
getEnabled( )	Булево значение, указывающее на то, активен переключатель (группа) или затенен	✓	✓
getGroupName( )	Возвращает имя переключателя или группы	✓	✓
getLabel( )	Возвращает метку переключателя	✓	×
getState( )	Возвращает состояние переключателя – выбран (`true`) или не выбран (`false`)	✓	×
getValue( )	Возвращает значение выбранной в группе кнопки	×	✓
registerSkinElement( )	Регистрирует элемент скина для изменения внешнего вида переключателя или группы	✓	✓

Таблица G.2 (продолжение)

Метод	Описание	FRadioButton	FRadioButtonGroup
setChangeHandler( )	Задает функцию обратного вызова, которая вызывается при изменении состояния переключателя или группы	✓	✓
setData( )	Ассоциирует с переключателем элемент данных	✓	×
setEnabled( )	Включает или выключает (то есть деактивизирует) переключатель или группу	✓	✓
setGroupName( )	Задает имя переключателя или группы	✓	✓
setLabel( )	Задает строку – метку переключателя	✓	×
setLabelPlacement( )	Задает расположение меток в переключателе или группе («left» или «right»)	✓	✓
setSize( )	Задает ширину переключателя или группы в пикселах	✓	✓
setState( )	Задает состояние переключателя – выбран (`true`) или не выбран (`false`)	✓	×
setStyleProperty( )	Задает для переключателя или группы определенное свойство стиля (список поддерживаемых свойств см. в объекте FStyleFormat)	✓	✓
setValue( )	Выбирает переключатель в группе по значению его данных	×	✓

FScrollBar

Поддержка стандартного элемента UI «полоса прокрутки» (scrollbar), связанного с текстовым полем посредством setScrollTarget()

Методы

getEnabled( )

Булево значение, указывающее на то, что полоса прокрутки включена (то есть реагирует на действия мыши).

getScrollPosition( )

Возвращает текущее положение прокрутки как целое число между минимальным и максимальным положениями.

registerSkinElement( )

Регистрирует скин для изменения внешнего вида полосы прокрутки.

setChangeHandler( )

Задает функцию обратного вызова, которая вызывается при прокрутке компонента.

setEnabled( )

Включает или выключает полосу прокрутки.

setHorizontal( )

Булево значение, задающее горизонтальное (`true`) или вертикальное (`false`) расположение полосы прокрутки.

setLargeScroll( )

Задает количество единиц прокрутки при щелчке пользователя у края ползунка (по умолчанию равно `pageSize`, заданному с помощью *setScrollProperties( )*).

setScrollPosition( )

Задает текущее положение прокрутки как целое число между минимальным и максимальным положениями.

setScrollProperties( )

Задает минимальное и максимальное положения полосы прокрутки и размер страницы.

setScrollTarget( )

Задает текстовое поле, которым управляет полоса прокрутки, в виде строки с именем экземпляра.

setSize( )

Задает длину полосы прокрутки (ее ширина фиксирована).

setSmallScroll( )

Задает количество единиц прокрутки при щелчке пользователя по стрелке полосы прокрутки (по умолчанию 1).

setStyleProperty( )

Задает для полосы прокрутки определенное свойство стиля (список поддерживаемых свойств см. в объекте *FStyleFormat*).

FScrollPane

поддержка окна с прокруткой для представления фильма, клипа или файла JPEG

Методы

getPaneHeight( )

Возвращает высоту окна прокрутки, задаваемую *setSize( )*.

getPaneWidth( )

Возвращает ширину окна прокрутки, задаваемую *setSize( )*.

getScrollContent( )

Возвращает ссылку на фильм, показываемый в области прокрутки.

getScrollPosition( )

Возвращает текущее положение прокрутки в виде объекта со свойствами `x` и `y`.

loadScrollContent( )

Загружает файл *.swf* или *.jpg* в окно прокрутки для показа.

refreshPane( )

Обновляет окно прокрутки (полезно, когда меняется размер содержимого).

registerSkinElement( )

Регистрирует скин для изменения внешнего вида окна прокрутки.

setDragContent( )

Булево значение, определяющее возможность прокрутки содержимого путем буксировки.

setHScroll( )

Задает показ горизонтальной полосы прокрутки: всегда (`true`), никогда (`false`), при необходимости (`"auto"`).

setScrollContent( )

Задает символ библиотеки или экземпляр клипа для показа в окне прокрутки.

setScrollPosition( )

Задает текущее положение прокрутки с помощью координат `x` и `y`.

setSize( )

Задает ширину и высоту окна прокрутки.

setStyleProperty( )

Задает для окна прокрутки определенное свойство стиля (список поддерживаемых свойств см. в объекте *FStyleFormat*).

setVScroll( )

Задает показ вертикальной полосы прокрутки: всегда (`true`), никогда (`false`), при необходимости (`"auto"`).

FStyleFormat — поддержка применения стилей к элементам Flash UI Components

Методы

addListener( )

Добавляет компоненты UI к списку элементов, обновляемых при изменеии *FStyleFormat*.

applyChanges( )

Применяет изменения стилей и уведомляет (обновляет) все приемники.

removeListener( )

Удаляет приемник события изменения формата.

Свойства

arrow

Задает RGB-значение цвета стрелок, используемых в полосах прокрутки, окнах списков и т. д.

background

Задает RGB-значение цвета фона для ассоциированных компонентов UI.

backgroundDisabled

Задает RGB-значение цвета фона для отключенных компонентов UI.

check

Задает RGB-значение цвета галочек флажков.

darkshadow

Задает RGB-значение цвета темной тени в ассоциированных переключателях, флажках и т. д.

embedFonts

Определяет, встраиваются ли шрифты для текста в компонентах UI.

face

Задает RGB-значение основного цвета, используемого для ассоциированных компонентов UI.

focusRectInner

Задает RGB-значение цвета внутреннего прямоугольника фокуса компонентов UI.

focusRectOuter

Задает RGB-значение цвета внешнего прямоугольника фокуса компонентов UI.

foregroundDisabled

Задает RGB-значение цвета переднего плана отключенных компонентов UI.

highlight

Задает RGB-значение цвета для выделенных компонентов UI.

highlight3D

Задает RGB-значение цвета для 3D-выделения компонентов UI.

radioDot

Задает RGB-значение цвета точек в кнопках переключателей.

scrollTrack

Задает RGB-значение цвета фона полосы прокрутки.

selection

Задает RGB-значение цвета для выделений.

selectionDisabled

Задает RGB-значение цвета для отключенных выделений.

selectionUnfocused

Задает RGB-значение цвета выделений, не находящихся в фокусе.

shadow

Задает RGB-значение цвета теней.

textAlign

Задает выравнивание текста («right», «left» или «center»).

textBold

Булево значение, определяющее необходимость применения полужирного начертания шрифта.

textColor

Задает RGB-значение цвета текста в компонентах UI.

textDisabled

Задает RGB-значение цвета неактивного текста.

textFont

Задает шрифт текста в компонентах UI.

textIndent

Задает отступ текста в компонентах UI как целое число пикселов.

textItalic

Булево значение, определяющее необходимость применения курсивного начертания шрифта.

textLeftMargin

Задает размер левого поля для текста в компонентах UI как целое число пикселов.

textRightMargin

Задает размер правого поля для текста в компонентах UI как целое число пикселов.

textSelected

Задает RGB-значение цвета выделенного текста в компонентах UI.

textSize

Задает размер шрифта текста в пикселах.

textUnderline

Булево значение, определяющее необходимость подчеркивания текста.

H

Встраивание фильмов Flash в веб-страницы

Для встраивания фильма Flash в веб-страницу применяются теги HTML <EMBED> и <OBJECT>, аналогично тому, как с помощью тега <IMG> на странице размещается файл GIF или JPEG. Последним версиям Internet Explorer (IE) для Windows (IE5.5 SP2 и выше) требуется тег <OBJECT> (тег <OBJECT> предпочтительнее тега <EMBED> даже в более ранних версиях IE для Windows, которые поддерживали оба тега). IE для Макинтошей и Netscape на обеих платформах требуют тега <EMBED>, как и многие менее распространенные броузеры. Поэтому для того, чтобы поддерживались все броузеры, мы вкладываем тег <EMBED> внутрь тега <OBJECT>, как показано в примере H.1, встраивающем в страницу HTML фильм *main.swf*.

Пример H.1. Теги <OBJECT> и <EMBED> для встраивания файла .swf в страницу HTML

```
<OBJECT
  CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  CODEBASE=
"http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=6,0,0,0"
  WIDTH="550"
  HEIGHT="400"
  ID="main"
  ALIGN="">
  <PARAM NAME="movie" VALUE="main.swf">
  <PARAM NAME="quality" VALUE="high">
  <PARAM NAME="bgcolor" VALUE="#FFFFFF">

    <EMBED
      SRC="main.swf"
      QUALITY="high"
      BGCOLOR="#FFFFFF"
      WIDTH="550"
      HEIGHT="400"
      NAME="main"
```

```
        ALIGN=""
        TYPE="application/x-shockwave-flash"
        PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
      </EMBED>
  </OBJECT>
```

В среде разработки Flash команда File → Publish генерирует теги <EMBED> и <OBJECT> автоматически. Инструкции можно найти в Help → Using Flash → Publishing → Publishing Flash documents.

Дополнительные сведения о тегах <EMBED> и <OBJECT> см. в Help → Using Flash → Publishing → Editing Flash HTML settings. См. также следующие технические замечания:

http://www.macromedia.com/support/flash/ts/documents/tn4150.html

http://www.macromedia.com/support/flash/ts/documents/tag_attributes.htm

Среда разработки Flash предоставляет помощь, но многие разработчики предпочитают кодировать HTML вручную, и данное приложение предназначено для них. Здесь рассказано также об атрибутах, не поддерживаемых средой разработки.

В табл. Н.1 перечислен минимум атрибутов, рекомендуемых к применению в тегах <EMBED> и <OBJECT>, а некоторые из них являются строго обязательными. За таблицей следует подробное описание этих и других атрибутов.

Таблица Н.1. Рекомендованные атрибуты тегов <OBJECT> и <EMBED>

Описание	Атрибуты <OBJECT>	Атрибуты <EMBED>
Идентифицирует содержимое как принадлежащее Flash	CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8- 444553540000"	TYPE="application/x-shockwave-flash"
Адрес подключаемого к броузеру модуля Flash Player	CODEBASE="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=6,0,0,0"	PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer"
URL файла *.swf*	<PARAM NAME="MOVIE" VALUE="*url*">	SRC="*url*"
Высота фильма	HEIGHT="*height*"	HEIGHT="*height*"
Ширина фильма	WIDTH="*width*"	WIDTH="*width*"

Атрибуты тега <OBJECT>

Для использования в теге <OBJECT> атрибутов, отличных от CLASSID, CODEBASE, WIDTH, HEIGHT, ID и ALIGN, создайте тег <PARAM> следующего вида:

```
<PARAM NAME=ATTR_NAME VALUE=VAL>
```

Здесь *ATTR_NAME* представляет имя атрибута, а *VAL* – его значение (как в примере Н.1).

Следующий список атрибутов относится исключительно к тегу <OBJECT>.

CLASSID

Идентификатор управляющего элемента Flash ActiveX, который должен иметь значение "clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" для любой версии проигрывателя.

CODEBASE

Адрес управляющего элемента Flash ActiveX, который должен иметь значение "http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=6,0,0,0". Его использует IE для Windows, чтобы загрузить версию Flash Player в виде ActiveX, если он не установлен или его версия более старая, чем указанная в #version=w,x,y,z. Если отсутствует #version, то броузер не пытается обновлять имеющийся на машине ActiveX Player, но если проигрыватель не будет обнаружен, попытается загрузить новейшую версию проигрывателя.

ID

Строка идентификации, которой сценарии могут руководствоваться для управления фильмом Flash. Она должна совпадать с атрибутом NAME в теге <EMBED>. См. описание *fscommand( )* в «Справочнике по языку» и *http://www.moock.org/webdesign/flash/fscommand*.

MOVIE

Адрес фильма (файл *.swf*) в виде относительного или абсолютного URL, например "mymovie.swf". Должен совпадать с атрибутом SRC в теге <EMBED>.

Дополнительные сведения см. в табл. Н.1 и разделе «Общие атрибуты тегов <OBJECT> и <EMBED>» далее в этом приложении.

Атрибуты тега <EMBED>

Ниже следует список атрибутов, применяющихся исключительно в теге <EMBED>.

NAME

Строка идентификации, которой сценарии могут руководствоваться для управления фильмом Flash. Должна совпадать с атрибутом ID в теге <OBJECT>. См. описание *fscommand( )* в «Справочнике по языку» и *http://www.moock.org/webdesign/flash/fscommand*.

PLUGINSPAGE

Адрес подключаемого модуля Flash на сайте Macromedia, который Netscape показывает пользователю, предлагая загрузить модуль, если он еще не установлен. Должен иметь значение "http://www.macromedia.com/go/getflashplayer".

SRC

Адрес фильма (файл *.swf*) в виде относительного или абсолютного URL, например "mymovie.swf". Должен совпадать с атрибутом MOVIE в теге <OBJECT>.

SWLIVECONNECT

Если этот атрибут имеет значение `true`, броузер загружает Java, когда впервые загружается Flash Player. По умолчанию равен `false` (броузер не загружает Java, если он уже не выполняется). Для фильмов, управляемых JavaScript или использующих *fscommand( )*, надо задать для `SWLIVECONNECT` значение `true`.

TYPE

MIME-тип содержимого для Flash. Требуется Netscape, если веб-сервер не указывает тип MIME для файлов *.swf*. Если ваша страница HTML загружает Flash Player в IE для Windows, но не в других броузерах, проверьте, правильно ли задан `TYPE`. Должен иметь значение `"application/x-shockwave-flash"`.

Дополнительные сведения см. в табл. Н.1 и разделе «Общие атрибуты тегов <OBJECT> и <EMBED>».

Общие атрибуты тегов <OBJECT> и <EMBED>

Следующий список атрибутов применим к обоим тегам – `<OBJECT>` и `<EMBED>`. Их применение демонстрирует пример Н.1.

ALIGN

Располагает фильм в окне броузера. Если атрибут опущен, фильм располагается в центре, но явного значения `"CENTER"` нет. Допустимы следующие значения: `""` (без значения, располагать в центре), `"LEFT"`, `"RIGHT"`, `"TOP"` и `"BOTTOM"`.

BASE

URL, выступающий в качестве базового для всех относительных URL в фильме (например, URL, используемых методами *getURL( )*, *loadMovie( )*, *XML.send( )* и т. д.).

BGCOLOR

Цвет фона фильма в виде шестнадцатеричного триплета RGB – `#RRGGBB`, где `RR`, `GG` и `BB` представляют собой двузначные шестнадцатеричные числа, соответствующие красной, зеленой и синей составляющим. Имеет приоритет над значением, заданным в среде разработки с помощью Modify → Document.

HEIGHT

Высота фильма в пикселах или в процентном отношении к ширине страницы HTML в броузере.

LOOP

Задает непрерывное циклическое воспроизведение анимации Flash (если атрибут равен `true` – значению по умолчанию) или остановку (при `false`), когда головка воспроизведения достигает конца фильма.

MENU

Указывает, будет ли контекстное меню Flash Player содержать полный список элементов (`true`) или минимальный набор (`false`). По умолчанию имеет значение `true`, но часто задается равным `false`, чтобы помешать пользователю управлять фильмом нежелательным образом, например, останавливая, перематывая в начало или изменяя масштаб. Доступ к контекстному меню осуществляется по щелчку правой кнопкой мыши в Windows или посредством Ctrl-щелчка на Макинтошах.

PLAY

Если атрибут равен `true` (значение по умолчанию), то воспроизведение фильма начинается сразу после загрузки. А если `false`, то воспроизведение фильма инициируется через ActionScript с помощью функций *play( )* или *gotoAndPlay( )*.

QUALITY

Задает сглаживание текста и графики, управляя качеством воспроизведения и производительностью. Допустимы значения `"BEST"`, `"HIGH"`, `"AUTOHIGH"`, `"MEDIUM"`, `"AUTOLOW"` и `"LOW"`.

SALIGN

Определяет расположение фильма в пространстве, которое доступно Flash Player. Допустимые значения и применение см. в табл. С.20 в статье, посвященной `Stage.align` в «Справочнике по языку».

SCALE

Управляет масштабированием. Допустимы значения `"EXACTFIT"`, `"NOBORDER"`, `"NOSCALE"` и `"SHOWALL"`. Применение описывается в статье о `Stage.scaleMode` в «Справочнике по языку».

WIDTH

Ширина фильма в пикселах или в процентном отношении к ширине страницы HTML в броузере.

WMODE

Позволяет воспроизводить фильмы Flash без окон, обеспечивая с помощью слоев HTML более гладкую интеграцию с элементами, отличными от Flash. Поддерживаются значения `"WINDOW"` (по умолчанию), `"OPAQUE"` (непрозрачный) и `"TRANSPARENT"` (прозрачный). Во Flash Player 6.0.61.0 добавлена поддержка атрибута `WMODE` для Mac OS X и Netscape под Windows. Предшествовавшие версии поддерживали `WMODE` только в Internet Explorer для Windows. Подробности см. в примечаниях к релизу Flash Player 6.0.61.0 или в Help → Using Flash.

Доступ на этапе исполнения, влияющий на атрибуты HTML

В табл. Н.2 перечислены атрибуты HTML, которые относятся к ActionScript или могут изменяться им на этапе исполнения.

Таблица Н.2. Атрибуты тегов <EMBED> и <OBJECT>, относящиеся к этапу исполнения

Доступ на этапе исполнения	Атрибут <OBJECT>	Атрибут <EMBED>
Для загрузки дополнительных фильмов в проигрыватель применяется *loadMovie( )*	<PARAM NAME="MOVIE" VALUE="*url*">	SRC="*url*"
С помощью `_root._url` определить адрес, с которого загружен фильм	BASE="*url*"	BASE="*url*"
Начертить цветной прямоугольник с `depth`, равным 16 384, с помощью *MovieClip.beginFill( )*, `Stage.height` и `Stage.width`	<PARAM NAME="bgcolor" VALUE="#FFFFFF">	BGCOLOR=="#FFFFFF"
Требует Flash Player 6. Подробности см. в разделе «Загрузка внешних переменных» в главе 2, в том числе об альтернативах для Flash Player 5	<PARAM NAME="FLASHVARS" VALUE="AUTHOR=Moock ">	FLASHVARS="AUTHOR=Moock"
С помощью *gotoAndPlay( )* воспроизводить фильм с кадра 1	<PARAM NAME="LOOP" VALUE="TRUE">	LOOP="TRUE"
`Stage.showMenu` (требует Flash Player 6). Для Flash Player 5 следует применять *fscommand("showmenu", "false")* и *fscommand("showmenu", "true")*	<PARAM NAME="MENU" VALUE="TRUE">	MENU="TRUE"
С помощью *stop( )* остановить фильм в кадре 1	<PARAM NAME="PLAY" VALUE="TRUE">	PLAY="TRUE"
`_quality`	<PARAM NAME="QUALITY" VALUE="HIGH">	QUALITY="HIGH"
`Stage.align` (требует Flash Player 6)	<PARAM NAME="SALIGN" VALUE="LT">	SALIGN="LT"
`Stage.scaleMode` (требует Flash Player 6). Для Flash Player 5 следует применять *fscommand("allowscale")*	<PARAM NAME="SCALE" VALUE="NOSCALE">	SCALE="NOSCALE"

Алфавитный указатель

Специальные символы

/* */, маркеры комментариев, 445
– (минус), оператор вычитания, 121, 158
–, унарный оператор отрицания, 160
––, оператор декрементирования, 158
+, оператор сложения, 46, 121, 157
преобразование данных в строки, 108
++, оператор инкрементирования, 157
*, оператор умножения, 121, 159
*=, оператор умножения с присваиванием, 154
/, оператор деления, 121
/=, оператор деления с присваиванием, 154
!, оператор логического отрицания (Flash 5), 176
!=, оператор неравенства (Flash 5), 130, 165
%, оператор деления по модулю, 160
%=, оператор деления по модулю с присваиванием, 154
, (запятая), оператор, 178
. (точка), оператор, 180
доступ к свойствам клипа, 375
доступ к свойствам объекта, 324
обход иерархии экземпляров клипов, 398
; (точка с запятой), завершение инструкций с помощью, 226, 442
?: условный оператор, 181, 200
[] (квадратные скобки), оператор элемента массива/свойства объекта, 180, 293
обращение к свойствам объекта, 325
создание динамических ссылок на объекты клипов, 404
() круглые скобки
оператор вызова функции, 225
оператор группирования, 177
приоритеты операторов и, 152
присваивание и вызов методов, 333
{ } фигурные скобки (инициализаторы объекта/ограничители блока инструкций), 187
<>, оператор неравенства (Flash 4), 165
=, оператор присваивания, 155
= =, оператор равенства, 166
Flash 5, 129, 161
^=, оператор поразрядного исключающего ИЛИ с присваиванием, 154
|=, оператор поразрядного ИЛИ с присваиванием, 154
||, оператор логического ИЛИ (Flash 5), 173
&, оператор конкатенации (Flash 4), 1051
&&, оператор логического И (Flash 5), 175
&=, оператор поразрядного И с присваиванием, 154
< (меньше), оператор (Flash 5), 46
<<=, оператор поразрядного сдвига влево с присваиванием, 154
> (больше), оператор (Flash 5), 131
>= (больше или равно), оператор, 169
>>=, оператор поразрядного сдвига вправо со знаком с присваиванием, 154
>>>=, оператор поразрядного беззнакового сдвига вправо с присваиванием, 154

A

<A>, тег (HTML), 1062
abs(), метод (Math, объект), 657
Accessibility, объект
методы
isActive(), 489
sendEvent(), 489
acos(), метод (Math, объект), 658
Actions, панель
Expert Mode, режим, 452
Normal Mode, режим, 451
переключение из Expert Mode, 452
Parameters, окно, 452

Script, окно, 450
добавление инструкций в, 451
добавление кода в кнопки, 454
Toolbox, окно, 450
методы поиска кода, 456
открытие, 39
перекрытие методов и глобальных функций, 411
подсказка кода, 76
режимы редактирования, 451
задание, 452
ActionScript
(*см. также* Flash), 26
аргументы, 45
веб-сайты, использующие, 29
возможности, 48
Flash-программы, 48
вопросы обратной совместимости, 1042
зарезервированные слова, 446
идентификаторы, правила образования, 447
импорт текста из внешних файлов, 458
инструкции (*см.* инструкции), 42
условные, 50
интерактивность в, 27
интерпретатор, 44
использование цепочки областей видимости, 238
преобразование данных, 104, 164
удаление локальных переменных, 93
категорирование данных, 103
лексическая структура, 441
максимальные/минимальные значения, 120, 774
модель исполнения, управляемого событиями, 56, 263
наследование, 348, 359
новые характеристики, 17
обзор, 26
операторы, 46
отключение автопилота, 41
переменные (*см.* переменные), 43
программирование, введение в, 37
различия между EMCA-262 и JavaScript, 1055
ресурсы, 1029
связь из JavaScript с, 1063, 1067
связь с JavaScript
fscommand() и, 577
связь с сервером, 28
синхронное выполнение кода, 263
среда разработки (*см.* среда разработки), 450
сценарии, 48
типы данных, 100
управление
визуальным содержимым, 27
временной диаграммой, 27
звуковым содержимым, 27
функции, 52
циклы, 51
чувствительность к регистру в, 447
эквивалентность методов и свойств Java и ActionScript, 338
Active Server Pages (ASP), ресурсы, 1032
addListener(), метод
Key, объект, 612
Mouse, объект, 672
Selection, объект, 808
Stage, объект, 854
TextField, класс, 890
addProperty(), метод (Object, класс), 778
align, свойство
Stage, объект, 855
TextFormat, класс, 953
allowDomain(), обработчик события
LocalConnection, класс, 647
System.security, объект, 878, 881
allowscale, команда (автономный проигрыватель), 578
_alpha, свойство
Button, класс, 511
MovieClip, класс, 679, 686
TextField, класс, 891
and, оператор логического И (Flash 4), 446
appendChild(), метод (XML, класс), 984
apply(), метод (Function, класс), 582
arguments, объект, 491
доступ к лишним параметрам, 243
извлечение значений параметров из массива, 243
ошибка реализации, 493
свойства
callee, 244, 491
caller, 244, 492
length, 493
Array(), конструктор, 294
Array, класс, 493
методы
concat(), 305, 495
join(), 312, 496
pop(), 307, 497
push(), 301, 498
reverse(), 309, 498
shift(), 308, 498

slice(), 164, 499
sort(), 310, 500
sortOn(), 310, 501
splice(), 303, 309, 503
toString(), 313, 504
unshift(), 302, 504
свойства
length, 297, 496
расширение массивов без присвоения значений элементам, 301
удаление элементов массива, 306
ASCII, набор символов, 122
ASCII-коды клавиш, 613, 1035
asin(), метод (Math, объект), 658
ASP (Active Server Pages), ресурсы, 1032
atan(), метод (Math, объект), 659
atan2(), метод (Math, объект), 659
attachMovie(), метод (MovieClip, класс), 375, 384, 687
добавление созданных клипов в стек, 391
создание экземпляров, 382
attachSound(), метод (Sound, класс), 832
attributes, свойство (XML, класс), 985
autoSize, свойство (TextField, класс), 892

B

<B>, тег (HTML), 1059
background, свойство (TextField, класс), 894
backgroundColor, свойство (TextField, класс), 894
BACKSPACE, свойство (Key, объект), 610
Ball, класс (пример), 333
Ball, конструктор, модификация кода, 434
MovieClip, компоненты, создание, 437
MovieClip, надкласс, назначение, 429
вывод на экран, 345
вызов конструктора подкласса, 426
объединение кода подкласса и символа библиотеки, 430
пример подкласса, 423
создание символов библиотеки, 426
beginFill(), метод (MovieClip, класс), 689
beginGradientFill(), метод (MovieClip, класс), 694
blockIndent, свойство (TextFormat, класс), 954
bold, свойство (TextFormat, класс), 955
Boolean, класс, 506
методы
toString(), 506
valueOf(), 507
Boolean, тип данных
правила равенства для, 162
преобразование в, 106, 109
целые числа, 145
Boolean(), глобальная функция, 109, 505
border, свойство (TextField, класс), 895
borderColor, свойство (TextField, класс), 895
bottomScroll, свойство (TextField, класс), 896

, тег (HTML), 1062
break, инструкция, 188, 215
bullet, свойство (TextFormat, класс), 956
Button, класс, 507
методы
getDepth(), 513
обработчики событий
keyPress, 514
onDragOut(), 516
onDragOver(), 516
onKillFocus(), 517
onPress(), 518
onRelease(), 519
onReleaseOutside(), 519
onRollOut(), 520
onRollOver(), 521
onSetFocus(), 521
свойства
_alpha, 511
_focusrect, 512
_height, 513
_name, 515
_parent, 522
_rotation, 523
_target, 524
_url, 526
_visible, 527
_width, 527
_x, 528
_xmouse, 529
_xscale, 529
_y, 529
_ymouse, 530
_yscale, 530
enabled, 511
tabEnabled, 523
tabIndex, 524
trackAsMenu, 525
useHandCursor, 526

C

call(), глобальная функция, 530
call(), метод (Function, класс), 585
callee, свойство (arguments, объект), 244, 491, 492
caller, свойство (arguments, объект), 492, 493
Capabilities, объект, 531
    свойства
        hasAccessibility, 533
        hasAudio, 534
        hasAudioEncoder, 534
        hasMP3, 534
        hasVideoEncoder, 535
        input, 535
        isDebugger, 536
        language, 536
        manufacturer, 537
        os, 538
        pixelAspectRatio, 539
        screenColor, 539
        screenDPI, 540
        screenResolutionX, 540
        screenResolutionY, 541
        serverString, 542
        version, 542
capabilities, свойство (System, объект), 878
Caps Lock, клавиша и isToggled(), 617
CAPSLOCK, свойство (Key, объект), 610
case, ключевое слово в инструкции switch, 204
ceil() (Math, объект), 661
charAt() (String, класс), 134, 867
charCodeAt() (String, класс), 144, 868
childNodes, свойство (XML, класс), 986
clear(), метод (MovieClip, класс), 700
clearInterval(), глобальная функция, 544
cloneNode(), метод (XML, класс), 987
close(), метод
    LocalConnection, класс, 650
    XMLSocket, класс, 1020
Code Depot, хранилище файлов для обучения, 28
Color, класс, 545
    методы
        getRGB(), 549
        getTransform(), 550
        setRGB(), 551
        setTransform(), 553
    объекты, создание, 545
color, свойство (TextFormat, класс), 957
Comm Server (*см.* Macromedia Flash Communication Server MX), 14
concat(), метод
    Array, класс, 305, 495
    String, класс, 868
        в сравнении с оператором +=, 129
condenseWhite, свойство (TextField, класс), 897
connect(), метод
    LocalConnection, класс, 650
    XMLSocket, класс, 1020
constructor, свойство (Object, класс), 351, 783, 1058
contentType, свойство
    LoadVars, класс, 633
    XML, класс, 988
continue, инструкция, 188, 216
CONTROL, свойство (Key, объект), 610
copyright, символ, 144
cos(), метод (Math, объект), 661
createElement(), метод (XML, класс), 989
createEmptyMovieClip(), метод (MovieClip, класс), 375, 384, 701
createTextField(), метод (MovieClip, класс), 27, 703
createTextNode(), метод (XML, класс), 990
_currentframe, свойство (MovieClip, класс), 679, 704
curveTo(), метод (MovieClip, класс), 705

D

data, свойство (SharedObject, объект), 823
Date(), глобальная функция, 554
Date, класс, 554
    методы
        getDate(), 560
        getDay(), 561
        getFullYear(), 561
        getHours(), 561
        getMilliseconds(), 561
        getMinutes(), 562
        getMonth(), 562
        getSeconds(), 562
        getTime(), 562
        getTimezoneOffset(), 563
        getUTCDate(), 563
        getUTCDay(), 563
        getUTCFullYear(), 564
        getUTCHours(), 564
        getUTCMilliseconds(), 564
        getUTCMinutes(), 564
        getUTCMonth(), 564
        getUTCSeconds(), 565
        getYear(), 565
        setDate(), 565

setFullYear(), 565
setHours(), 566
setMilliseconds(), 566
setMinutes(), 566
setMonth(), 567
setSeconds(), 567
setTime(), 567
setUTCDate(), 568
setUTCFullYear(), 568
setUTCHours(), 569
setUTCMilliseconds(), 569
setUTCMinutes(), 569
setUTCMonth(), 570
setUTCSeconds(), 570
setYear(), 570
toString(), 571
UTC(), 571
valueOf(), 572
объекты, создание, 558
строка, читаемая человеком, 1057
decode(), метод (LoadVars, класс), 634
default, ключевое слово в инструкции switch, 203
delete, оператор, 179, 572
именованные элементы массива, удаление, 300
элементы массива, удаление, 306
DELETEKEY, свойство (Key, объект), 610
Director, связь с, 578, 589
docTypeDecl, свойство (XML, класс), 990
domain(), метод (LocalConnection, класс), 651
do-while, цикл, 189, 211
DOWN, свойство (Key, объект), 610
Drawing API, 413, 414
_droptarget, свойство (MovieClip, класс), 679, 707
duplicateMovieClip()
глобальная функция, 573
метод (MovieClip, класс), 375, 708
target, параметр и, 413
добавление созданных клипов в стек, 392
порождение клипов, 379
создание экземпляров, 381
duration, свойство (Sound, класс), 834

E

E, свойство (Math, объект), 662
ECMA-262, стандарт
void, оператор и, 178
потенциально зарезервированные слова, 446
различия между ActionScript и JavaScript, 1055
расхождения с ActionScript, 1055
ресурсы, 1032
по объектно-ориентированному программированию, 372
требования к регистру букв, 448
else, инструкция, 199
else if, инструкция, 201
Embed Fonts, параметр для текстовых полей, 910
<EMBED>, тег (HTML), 1083
embedFonts, свойство (TextField, класс), 897
empty, инструкция, 189
enabled, свойство
Button, класс, 511
MovieClip, класс, 709
END, свойство (Key, объект), 610
endFill(), метод (MovieClip, класс), 710
#endinitclip, прагма, 574
Enter, клавиша, отключение параметром Single Line, 913
перехват, 620
ENTER, свойство (Key, объект), 610
enterFrame, обработчик события клипа
создание переносимого предварительного загрузчика, 712
создание циклов с помощью, 220, 378
эмуляция динамической обработки событий, 286
eq, оператор равенства (Flash 4), 1051
Escape, клавиша, перехват, 619
ESCAPE, свойство (Key, объект), 610
escape(), глобальная функция, 574
escape-последовательности
Unicode, 127
в ActionScript, 124
в строковых литералах, 124
шестнадцатеричные, 125, 574
eval(), глобальная функция, 144, 575, 1056
event, протокол (getURL()), 591
exec, команда (автономный проигрыватель), 578
exp(), метод (Math, объект), 662
Expert Mode, режим (Actions, панель), 41, 452
многострочные комментарии и, 445

F

FComboBox, компонент FUI, 1072
FDataGrid, компонент FUI, 1071
FIFO, очереди, 303

file, протокол (getURL()), 591
firstChild, свойство (XML, класс), 991
Flash
веб-сайты, 29
нераспознаваемые теги и атрибуты HTML, 1066
программирование, введение в, 37
средства для создания профессиональных компонентов, 467
фильмы
(*см. также* клипы), 27
встраивание в веб-страницы, 1081, 1086
интерактивность, 27
программы, управляемые событиями, 263
управление временной диаграммой, 27
формы, создание, 469
экземпляры, 55
Flash 4, в сравнении с Flash 5 (и последующими)
зарезервированные слова, 446
локальные переменные в подпрограммах, 1051
операторы конкатенации строк, 1052
операторы равенства/неравенства, 1052
преобразования типов данных, 1051
присваивание значений динамически именуемым переменным, 191
производительность извлечения подстрок, 140
синтаксис обращения к переменным, 1050
эквивалентные строковые операции, 171
Flash 5
on() и onClipEvent(), обработчики событий, 272
выполняемые проекторы, ошибки, 629
ошибки реализации (*см.* ошибки реализации, Flash 5), 874
Flash 6
Drawing API, 413, 414
клипы в качестве кнопок, 414
лучшие приемы программирования, 1043
недокументированные функции, 24
новые характеристики, 17
обновленные сборки
Build 29, 1046
Build 47, 1045
Build 61, 1044
ошибки реализации (*см.* ошибки реализации, Flash 6), 544
присваивание TextField.restrict пустой строки, 926
упаковка графики компонента, 435
экземпляры клипов, получающие фокус ввода, 417
экземпляры новых клипов, 379, 384
Flash Kit, ресурсы (Internet.com), 1030
Flash Remoting MX, 1032
Flash UI (FUI), компоненты, 1071
FCheckBox, 1072
FDataGrid, 1071
FPushButton, 1074
FRadioButton, 1075
FRadioButtonGroup, 1075
FScrollBar, 1076
FStyleFormat, 1078
Flashcoders, почтовый список, 1030
Flash-программы, 48
Flash-фильмы
добавление кода в, 39
логика в, 50
программы, управляемые событиями, 56
создание, 48
создание интерфейса, 60
структура слоев и, 58, 390
FListBox (FUI компонент), 1072
floor(), метод (Math, объект), 121, 663
flush(), метод (SharedObject, объект), 824
focusEnabled, свойство (MovieClip, класс), 710
_focusrect, глобальное свойство, 577
Button, класс, 512
MovieClip, класс, 711
font, свойство (TextFormat, класс), 957
<FONT>, тег (HTML), 1060
for, цикл, 189, 212
оператор запятая в, 178
for-in, цикл, 189, 213
встроенные объекты и, 214
доступ к именованным элементам массива, 299
доступ к клипам, 406
FPushButton, компонент FUI, 1074
FRadioButton, компонент FUI, 1075
FRadioButtonGroup, компонент FUI, 1075
_framesloaded, свойство (MovieClip, класс), 680, 712
добавление сценариев в кадры, 454
использование вместо инструкции ifFrameLoaded, 195

fromCharCode(), метод (String, класс), 126, 127, 143, 869
fscommand(), глобальная функция, 577
FScrollBar, компонент FUI, 1076
FStyleFormat, компонент FUI, 1078
ftp, протокол (getURL()), 591
fullscreen, команда (автономный проигрыватель), 578
function, инструкция, 53, 189, 192
Function, класс, 581
    методы
        apply(), 582
        call(), 585
        toString(), 586
    свойства
        prototype, 585
Function(), конструктор, отсутствие поддержки, 234

G

ge, оператор больше или равно (Flash 4), 1051
GET, метод запроса HTTP, 1069
getAscii(), метод (Key, объект), 613
getBeginIndex(), метод (Selection, объект), 808
getBounds(), метод (MovieClip, класс), 714
getBytesLoaded(), метод, 712
    LoadVars, класс, 634
    MovieClip, класс, 715
    Sound, класс, 835
    XML, класс, 992
getBytesTotal(), метод
    LoadVars, класс, 636
    MovieClip, класс, 717
    Sound, класс, 837
    XML, класс, 994
getCaretIndex() (Selection, объект), 809
getCode(), метод (Key, объект), 614
getDate(), метод (Date, класс), 560
getDay(), метод (Date, класс), 561
getDepth(), метод
    Button, класс, 513
    MovieClip, класс, 718
    TextField, класс, 900
getEndIndex(), метод (Selection, объект), 810
getFocus(), метод (Selection, объект), 810
getFontList(), метод (TextField, класс), 900
getFullYear(), метод (Date, класс), 561
getHours(), метод (Date, класс), 561
getLocal(), метод (SharedObject, объект), 826
getMilliseconds(), метод (Date, класс), 561
getMinutes(), метод (Date, класс), 562
getMonth(), метод (Date, класс), 562
getNewTextFormat(), метод (TextField, класс), 902
getPan(), метод (Sound, класс), 837
getProperty(), глобальная функция, 587
getRGB(), метод (Color, класс), 549
getSeconds(), метод (Date, класс), 562
getSize(), метод (SharedObject, объект), 828
getTextExtent(), метод (TextFormat, класс), 959
getTextFormat(), метод (TextField, класс), 903
getTime(), метод (Date, класс), 562
getTimer(), глобальная функция, 588
getTimezoneOffset(), метод (Date, класс), 563
getTransform(), метод
    Color, класс, 550
    Sound, класс, 838
getURL()
    MovieClip, класс, 719
    глобальная функция, 589
        передача данных формы веб-серверу, 470
getUTCDate(), метод (Date, класс), 563
getUTCDay(), метод (Date, класс), 563
getUTCFullYear(), метод (Date, класс), 564
getUTCHours(), метод (Date, класс), 564
getUTCMilliseconds(), метод (Date, класс), 564
getUTCMinutes(), метод (Date, класс), 564
getUTCMonth(), метод (Date, класс), 564
getUTCSeconds(), метод (Date, класс), 565
getVersion(), глобальная функция, 593
getVolume(), метод (Sound, класс), 838
getYear(), метод (Date, класс), 565
_global, объект, 593
globalToLocal(), метод (MovieClip, класс), 719
GMT (Greenwich Mean Time)
    Date(), глобальная функция и, 554
    getTimezoneOffset() и, 563
    toString() и, 571
    в сравнении с UTC, 558
gotoAndPlay()
    MovieClip, класс, 720
    глобальная функция, 595
        не работает в событиях клипов, 597
gotoAndStop()
    MovieClip, класс, 721

глобальная функция, 597
добавление меток кадров, 64
gt, оператор больше (Flash 4), 1051

H

hasAccessibility, свойство (Capabilities, объект), 533
hasAudio, свойство (Capabilities, объект), 534
hasAudioEncoder, свойство (Capabilities, объект), 534
hasChildNodes(), метод (XML, класс), 995
hasMP3, свойство (Capabilities, объект), 534
hasOwnProperty(), метод (Object, класс), 783
hasVideoEncoder, свойство (Capabilities, объект), 535
_height, свойство
Button, класс, 513
MovieClip, класс, 680, 721
TextField, класс, 904
height, свойство (Stage, объект), 856
hide(), метод (Mouse, объект), 673
_highquality, глобальное свойство, 599
hitArea, свойство (MovieClip, класс), 722
hitTest(), метод (MovieClip, класс), 724
HOME, свойство (Key, объект), 610
hscroll, свойство (TextField, класс), 905
HTML-атрибуты, изменяемые на этапе исполнения, 1086
HTML, поддержка, 1059, 1068
гипертекстовые ссылки в текстовых полях, 1062
вызов функций ActionScript из, 1068
выполнение JavaScript из, 1068
заключение в кавычки значений атрибутов, 1065
курсивный текст в текстовых полях, 1060
перенос строк в текстовых полях, 1062
показ текста HTML на экране, 1066
теги якоря в текстовых полях, 1062
не добавляются в обход при табуляции, 1063
управление шрифтами в текстовых полях, 1060
HTML, создание форм, 472
HTML-теги
<A>, 1062
<B>, 1059

, 1062
<FONT>, 1060
<I>, 1060
<LI>, 1060
<P>, 1061
<TAB>, 1064
<TEXTFORMAT>, 1063
<U>, 1060
html, свойство (TextField, класс), 906
htmlText, свойство (TextField, класс), 907
http, протокол (getURL()), 591
HTTP-запрос, методы, 1069

I

<I>, тег (HTML), 1060
id3, свойство (Sound, класс), 839
if, инструкция, 198
if, ключевое слово в сценариях расчета, 71
if-else, инструкция, 189
ifFrameLoaded, инструкция, 189
ignoreWhite, свойство (XML, класс), 995
#include, директива, 458, 599
indent, свойство (TextFormat, класс), 960
indexOf(), метод (String, класс), 134, 870
substring() и, 140
–Infinity, глобальное свойство, 602
Infinity, глобальное свойство, 120, 602
isFinite() и, 608
initObject, параметр (attachMovie()), 383
#initclip, прагма, 603
input, свойство (Capabilities, объект), 535
Insert Target Path, инструмент, 403
INSERT, свойство (Key, объект), 610
insertBefore(), метод (XML, класс), 996
instanceof, оператор, 181, 607
int(), глобальная функция, 607
Math.floor(), метод и, 663
isActive(), метод (Accessibility, объект), 489
isDebugger, свойство (Capabilities, объект), 536
isDown(), метод (Key, объект), 615
isFinite(), глобальная функция, 608
isNaN(), глобальная функция, 119, 609
ISO 8859-1 (Latin 1), символы, 122
эквиваленты в Unicode, 1036, 1040
ISO-8859, наборы символов, 122
isToggled(), метод (Key, объект), 616
italic, свойство (TextFormat, класс), 961

J

Java и ActionScript, эквивалентные методы и свойства, 338

JavaScript
как работает eval(), 144
различия между ActionScript и ECMA-262, 1055
связь из ActionScript, 1063, 1067
fscommand(), 577
чувствительность к регистру и ActionScript, 448
javascript, протокол (getURL()), 591
join(), метод (Array, класс), 312, 496

K

Key, объект, 609
в сравнении с событием кнопки keyPress, 514
коды специальных клавиш, 1040
методы
addListener(), 612
getAscii(), 613
getCode(), 614
isDown(), 615
isToggled(), 616
removeListener(), 622
свойства кодов клавиш, 610
события приемника
onKeyDown(), 617
onKeyUp(), 620
keyPress, событие кнопки, 514

L

language, свойство (Capabilities, объект), 536
lastChild, свойство (XML, класс), 996
lastIndexOf(), функция (String, класс), 135, 871
Latin 1, набор символов, 122
коды клавиш и кодовые позиции, 611
кодовые позиции, получение, 144
создание строк из кодовых позиций, 143
сортировка массивов, 310
leading, свойство (TextFormat, класс), 962
LEFT, свойство (Key, объект), 610
leftMargin, свойство (TextFormat, класс), 963
length, свойство
arguments, объект, 493
Array, класс, 297, 496
расширение массивов без присвоения значений элементам, 301
удаление элементов массивов, 306
String, класс, 133, 871
TextField, класс, 910
_leveln, глобальное свойство, 91, 622
ссылки на документы в Player, 402
стеки документов, 393, 402
<LI>, тег (HTML), 1060
LIFO (последним пришел – первым ушел), стеки, 303
lineStyle(), метод (MovieClip, класс), 726
lineTo(), метод (MovieClip, класс), 727
Lingo, связь с, 578
lingo, протокол (getURL()), 591
ll, свойство (transformObject), 849
LN10, свойство (Math, объект), 663
LN2, свойство (Math, объект), 663
load(), метод
LoadVars, класс, 636
XML, класс, 997
loaded, свойство
LoadVars, класс, 638
XML, класс, 998
loadMovie(), метод
MovieClip, класс, 729
глобальная функция, 623
target, параметр и, 413
использование обработчиков событий данных, 735
загрузка внешних файлов SWF в проигрыватель, 387
использование с attachMovie(), 389
loadMovieNum(), глобальная функция, 626
loadSound(), метод (Sound, класс), 840
loadVariables(), глобальная функция, 627
target, параметр и, 413
использование обработчиков событий данных, 734
ошибка реализации, 629
loadVariables(), метод (MovieClip, класс), 729
loadVariablesNum(), глобальная функция, 629, 630
LoadVars, класс, 630
методы
decode(), 634
getBytesLoaded(), 634
getBytesTotal(), 636
load(), 636
send(), 642
sendAndLoad(), 643
toString(), 645
обработчики событий
onData(), 639
onLoad(), 640
свойства
contentType, 633
loaded, 638

LocalConnection, класс
методы
close(), 650
connect(), 650
domain(), 651
send(), 654
обработчики событий
allowDomain(), 647
onStatus(), 652
localToGlobal(), метод (MovieClip, класс), 730
log(), метод (Math, объект), 664
LOG10E, свойство (Math, объект), 664
LOG2E, свойство (Math, объект), 664
lr, свойство (transformObject), 849
lt, оператор меньше (Flash 4), 1051

M

Macromedia
Director, 578, 589, 1030
Flash (*см.* Flash), 1030
биржа Flash, 1030
Macromedia Flash Communication Server MX (Comm Server), 14, 40
manufacturer, свойство (Capabilities, объект), 537
Math, объект, 120
математические
встроенные функции, 121, 656
константы, 120, 656
методы
abs(), 657
acos(), 658
asin(), 658
atan(), 659
atan2(), 659
ceil(), 661
cos(), 661
exp(), 662
floor(), 121, 663
log(), 664
max(), 665
min(), 665
pow(), 118, 666
random(), 121, 667
round(), 668
sin(), 669
sqrt(), 670
tan(), 670
свойства
E, 662
LN10, 663
LN2, 663
LOG10E, 664
LOG2E, 664
PI, 666
SQRT1_2, 670
SQRT2, 670
maxChars, свойство (TextField, класс), 911
maxhscroll, свойство (TextField, класс), 911
maxscroll, свойство (TextField, класс), 912
maxscroll, свойство (Flash 4), 671
MAX_VALUE, свойство (Number, класс), 120, 774
MIME, тип содержимого, задание, 471
min(), метод (Math, объект), 665
Ming, библиотека C с открытым исходным кодом, 1033
MIN_VALUE, свойство (Number, класс), 120, 775
Mouse, объект, 671
методы
addListener(), 672
hide(), 673
removeListener(), 678
show(), 677
события приемника
onMouseDown(), 675
onMouseMove(), 676
onMouseUp(), 677
MoveClip.onData(), обработчик события ошибка реализации, 736
moveTo(), метод (MovieClip, класс), 731
Movie Explorer, 457
MovieClip, класс, 27, 678
компоненты, 440
методы
attachMovie(), 375, 384, 687
добавление созданных клипов в стек, 391
создание экземпляров, 382
beginFill(), 413, 689
beginGradientFill(), 413, 694
clear(), 413, 700
createEmptyMovieClip(), 375, 384, 701
createTextField(), 27, 703
curveTo(), 413, 705
duplicateMovieClip(), 375, 708
добавление созданных клипов в стек, 392
порождающие клипы, 379
создание экземпляров, 381
endFill(), 413, 710
getBounds(), 714

getBytesLoaded(), 715
getBytesTotal(), 717
getDepth(), 718
getURL(), 719
globalToLocal(), 719
gotoAndPlay(), 720
gotoAndStop(), 721
hitTest(), 724
lineStyle(), 413, 726
lineTo(), 413, 727
loadMovie(), 729
loadVariables(), 729
localToGlobal(), 730
moveTo(), 413, 731
nextFrame(), 733
play(), 751
prevFrame(), 752
removeMovieClip(), 752
setFocus(), 417
setMask(), 753
startDrag(), 755
stop(), 756
stopDrag(), 756
swapDepths(), 756
unloadMovie(), 762
valueOf(), 764
обеспечение независимости подклассов, 430
обработчики событий
onData(), 734
onDragOut(), 736
onDragOver(), 737
onKeyDown(), 737, 738
onKeyUp(), 740
onKillFocus(), 741
onLoad(), 742
onMouseDown(), 745
onMouseMove(), 745
onMouseUp(), 746
onPress(), 747
onRelease(), 747
onReleaseOutside(), 747
onRollOut(), 748
onRollOver(), 748
onSetFocus(), 748
onUnload(), 749
подклассы, 440
свойства
_alpha, 679, 686
_currentframe, 679, 704
_droptarget, 679, 707
enabled, 709
focusEnabled, 710
_focusrect, 711
_framesloaded, 680, 712
добавление сценариев в кадры, 454
вместо инструкции ifFrameLoaded, 195
_height, 680, 721
hitArea, 722
_name, 680, 733
определение имен клипов, 407
хранение имен экземпляров, 385
_parent, 680, 751
доступ к переменным различных временных диаграмм, 89
недоступно на главной временной диаграмме, 400
ссылка на текущий клип, 375, 399
_rotation, 680, 752
tabChildren, 758
tabEnabled, 758
tabIndex, 760
_target, 407, 680, 761
_totalframes, 680, 761
вместо инструкции ifFrameLoaded, 195
trackAsMenu, 762
_url, 680, 762
useHandCursor, 763
_visible, 680, 764
_width, 680
_x, 680, 765
_xmouse, 680, 767
_xscale, 680, 768
_y, 680, 768
_ymouse, 681, 769
_yscale, 681, 770
создание компонентов, 437
movieclip, тип данных, 375
multiline, свойство (TextField, класс), 913

N

_name, свойство
Button, класс, 515
MovieClip, класс, 680, 733
определение имен клипов, 407
хранение имен экземпляров, 385
TextField, класс, 914
NaN, нечисло
isNan(), глобальная функция и, 609
Number, класс, 775
глобальное свойство, 104, 119, 770
ne, оператор «не равно» (Flash 4), 1051

NEGATIVE_INFINITY, свойство (Number, класс), 776
(*см. также* –Infinity, глобальное свойство), 776
Netscape, руководство по базовому JavaScript, 1032
new MovieClip(), команда (недопустимая), 379
new, оператор, 179, 771
newline, константа, 771
nextFrame()
глобальная функция, 772
метод (MovieClip, класс), 733
nextScene(), глобальная функция, 772
nextSibling, свойство (XML, класс), 998
nodeName, свойство (XML, класс), 999
nodeType, свойство (XML, класс), 1000
nodeValue, свойство (XML, класс), 1001
Normal Mode (Actions, панель), 39, 451
переключение из Expert Mode, 452
not, оператор логического НЕ (Flash 4), 446
null, тип данных, 101, 148
правила равенства для, 163
Num Lock, клавиша и isToggled(), 617
Number(), глобальная функция, 108, 772
Number, класс, 773
методы
toString(), 774
свойства
MAX_VALUE, 120, 773
MIN_VALUE, 120, 773
NaN, 774
NEGATIVE_INFINITY, 774
(*см. также* –Infinity, глобальное свойство), 774
POSITIVE_INFINITY, 774
number, тип данных, 116, 122
Infinity/–Infinity, глобальные свойства, 120
NaN, нечисло, 119
использование операторов, 121
литералы чисел с плавающей точкой, 117
максимальное/минимальное значения, 120, 774
правила равенства для, 162
преобразование в, 105, 157
преобразование строк в, 108, 117
целочисленные литералы, 117
Number.toString(), метод, 776

O

Object, класс, 777
методы
addProperty(), 778
hasOwnProperty(), 783
registerClass(), метод класса, 785
toString(), 790, 1058
unwatch(), 791
valueOf(), 791
watch(), 792
свойства
constructor, 783
__proto__, 351, 785
Object(), конструктор, 324
<OBJECT>, тег (HTML), 1082
on(), обработчик события, 272
область видимости, 278
onChanged(), обработчик события и событие приемника (TextField, класс), 914
onClipEvent(), обработчик события, 272
область видимости, 278
события клипов для, 684
onClose(), обработчик события (XMLSocket, класс), 1022
onConnect(), обработчик события (XMLSocket, класс), 1022
onData(), обработчик события
LoadVars, класс, 639
MovieClip, класс, 734
XML, класс, 1002
XMLSocket, класс, 1023
onDragOut(), обработчик события
Button, класс, 516
MovieClip, класс, 736
onDragOver(), обработчик события
Button, класс, 516
MovieClip, класс, 737
onKeyDown(), обработчик события (MovieClip, класс), 738
onKeyDown(), событие приемника (Key, объект), 617
onKeyUp(), обработчик события (MovieClip, класс), 740
onKeyUp(), событие приемника (Key, объект), 620
onKillFocus(), обработчик события
Button, класс, 517
MovieClip, класс, 741
TextField, класс, 916
onLoad(), обработчик события
LoadVars, класс, 640
MovieClip, класс, 742
Sound, класс, 842

XML, класс, 1003
onMouseDown(), обработчик события (MovieClip, класс), 745
onMouseDown(), событие приемника (Mouse, объект), 675
onMouseMove(), обработчик события (MovieClip, класс), 745
onMouseMove(), событие приемника (Mouse, объект), 676
onMouseUp(), обработчик события (MovieClip, класс), 746
onMouseUp(), событие приемника (Mouse, объект), 677
onPress(), обработчик события
Button, класс, 518
MovieClip, класс, 747
onRelease(), обработчик события
Button, класс, 519
MovieClip, класс, 747
onReleaseOutside(), обработчик события
Button, класс, 519
MovieClip, класс, 747
onResize(), событие приемника (Stage, объект), 857
onRollOut(), обработчик события
Button, класс, 520
MovieClip, класс, 748
onRollOver(), обработчик события
Button, класс, 521
MovieClip, класс, 748
onScroller(), обработчик события и события приемника (TextField, класс), 917
onSetFocus(), обработчик события
Button, класс, 521
MovieClip, класс, 748
TextField, класс, 919
onSetFocus(), событие приемника (Selection, объект), 811
onSoundComplete(), обработчик события (Sound, класс), 843
onStatus(), обработчик события
LocalConnection, класс, 652
SharedObject, объект, 829
onUnload(), обработчик события (MovieClip, класс), 749
onXML(), обработчик события (XMLSocket, класс), 1024
OpenSWF, информационный центр, 1033
Options, меню, 41
or, оператор логического ИЛИ (Flash 4), 446
os, свойство (Capabilities, объект), 538
Output, окно, 42
trace() и, 44, 51

P

<P>, тег (HTML), 1061
Page Down (*см.* PGDN), 610
Page Up (*см.* PGUP), 610
Parameters, окно (Actions, панель), 452
_parent, свойство
Button, класс, 522
MovieClipм класс, 680, 751
доступ к переменным в различных временных диаграммах, 89
недоступно на главной временной диаграмме, 400
ссылка на текущий клип, 375, 399
TextField, класс, 920
parentNode, свойство (XML, класс), 1005
parseFloat(), глобальная функция, 109, 795
parseInt(), глобальная функция, 109, 796
parseXML(), метод (XML, класс), 1005
password, свойство (TextField, класс), 921
Perl
библиотека для генерации фильмов Flash/SWF, 1034
интеграция с Flash, 472, 477
PGDN, свойство (Key, объект), 610
PGUP, свойство (Key, объект), 610
PI, свойство (Math, объект), 666
pixelAspectRatio, свойство (Capabilities, объект), 539
play(), метод (MovieClip, класс), 751
play(), глобальная функция, 798
Playerplay(), проигрыватель
автономный, 577, 581
версия, 975
загрузка внешних файлов SWF в, 387, 623, 729
обновление для (Build 41/42), 1046
получение количества загруженных байт, 715
стек уровней документа, 91, 393, 402, 622
удаление
клипов фильмов, 752, 762
прикрепленных и дублированных экземпляров, 410
pop(), метод (Array, класс), 307, 497
position, свойство (Sound, класс), 844
POSITIVE_INFINITY, свойство (Number, класс), 776
POST, метод запроса HTTP, 1069
pow(), метод (Math, объект), 118, 666
prevFrame(), глобальная функция, 799
prevFrame(), метод (MovieClip, класс), 752

previousSibling свойство (XML класс), 1007
prevScene(), глобальная функция, 799
print, протокол (getURL()), 591
print(), глобальная функция, 799
target, параметр и, 413
printAsBitmap(), глобальная функция, 801
target, параметр и, 413
printAsBitmapNum(), глобальная функция, 801
printNum(), глобальная функция, 802
__proto__, свойство (Object, класс), 351
prototype, свойство (Function, класс), 585
push(), метод (Array, класс), 301, 498

Q

_quality, глобальное свойство, 803
quit, команда (автономный проигрыватель), 578

R

random(), глобальная функция, 803
random(), метод (Math, объект)121, 667
Red, Green, Blue (*см.* RGB-тройки), 546
Red, Green, Blue и Alpha (*см.* RGBA-четверки), 546
registerClass(), метод класса (Object, класс), 785
removeListener(), метод
Key, объект, 622
Mouse, объект, 678
Selection, объект, 812
Stage, объект, 858
TextField, класс, 921
removeMovieClip()
глобальная функция, 804
target, параметр и, 413
метод (MovieClip, класс), 752
удаление прикрепленных и дублированных экземпляров, 410
removeNode(), метод (XML, класс), 1008
removeTextField(), метод (TextField, класс), 922
replaceSel(), метод (TextField, класс), 923
Reset, кнопка на формах, 479
restrict, свойство (TextField, класс), 924
return, инструкция, 189, 193
возврат значений из функций, 232
завершение функций, 231
Return, клавиша, перехват, 620
reverse(), метод (Array, класс), 309, 498
setTransform(), метод, 553
RGB-тройки, 545, 689
getRGB(), метод, 549
setRGB(), метод, 551
RGBA-четверки, 545
getTransform(), метод, 550
RIGHT, свойство (Key, объект), 610
rightMargin, свойство (TextFormat, класс), 964
rl, свойство (transformObject, объект), 849
_root, глобальное свойство, 805
доступ к переменным в различных временных диаграммах, 89
ссылка на текущий уровень главного фильма, 401
_rotation, свойство
Button, класс, 523
MovieClip, класс, 680, 752
TextField, класс, 926
round(), метод (Math, объект), 668
rr, свойство (transformObject, объект), 849

S

screenColor, свойство (Capabilities, объект), 539
screenDPI, свойство (Capabilities, объект), 540
screenResolutionX, свойство (Capabilities, объект), 540
screenResolutionY, свойство (Capabilities, объект), 541
Script, окно (Actions, панель), 39, 450
добавление инструкций в, 451
добавление кода в кнопки, 454
Scroll Lock, клавиша и isToggled(), 617
scroll, свойство
TextField, класс, 896, 927
глобальное, 805
текстовых полей
maxscroll, свойство и, 671
security, свойство (System, объект), 879
selectable, свойство (TextField, класс), 929
Selection, объект, 806
для клипов, 418
методы
addListener(), 808
getBeginIndex(), 808
getCaretIndex(), 809
getEndIndex(), 810
getFocus(), 810
removeListener(), 812
setFocus(), 812
setSelection(), 815

работа с выделением текста в поле, 889
события приемника
onSetFocus(), 811
send(), метод
LoadVars, класс, 642
LocalConnection, класс, 654
XML, класс, 1008
передача данных формы веб-серверу, 470
XMLSocket, класс, 1025
sendAndLoad(), метод
LoadVars, класс, 643
XML, класс, 1009
передача данных формы веб-серверу, 470
sendEvent(), метод (Accessibility, объект), 489
serverString, свойство (Capabilities, объект), 542
set, инструкция, 189
в сравнении с Set Variable во Flash 4, 191
setDate(), метод (Date, класс), 565
setFocus(), метод
MovieClip, класс, 417
Selection, объект, 812
setFullYear(), метод (Date, класс), 565
setHours(), метод (Date, класс), 566
setInterval(), функция
альтернатива циклам временной диаграммы, 223
глобальная, 816
эмуляция, 217, 1057
setMask(), метод (MovieClip, класс), 753
setMilliseconds(), метод (Date, класс), 566
setMinutes(), метод (Date, класс), 566
setMonth(), метод (Date, класс), 567
setNewTextFormat(), метод (TextField, класс), 929
setPan(), метод (Sound, класс), 847
setProperty(), глобальная функция, 820
setRGB(), метод (Color, класс), 551
setSeconds(), метод (Date, класс), 567
setSelection(), метод (Selection, объект), 815
setTextFormat(), метод (TextField, класс), 931
setTime(), метод (Date, класс), 567
setTimeout(), эмуляция, 217, 1057
setTransform(), метод
Color, класс, 553
Sound, класс, 848
setUTCDate(), метод (Date, класс), 568
setUTCFullYear(), метод (Date, класс), 568
setUTCHours(), метод (Date, класс), 569
setUTCMilliseconds(), метод (Date, класс), 569
setUTCMinutes(), метод (Date, класс), 569
setUTCMonth(), метод (Date, класс), 570
setUTCSeconds(), метод (Date, класс), 570
setVolume(), метод (Sound, класс), 849
setYear(), метод (Date, класс), 570
SharedObject, объект, 820
методы
flush(), 824
getLocal(), 826
getSize(), 828
обработчики событий
onStatus(), 829
свойства
data, 823
SHIFT, свойство (Key, объект), 610
shift(), метод (Array, класс), 308, 498
show(), метод (Mouse, объект), 677
showmenu, команда (автономный проигрыватель), 578
showMenu, свойство (Stage, объект), 860
showSettings(), метод (System, объект), 882
sin(), метод (Math, объект), 669
size, свойство (TextFormat, класс), 964
slice(), метод
Array, класс, 164, 499
String, класс, 138, 872
sort(), метод (Array, класс), 310, 500
sortOn(), метод (Array, класс), 501
Sound, класс, 830
методы
attachSound(), метод, 832
getBytesLoaded(), 835
getBytesTotal(), 837
getPan(), метод, 837
getTransform() метод, 838
getVolume() метод, 838
loadSound(), 840
setPan() метод, 847
setTransform() метод, 848
setVolume() метод, 849
start() метод, 850
stop() метод, 851
обработчики событий
onLoad(), 842
onSoundComplete(), 843
свойства
duration, 834
id3, 839
position, 844

_soundbuftime, глобальное свойство, 852
SPACE, свойство (Key, объект), 610
splice(), метод (Array, класс), 303, 309, 503
split(), метод (String, класс), 873
sqrt(), метод (Math, объект), 670
SQRT1_2, свойство (Math, объект), 670
SQRT2, свойство (Math, объект), 670
Stage, объект, 853
    методы
        addListener(), 854
        removeListener(), 858
    свойства
        align, 855
        height, 856
        showMenu, 860
        width, 862
    события приемника
        onResize(), 857
star, поле, генерация событием загрузки, 744
start(), метод (Sound, класс), 850
startDrag(), глобальная функция, 862
    target, параметр и, 413
startDrag(), метод (MovieClip, класс), 755
status, свойство (XML, класс), 1010
stop(), глобальная функция, 62, 863
stop(), метод
    (MovieClip, класс), 756
    (Sound, класс), 851
stopAllSounds(), глобальная функция, 864
stopDrag(), глобальная функция, 864
stopDrag(), метод (MovieClip класс), 756
#strict, прагма, 865, 1055
string, тип данных, 122
    правила равенства для, 162
    преобразование
        значений в, 865
        типов данных в, 106
String(), глобальная функция, 107, 865
String, класс, 866
    методы
        charAt(), 134, 867
        charCodeAt(), 144, 868
        concat(), 868
            в сравнении с оператором +=, 129
        fromCharCode(), метод класса, 126, 143, 869
        indexOf(), 134, 870
            substring() и, 140
        lastIndexOf(), 135, 871
        slice(), 138, 872
        split(), 873
        substr(), 137, 874
        substring(), 875
            indexOf() и, 140
        toLowerCase(), 141, 876
        toUpperCase(), 141, 877
    свойства
        length, 133, 871
substr(), метод (String, класс), 137, 874
substring(), метод (String, класс), 875
    indexOf() и, 140
super, «оператор», 182, 877
swapDepths(), метод (MovieClip, класс), 756
    назначение уровней экземплярам в программно генерируемых стеках клипов, 392
    обмен экземпляров в стеках внутренних слоев, 390
SWF-файлы
    _leveln, глобальное свойство, 393, 402, 622
    _url, свойство и, 762
    внешнее размещение кода ActionScript, 458
    добавление SWF-файлов пользовательских интерфейсов в компоненты, 466
    загрузка в Player, 387, 623, 729
    загрузка на уровни документов посредством loadMovie(), 393, 402
    обратная совместимость, 1042
    ресурсы по формату, 1033
    создание SWF-файлов пользовательских интерфейсов, 465
switch, инструкция, 202, 204
    case, ключевое слово и, 204
    эмуляция во Flash 5, 205
System, объект, 878
    методы
        security.allowDomain(), 878
        showSettings(), 882
    свойства
        capabilities, 878
        security, 879
        useCodepage, 882

T

Tab, перехват нажатий, 619
TAB, свойство (Key, объект), 610
<TAB>, тег (HTML), 1064
tabChildren, свойство (MovieClip, класс), 758
tabEnabled, свойство

Button, класс, 523
MovieClip, класс, 758
TextField, класс, 934
tabIndex, свойство
Button, класс, 524
MovieClip, класс, 760
TextField, класс, 935
tabStops, свойство (TextFormat, класс), 966
tan(), метод (Math, объект), 670
_target, свойство
Button, класс, 524
MovieClip, класс, 407, 680, 761
TextField, класс, 936
target, параметр и глобальные функции, 411
target, свойство (TextFormat, класс), 968
targetPath(), глобальная функция, 407
Tell Target, инструмент, 1053
tellTarget(), глобальная функция, 885, 1053
text, свойство (TextField, класс), 937
textColor, свойство (TextField, класс), 938
TextField, класс, 885
методы
addListener(), 890
getDepth(), 900
getFontList(), метод класса, 900
getNewTextFormat(), 902
getTextFormat(), 903
removeListener(), 921
removeTextField(), 922
replaceSel(), 923
setNewTextFormat(), 929
setTextFormat(), 931
обработчики событий
onChanged(), 914
onKillFocus(), 916
onScroller(), 917
onSetFocus(), 919
свойства
_alpha, 891
autoSize, 892
background, 894
backgroundColor, 894
border, 895
borderColor, 895
bottomScroll, 896
condenseWhite, 897
embedFonts, 897
hscroll, 905
_height, 904
html, 906
htmlText, 907
length, 910
maxChars, 911
maxhscroll, 911
maxscroll, 912
multiline, 913
_name, 914
_parent, 920
password, 921
restrict, 924
_rotation, 926
scroll, 927
selectable, 929
tabEnabled, 934
tabIndex, 935
_target, 936
text, 937
textColor, 938
textHeight, 938
textWidth, 940
type, 941
_url, 942
variable, 942
_visible, 943
_width, 943
wordWrap, 944
_x, 945
_xmouse, 946
_xscale, 946
_y, 947
_ymouse, 948
_yscale, 948
TextFormat, класс, 949
методы
getTextExtent(), 959
свойства
align, 953
blockIndent, 954
bold, 955
bullet, 956
color, 957
font, 957
indent, 960
italic, 961
leading, 962
leftMargin, 963
rightMargin, 964
size, 964
tabStops, 966
target, 968
underline, 969
url, 970
<TEXTFORMAT>, тег (HTML), 1063
textHeight, свойство (TextField, класс), 938

textWidth, свойство (TextField, класс), 940
this, ключевое слово
значения, 281
в объектах, 276
передача неименованных клипов в функции в виде ссылок, 277
явная ссылка на текущий экземпляр, 397
toggleHighQuality(), глобальная функция, 971
toLowerCase(), метод (String, класс), 141, 876
Toolbox, окно (Actions, панель), 39, 450
toString(), метод, 107
Array, класс, 313, 504
Boolean, класс, 506
Date, класс, 571
Function, класс, 586
LoadVars, класс, 645
Number, класс, 776
Object, класс, 790, 1058
XML, класс, 1012
_totalframes, свойство (MovieClip, класс), 680, 761
вместо инструкции ifFrameLoaded, 195
toUpperCase(), метод (String, класс), 141, 877
trace(), глобальная функция, 43, 971
в циклах, 207
интерпретатор ActionScript и, 44
как обрабатывается интерпретатором, 44
trackAsMenu, свойство
Button, класс, 525
MovieClip, класс, 762
transformObject, свойства, 849
trapallkeys, команда (автономный проигрыватель), 579, 619
type, свойство (TextField, класс), 941
typeof, оператор, 46, 83, 179, 972
возвращаемые значения, 110

U

<U>, тег (HTML), 1060
Ultrashock.com, 1030
undefined, тип данных, 101, 147
правила равенства для, 163
underline, свойство (TextFormat, класс), 969
Unicode
escape-последовательности, 125
кодировки, 1040
поддержка в ActionScript, 122
unloadMovie(), 409
глобальная функция, 973
target, параметр и, 413
метод (MovieClip, класс), 762
удаление содержимого экземпляров клипов, 409
удаление уровней в стеках документов, 408
unloadMovieNum(), глобальная функция, 974
unshift(), метод (Array, класс), 302, 504
unwatch(), метод (Object, класс), 791
UP, свойство (Key, объект), 610
updateAfterEvent(), глобальная функция, 285, 974
эмуляция пользовательских курсоров мыши, 288
_url, свойство
Button, класс, 526
MovieClip, класс, 680, 762
TextField, класс, 942
url, свойство (TextFormat, класс), 970
useCodepage, свойство (System, объект), 882
useHandCursor, свойство
Button, класс, 526
MovieClip, класс, 763
UTC (Coordinated Universal Time), универсальное время, 560
getTimezoneOffset() и, 563
UTC(), метод (Date, класс), 571

V

valueOf(), метод
Boolean, класс, 507
Date, класс, 572
MovieClip, класс, 764
Object, класс, 791
var, инструкция, 189
внутри и вне функций, 190
var, ключевое слово, 43
локальные переменные и, 93
variable, свойство (TextFieldм класс), 942
vbscript, протокол (getURL()), 591
$version, «глобальное» свойство, 975
version, свойство (Capabilities, объект), 542
_visible, свойство
Button, класс, 527
MovieClip, класс, 680, 764
TextField, класс, 943
void, оператор, 178

W

watch(), метод (Object, класс), 792
while, цикл, 52, 189, 206
_width, свойство
  Button, класс, 527
  MovieClip, класс, 680, 765
  TextField, класс, 943
width, свойство (Stage, объект), 862
Windows, виртуальные коды клавиш, 611
with, инструкция, 95, 189, 193
wordWrap, свойство (TextField, класс), 944

X

_x, свойство
  Button, класс, 528
  MovieClip, класс, 680, 765
  TextField, класс, 945
xch, (обмен) экземпляры, 464
XML, класс, 976, 984
  иерархия объектов, 977
  методы
    appendChild(), 984
    cloneNode(), 987
    createElement(), 989
    createTextNode(), 990
    getBytesLoaded(), 992
    getBytesTotal(), 994
    hasChildNodes(), 995
    insertBefore(), 996
    load(), 997
    parseXML(), 1005
    removeNode(), 1008
    send(), 1008
      передача данных формы веб-серверу, 470
    sendAndLoad(), 1009
      передача данных формы веб-серверу, 470
    toString(), 1012
  обработчики событий
    onData(), 1002
    onLoad(), 1003
  прикрепление обработчиков событий к, 286
  свойства
    attributes, 985
    childNodes, 986
    contentType, 988
    docTypeDecl, 990
    firstChild, 991
    ignoreWhite, 995
    lastChild, 996
    loaded, 998
    nextSibling, 998
    nodeName, 999
    nodeType, 1000
    nodeValue, 1001
    parentNode, 1005
    previousSibling, 1007
    status, 1010
    xmlDecl, 1012
xmlDecl, свойство (XML, класс), 1012
XMLnode, класс, 1013
XMLSocket, класс, 1014
  методы
    close(), 1020
    connect(), 1020
    send(), 1025
  обработчики событий
    onClose(), 1022
    onConnect(), 1022
    onData(), 1023
    onXML(), 1024
  прикрепление обработчиков событий к, 286
_xmouse, свойство
  Button, класс, 529
  MovieClip, класс, 680, 767
  TextField, класс, 946
_xscale, свойство
  Button, класс, 529
  TextField, класс, 946, 948
  MovieClip, класс, 680, 768

Y

_y, свойство
  Button, класс, 529
  TextField, класс, 947
  MovieClip, класс, 680, 768
_ymouse, свойство
  Button, класс, 530
  TextField, класс, 948
  MovieClip, класс, 681, 769
_yscale, свойство
  Button, класс, 530
  MovieClip, класс, 681, 770
  TextField, класс, 948

А

абсолютные пути к клипам, 407, 884
абсолютные ссылки, 89, 401
автоматическая типизация, 82, 103
автоматическое преобразование
  значений переменных, 82
  типа данных, 104

автоматическое создание переменных, 75
автономный проигрыватель, 26
fscommand() и, 577, 581
поддержка GET и POST, 1070
автопилот ActionScript, отключение, 41
анимация, синхронизированная, 221
аргументы, 45
в сравнении с параметрами, 228
функции в качестве, 253
передача в функцию, 229
разделители, 45
с элементарными и составными данными, 245
арифметические операторы, 157
арккосинус, вычисление, 658
арксинус, вычисление, 658
арктангенс, вычисление, 659
асинхронное исполнение кода, 56
ассоциативность операторов, 155
атрибуты, значения в кавычках, 1065

Б

беззнакового сдвига вправо с присваиванием оператор (>>>=), 154
бесконечная рекурсия, предотвращение, 247
бесконечные циклы, 207, 210
предотвращение, 216
библиотека, создание
новых экземпляров из, 382, 687
экземпляров клипов вручную, 380
библиотеки совместного доступа и импорт на этапе исполнения, 458
бинарные операторы, 152
биржа Flash, 1030
блоки инструкций, 187
создание функций, 225
условные инструкции и, 205
больше (>), оператор, 131, 168
больше (gt), оператор (Flash 4), 1051
больше или равно (>=), оператор, 131, 169
больше или равно (ge), оператор (Flash 4), 1051
броузеры (*см.* веб-броузеры), 580
буксировка клипов, завершение, 756
булевы значения, 131
булевы логические операторы (*см.* логические операторы), 172

В

веб-броузеры, полноэкранный режим в, 580
веб-сайты, использующие ActionScript, 29
венгерская нотация, 77
ветвление программы и логические операторы, 171
вложенные
клипы, доступ к, 1049
комментарии, 445
массивы, 293
concat(), метод и, 305
преобразование в строки, 313
создание, 315
функции, 248
циклы, 210
экземпляры клипов, ссылка на, 398
внешние переменные, 97
внешние файлы ActionScript, импортирование текста из, 458
внешние фильмы, импортирование, 387
внутренние функции (*см. также* встроенные функции), 252
возврат каретки в строках, 124
вопросник с вариантами ответов
в стиле ООП, 369, 372
инициализация вопросника, 62
массивы и хранение ответов, 316
программирование кнопок ответов, 64
код кнопки вопроса 1, 65
создание концовки, 69, 72
создание, 58, 60
тестирование, 73
централизация кода, 256
временные диаграммы
альтернативы циклов, 223
воспроизведение с заданного кадра, 596, 720
добавление меток кадров, 64
остановка в заданном кадре, 721
переменные на разных временных диаграммах, 86
доступ к, 88
поиск скрытых слоев, 456
создание переменных, 84
ссылки, абсолютные/относительные, 89
структура слоев и, 59, 390
временные переменные, 238
временные преобразования типа данных, 110

время, определение истекшего, 588
всплывающее меню (*см.* Options, меню), 41
встроенные
классы, 359, 360, 485
объекты
for-in, циклы и, 214
функции, 54, 251, 484
математические, 121, 656
шрифты, ошибка в реализации, 806
вызов функций, 190, 133, 225, 326
выгрузка клипа, обработчик события, 750
выражения, 186
операторы и, 151
регулярные (не поддерживаются), 136, 1056
символьные, 49, 102
сложные, 102
автоматическая типизация, 103
создание данных с помощью, 102
высота клипов/фильмов
в пикселах, 721
в процентном отношении, 770
вычитания оператор (–), 158

Г

гипертекстовые ссылки (поддержка HTML в текстовых полях), 1062
вызов функций ActionScript из, 1068
выполнение JavaScript из, 1068
не добавляются в порядок обхода, 1063
главные фильмы, 377
ссылка на, 395
ссылка на документы через _leveln, 402
ссылка на текущий уровень через _root, 401
удаление, 408
явные и неявные ссылки, 396
глобальные
переменные, 85
свойства, 484
ссылки, 91
функции, 54, 484
target, необходимый параметр, 411
перекрытие методов и глобальных функций, 411
годы в дате, получение, 561
головка воспроизведения
воспроизведение временных диаграмм из заданного кадра, 596, 720
остановка в заданном кадре, 721
перемещение на кадр вперед, 772
циклы временной диаграммы, 217
группирования оператор, 177

Д

дата
вычисление в миллисекундах, 558
прибавление/вычитание дней, 559
применение пользовательского формата, 560
двоичные числа и поразрядные операторы, 183
двойная точность , 118
двойные кавычки в строках, 123
двумерные масссивы, 315
декрементирования (––) оператор, 158
деления оператор (/), 121, 159
деления по модулю (%) оператор, 160
деления по модулю с присваиванием (%=) оператор, 154
деления с присваиванием (/=) оператор, 154
десятичный логарифм e, 664
действительные числа, 116
действия (Actions)
в сравнении с функциями, 196, 252
динамическая область видимости, 239
динамическая установка типа данных для переменных, 82
динамически именуемые переменные, 77
динамические функции, 234
доступ на этапе исполнения, влияющий на атрибуты HTML, 1086

Ж

жесткий перевод строки, добавление в текст, 124

З

заглавные буквы в переменных, 65
заголовки циклов, 210
закодированные строки, создание, 574
запятая (,) оператор, 178
в циклах for, 213
зарезервированные слова, 446
зациклившиеся сценарии, проблемы прерывания, 217
звук, экспортированный из библиотеки, ошибка реализации во Flash 6, 834
значения данных и логические выражения, 172

И

идентификаторы
в операциях присваивания, 156
правила образования, 447
создание имен экземпляров, 385
иерархия объектов (XML, класс), 977
именование переменных, 65, 76
именованные элементы массивов
создание/обращение к, 299
удаление, 300
импорт кода
#include, директива, 458
на этапе исполнения и библиотеки совместного доступа, 458
индексные переменные в циклах, 212
индексы символов
строковые функции и, 133
инициализация переменных, 62
инициализация цикла, 207, 209
инкрементирования (++) оператор, 157
инструкции, 185
else, 199
else if, 201
switch, 202, 204
эмуляция во Flash_5, 205
в сравнении с действиями, 196
вызовы функций, 190, 192
выражений, 190
добавление в панели Actions, 42
завершение, 45, 442
ограничители блока ({}), 187
подынструкции, 187
синтаксис, 186
типы, 186
условные, 190
циклов, 190
интерпретатор ActionScript (*см.* ActionScript, интерпретатор)
интерфейс с буксировкой, создание, 707
иррациональные числа, 120, 656
итераторы в циклах, 212
итерации в циклах, 51, 210

К

кавычки в строках, 123
кадры
Frame Actions, окно, 63
вывод в случайном порядке, 667
выполнение удаленных сценариев, 530
добавление пустых ключевых, 70
добавление сценариев в, 453
ключевые как устройства хранения, 453
метки в сравнении с номерами, 64
добавление, 64
структура слоев и, 59
перемещение головки воспроизведения на кадр вперед, 772
прикрепление кода к, 42, 453
программирование на основе событий и, 57
создание переменных главной временной диаграммы, 84
текущее положение головки воспроизведения, 704
категорирование данных, 102
квадратный корень из 2, 670
клавиатура, обнаружение команд, 610
клавиши сокращенного набора, перехват, 620
классы
встроенные, 359, 485
методы, создание, 331, 333
модификация свойств объектов, 329
надклассы и подклассы, 338
объектно-ориентированное программирование и, 319, 327
создание, 328
создание экземпляров, 329
классы-оболочки
Boolean, 506
String, 866
клипы, 374
with, инструкция и, 194
абсолютные пути к, 407, 884
вложенные, доступ через оператор точка, 375
вычисление расстояния между, 98
главных фильмов (*см.* главные фильмы), 377
клипы
добавление сценариев в, 455
иерархия, 398
имена экземпляров
использование, 396
назначение, 385
как кнопки, 414, 417
компоненты, 440
определение, 460
построение стандартных интерфейсов для, 461
круговое движение, 661
методы, 681
перекрытие методов и глобальных функций, 411

клипы
непрозрачность, задание, 686
новые экземпляры и Flash Player 6, 379
обеспечение независимости подклассов, 430
обрабатывающие, 378
обработчики событий
onEnterFrame(), 738
выгрузка, 750
обычные клипы фильмов, 377
ограничивающие прямоугольники, определение, 714
остановка, 756
переменные
время жизни, 92
доступ с помощью оператора «точка», 375
на разных временных диаграммах, 86
поворот, 752
подклассы, 440
получение фокуса, 417
порождающие клипы, 379, 573
дублированные экземпляры, трансформированные, 381
прикрепление кода к, 42
прикрепление обработчиков событий к, 273, 275
прозрачность, задание, 686
рисование на этапе исполнения, 413, 414
свойства, 678
доступ с помощью оператора «точка, 375
получение значений, 587
связанные клипы, 378
символы, 379
экспортирование из библиотеки, 382
создание, 379
создание копий, 708
сценариев, клипы, 378
типы, 376
точка регистрации, 685
_x, свойство и, 945
_y, свойство и, 947
тройки RGB, 545
getRGB(), метод, 549
setRGB(), метод, 551
удаление из Player, 752, 762
управление методами Flash 4, 1049
фокус ввода с клавиатуры и, 418
цвет и прозрачность, программирование, 545
клипы
экземпляры, 55, 379
attachMovie(), метод, 384
вложенные, 398
динамическое создание ссылок, 404
изменение уровней, 390
назначение уровней в стеках программно генерируемых клипов, 392
программно генерируемые, 391
расположение в стеке, 389
создание, 380, 385, 390
создание из экспортированных библиотечных символов, 382, 687
создание с помощью duplicateMovieClip(), 381
создание ссылок с помощью Insert Target Path, 403
ссылки на, 395
стеки документов (уровней), 393
текущий, ссылка на, 396
удаление, 408
ключевые кадры, добавление, 70
ключевые слова, 186
зарезервированные, 446
чувствительность к регистру для, 448
кнопки
выключение подсветки, 577
добавление кода в, 64, 454
прикрепление обработчиков событий к, 273, 275
точки регистрации, 528, 529
код
асинхронный (основанный на событиях), 263
добавление в кнопки, 454
импорт, директива #include, 458
комментарии в, 62, 444
код
обнаружение скрытого, 456
оптимизация создания, 457
организация в функции, 257
прикрепление к кадру, 42, 453
синхронный, 263, 453
централизация, 256, 286, 455
кодированные строки, создание, 122
кодировка символов, 122, 1035, 1040
кодовые позиции символов Latin 1, 123
и коды клавиш, 611
получение, 144
создание строк из, 143
сортировка массивов, 310

коды клавиш, 614
комментарии в коде, 62, 444
вложенные, 445
компоненты
xch (обмен) экземпляров и, 464
добавление пользовательских интерфейсов в, 466
использование в фильмах, 467
настройка, 462
параметры
удаление/переупорядочение, 464
построение пользовательских интерфейсов для, 464
построение стандартных интерфейсов для, 461
средства для создания профессиональных, 467
компоненты Flash UI (FUI)
FComboBox, 1072
FListBox, 1072
компоненты клипов, 440
определение, 460
создание, 437
шаблон, 435
конкатенации (&) оператор (Flash 4), 1051
конкатенации (+) оператор, 46, 157
преобразование данных в строки, 108
константы
e (основание натуральных логарифмов), 662
возведение в степень, 662
математические, 120, 656
конструктор, функция
добавление методов в классы, 336
объекты класса, 329
определение, 328
переопределение, 447
контейнеры данных, хранение ссылок на клипы в, 405
копирование данных, 112
косинус, вычисление, 661
круг, перемещение клипа по, 661
круглые скобки ()
оператор вызова функции, 225
оператор группирования, 177
приоритеты операторов и, 152
курсивный текст (поддержка HTML в текстовых полях), 1060
курсор
(*см. также* пользовательские курсоры мыши), 660
отображение, 677
разворот клипа в сторону курсора, 660
скрытие, 673
кэширование, предотвращение с помощью loadVariables(), 729

Л

лексическая область видимости, 239
лексическая структура ActionScript, 441
литералы, 49
в сравнении с переменными, 102
объектов, 347
строковые, 123
функций, 234
добавление методов в классы, 336
целочисленные, 117
чисел с плавающей точкой, 117
логарифм
(*см. также* натуральный логарифм), 662
e (по основанию 2 и 10), 664
логика в фильмах Flash, 50
логические выражения и значения данных, 172
логические операторы, 171
И (&&) (Flash 5), 175
И (and) (Flash 4), 446
ИЛИ (||) (Flash 5), 173
ИЛИ (or) (Flash 4), 446
НЕ (not) (Flash 4), 446
отрицания (!) (Flash 5), 176
локальная область видимости функций, 237
локальные переменные, 85, 1051
нелокальные переменные и, 94
создание, 240

М

маркеры комментариев /* */, 445
маскированные слои, поиск, 456
массивы, 290
анатомия, 291
в сравнении с объектами, 322
вложенные (*см.* вложенные массивы)
добавление элементов
без присвоения значений, 301
в конец с помощью push(), 301
в начало с помощью unshift(), 302
присвоением значений, 301
с помощью splice(), 303
именованные элементы
создание/обращение к, 299
удаление, 300
как объекты, 314
многомерные, 315
обработка данных с помощью циклов, 208

обращение порядка, 498
преобразование в строки
с помощью join(), 312, 496
с помощью tostring(), 313
размер, определение, 297
создание
из подмножества элементов существующего массива, 164
использование литералов массивов, 292
путем добавления элементов в конец, 305, 495
с помощью оператора new и конструктора Array(), 294
сортировка, 310, 501, 503
удаление элементов
из конца с помощью pop(), 307
из начала с помощью shift(), 308
с помощью splice(), 303, 309
с помощью оператора delete, 306
с помощью свойства length, 306
элементы, 291
извлечение значений, 295
инвертирование порядка, 309, 498
индексы, указание, 295
определение количества, 297
присвоение значений, 296
сортировка, 310
ссылки на, 295
математические константы, 120, 656
математические функции, 121, 656
меньше (<), оператор, 131, 168
меньше (lt), оператор (Flash 4), 1051
меньше или равно (<=), оператор, 169
меньше или равно (le), оператор (Flash 4), 1051
месяц в дате, получение, 562
метки кадров, 59
методы 319, 326
альтернативные способы назначения, 336
доступа, 333, 335
клипов, 681
объектов, инструкция with и, 194
определение, 55
перекрытие методов и глобальных функций, 411
статические, 337
эквивалентность методов и свойств Java и ActionScript, 338
экземпляров и переменные, 338
миллисекунды в дате
расчет даты и, 558, 562
получение, 561
минус (–), оператор вычитания, 121, 158
минуты в дате, получение, 562
многобайтовые и однобайтовые операции, 1053
многомерные массивы, 315
многострочные комментарии, 445
множественное наследование, 338
модель исполнения, управляемого событиями, 56, 263
модульный код в ActionScript, 52
мышь
отображение курсора, 677
разворот клипа в сторону курсора, 660
скрытие курсора, 673

Н

надклассы
наиболее сложные вопросы, 352
определение принадлежности объектов, 344
подклассы и, 339, 358
пример, 341
создание, 339
наследование, 338, 346
глобальные переменные/методы, 345
надклассы и подклассы, 338
полиморфизм и, 342
система ActionScript, 348, 359
натуральный логарифм
10 (LN10, свойство), 663
2 (LN2, свойство), 663
вычисление, 664
основание (константа e), 662
возведение в степень, 662
не равно (!=), оператор (Flash 5), 130
не равно (ne), оператор (Flash 4), 1051
нечисло, свойство, 104
недокументированные функции
#strict, прагма, 865
Flash MX, 24
LoadVars.onData(), обработчик событий, 639
Object.hasOwnProperty(), метод, 783
System.showSettings(), метод, 882
TextField.condenseWhite, свойство, 897
XMLnode, класс, 1013
нелокальные переменные, 94
необъявленные переменные, 1058
непрозрачность в клипах, задание, 686
неравенства операторы
!= (Flash 5), 130, 165

ne (Flash 4), 165
строгого, 166
нестандартные обработчики событий on() и onClipEvent(), 1058
неявное и явное объявление переменных, 75
новое в ActionScript для Flash MX , 17

О

область видимости
обработчик события, 275
переменных, 83
глобальных и локальных, 85
на разных временных диаграммах, 86
функций, 237
обновленные сборки проигрывателя
Flash Player 6.0.29.0, 1046
Flash Player 6.0.47.0, 1045
Flash Player 6.0.61.0, 1044
обновляющие инструкции в циклах, 207, 210
обрабатывающие клипы, 378
обработчики событий, 56, 265
loadMovie() и, 735
loadVariables() и, 734
on() и onClipEvent(), 272
клипы фильмов, 683
кнопок/клипов, 273
копирование, 284
нестандартные, 1058
область видимости, 96, 275, 281
порядок выполнения, 281
прикрепление к кнопкам и клипам, 273, 275
применение, 287
свойства, 266
срок жизни, 275
сценарии для кнопок, 66
централизация кода, 286
эмуляция динамической обработки событий, 286
обратная величина для корня из 2, 670
объект узла, 978
объекта свойство (оператор []), 325
объектно-ориентированное программирование (ООП), 54, 319, 373
классы и, 327
книги, 1029
краткий справочник по, 362, 367
надклассы и подклассы, 338, 346
полиморфизм, 342
ресурсы, 1029, 1033
сводка, 372
терминология, 319, 321
объекты, 54, 319
анатомия, 322
в сравнении с массивами, 322
встроенные, 360
документы XML, обработка, 977
литералы, 347
массивы как, 314
методы, 194, 326
альтернативные способы назначения методов, 336
доступа, 333, 335
определение и вызов, 333
пример программирования, 320
принадлежащие надклассам, 344
свойства, 193
модификация, 329, 331
точка (.), оператор и, 324
циклы for-in и, 213
создание экземпляров, 323
структура (диаграмма), 323
обычные клипы фильмов, 377
ограничивающие прямоугольники, 714
ограничители блока инструкций ({})
точка с запятой не требуется, 444
одинарные кавычки (') в строках, 123
однобайтовые и многобайтовые операции, 1053
однострочные комменарии, 445
окна, открытие из Flash, 589
округление чисел до произвольного числа десятичных знаков, 118
операнды, 47, 151
количество, 152
типы данных, 164
операторы, 46, 151
Flash 4 в сравнении с Flash 5, 1051
instanceof, 181
number, тип данных и, 121
super (инструмент типа оператора), 182
арифметические, 157
ассоциативность, 155
вызова функции, 225
выражения и, 151
поразрядные, 183
приоритеты, 121, 152
строгого равенства и неравенства, 166
сложное присваивание, 156
сравнения (или отношения), 167
условный (?:), 200
элемента массива/свойства объекта, 180, 293

основание 10, логарифм e, 664
основание 16
escape-последовательности, 125
значения цветов, 549
основание 2, логарифм e, 664
отбрасывание дробной части чисел, 607
относительные ссылки, 89
ошибки реализации
#strict, прагма, 865
arguments, объект, применение, 493
duplicateMovieClip(), глобальная функция, 574
Flash 5
Array.sort(), метод, 500
Math.LOG2E, свойство, 664
Math.pow(), метод, 667
MoveClip.onData(), обработчик события, 736
выполняемые проекторы, 629
Flash 6
Array.sortOn(), метод, 502
Button.onKillFocus(), 517
capabilities.pixelAspectRatio, свойство, 539
Object.addProperty(), метод, 782
position, свойство в экземплярах Sound, 845
scroll или hscroll текстового поля, 918
TextField.getFontList(), метод, 901
TextField.prototype и встроенные свойства, 890, 899
TextField.wordWrap, свойство, 944
встроенные шрифты, 806
документ XML с объявлением XML, 987
загрузка фильма по URL, 828
присваивание пустой строки свойству TextField.restrict, 926
пустые строки как разделители, 874
строка сервера для версии, 544
экспорт звука из библиотеки, 834
getter/setter, свойство и _global, объект, 595
gotoAndStop(), глобальная функция, 598
loadVariables(), глобальная функция, 629
LoadVars.getBytesLoaded(), метод, 636
LoadVars.send(), метод, 643
Sound.id3, свойство, 840
Sound.onLoad(), обработчик события, 843
Stage.addListener(), обработчик события, 858
Stage.height, свойство, 856
Stage.width, свойство, 862
быстрый ввод текста в текстовое поле, 915

П

пакетные и управляемые событиями программы, 56
параметры, 45
в сравнении с аргументами, 228
вызов функций с, 228
доступ к дополнительным через объект arguments, 243
доступность инструкций, 237
количество, 242
компонента, удаление/переупорядочение, 464
определение, 228
передача по значению и по ссылке, 245
получение значений из массива arguments, 243
создание функций с, 227
первым пришел – первым ушел (FIFO), очереди, 303
перевод строки, 124, 771
в коде ActionScript, 441
передача данных, 112
переменные, 74
undefined, тип данных, 147
автоматическая типизация, 82, 103
автоматическое преобразование значений, 82
в сравнении с литералами, 102
временной диаграммы, 86
доступность, 238
инициализация, 259
предотвращение конфликтов с помощью локальных , 241
создание, 84, 241
временные, 238
глобальные, 85
динамическая установка типа данных, 82
доступ в различных временных диаграммах, 88
загрузка из внешнего источника, 97
изменение значений, 80
именование, 65, 76
динамически именуемые, 77

переменные
импорт, 729
инициализация, 62
как выражения, 49
клипов, 86
время жизни, 92
локальные, 85, 1051
нелокальные и, 94
создание, 240
необъявленные, 1058
область видимости, 83, 85
на разных временных диаграммах, 86
обращение к находящимся на разных уровнях, 91
определение типа данных вручную, 83
отправка сценариям, выполняемым на сервере, 629
официальная номенклатура, 43
прикрепление к уровням документов, 630
присваивание значений, 78
проверка типов данных, строгая, 82
синтаксис с точкой и, 88
ссылки
абсолютные/относительные, 89
разрешение через цепочку областей видимости, 238
типы значений, 81
эквивалентность методов и свойств Java и ActionScript, 338
переполнение, 602
поворот клипа в сторону курсора мыши, 660
подклассы клипов, 440
подпрограммы, локальные переменные в, 1051
подсказка кода в панели Actions, 76
подстроки
извлечение из строк на основе
начальной позиции и длины, 137
положительных или отрицательных позиций символов, 138
поиск
первого вхождения, 134
последнего вхождения, 135
подчеркнутый текст (поддержка HTML в текстовых полях), 1060
подынструкции, 187
в инструкции if, 198
в циклах for-in, 213
в циклах while, 206
поиск в строках, 134
показать/скрыть
кнопки, 527
текстовые поля, 943
полиморфизм, 342
полноэкранный режим запуска в веб-броузерах, 580
положение по вертикали
клипов/фильмов, 768
курсора мыши, 769
положение по горизонтали
клипов/фильмов, 765
курсора мыши, 767
пользовательские курсоры мыши
Mouse.hide(), метод и, 673
эмуляция, 288
пользовательские события, 264
поразрядного И с присваиванием (&=), оператор, 154
поразрядного ИЛИ с присваиванием (|=), оператор, 154
поразрядного исключающего ИЛИ с присваиванием (^=), оператор, 154
поразрядного сдвига вправо со знаком с присваиванием (>>=), оператор, 154
поразрядные операторы, 183
порождающие клипы, 379, 573
наследование трансформаций, 381
порядок выполнения
кода в слоях, 395
обработчиков событий клипов, 281
порядок обхода при табуляции, пример, 814
последним пришел – первым ушел (LIFO), стеки, 303
постоянные преобразования типа данных, 110
постфиксного инкрементирования (++), оператор, 157
предзагрузчики
_framesloaded, свойство и, 712
создание и булевы выражения, 146
преобразование
операции равенства и преобразования типов, 164
регистра символов, 141
температуры, пример, 99
типов данных
Flash 4 во Flash 5, 1051
автоматическое, 82, 104
в булев тип, 106, 109
в строки, 106
в числа, 105, 108, 157
временное, 110
операции сравнения и, 170

постоянное, 110
явное, 107
префиксного декрементирования (––), оператор, 159
префиксного инкрементирования (++), оператор, 157
приемники событий, 268, 271
приоритеты операторов, 121, 152
присваивания оператор (=), 155
присваивание значений переменным, 78
пробельные символы
в коде, 441
удаление из документв XML, 979
проблемы (*см.* ошибки реализации), 493
проверочные выражения в циклах, 210
программирование
введение в, 37
краткий справочник по ООП, 362, 367
игр, ресурсы, 1031
программные функции, 224
доступность, 235
прозрачность в клипах, 545
задание, 686
проигрыватель (*см.* Player)
производительность
при извлечении подстрок, 140
пространства имен, 368
пространства координат (MovieClip, класс), 685
globalToLocal(), метод и, 719
localToGlobal(), метод и, 730
процентные соотношения цветов, 545, 554
пустые
инструкции, 195
ключевые кадры, добавление, 70
строки, 123

Р

рабочий стол
globalToLocal() и, 719
высота, 680, 721
обновление и циклы временной диаграммы, 218
ширина, 680, 765
равенства операторы
= = (Flash 5), 129, 161, 166
eq (Flash 4), 1051
= = = строгого, 166
равенство элементарных типов данных, 162
разделители
в строках, преобразованных из массивов, 312
аргументов, 45
размытие (*см.* сглаживание)
регулярные выражения, отсутствие поддержки, 136, 1056
режимы (панель Action)
Expert, 41
Normal, 39
рекурсия
нахождение всех клипов временной диаграммы, 406
создание функций для, 246, 284
рисование в фильме на этапе исполнения, 413, 414

С

самодокументированный код, 444
свойства
глобальные, 484
клипов фильмов, 678
получение значений, 587
константы объекта Math, 120, 656
конструктор, 351
неточность в дробной части значения, 679
объектов, 323
инструкция with и, 193
оператор точка (.) и, 180
разрешение ссылок через цепочку областей видимости, 238
определение, 55
прикрепление к функциям, 254
текстового поля, 896
только для чтения и для чтения/записи, 679
связанные клипы, 378
сглаживание, 766, 945, 947
аппроксимация дробных значений
_x, 766, 945
_y, 947
сдвига вправо с присваиванием (>>=), оператор, 154
сдвига операторы, поразрядные, 153
секунды даты, получение, 562
символы пробелов в коде, 441
символы табуляции в коде, 441
символы клипов, 379
экспорт из библиотеки, 382
синтаксис
ActionScript, 441
инструкции, 186
обращения к переменным, 1050
с точкой
доступ к переменным, 88
цепочка областей видимости, 239

синус, вычисление, 669
синхронизированная анимация, 221
синхронное выполнение кода, 453
системные события, 264
скобки (), присваивание и вызов методов, 333
скорость вывода кадров, влияние на циклы событий временных диаграмм и клипов, 221
скрытые слои, поиск, 456
сложения (+) оператор, 46, 121, 157
преобразование данных в строки, 108
сложные выражения, 102
автоматическая типизация, 103
условные проверочные выражения, 198
сложные типы данных, правила для, 163
слои, поиск скрытых, 456
случайные кадры, вывод, 667
слэш-нотация и Tell Target, 1057
смещения для цветов, 545, 554
события, 263, 289
клипов, 683
обработчики, 455, 683, 684
циклы, 220
влияние частоты кадров на, 221
кнопок, обработчики, 454
приемников, 268, 271
типы, 264
составные и элементарные данные, 111
составные операторы присваивания, 156
состояния приложения, 61
специальные клавиши, коды, 1040
специальные символы, создание, 143
сравнение данных, 112
сравнения операторы
преобразование типов данных и, 170
различие для элементарных и составных типов данных, 114
среда разработки, 450
Actions, панель, 450
поиск кода, 456
расположение кода ActionScript во внешних файлах, 458
советы по повышению производительности, 457
ссылки
абсолютные/относительные, 89, 401
глобальные, 91
динамическое создание объектов клипов, 404
на вложенные экземпляры, 398
разрешение через цепочку областей видимости, 238
создание с помощью Insert Target Path, 403
статическая область видимости, 239
статические методы и свойства, 337
стеки
pop()/push() и, 303
внутренних слоев, 390
документов, 393
порядок исполнения, 395
порядок помещения на стек, 389
программно генерируемые, 391
строка сервера для версии, ошибка в документации Flash_MX, 544
строки
выполнение кода с помощью eval(), 144
извлечение
исследование строк и, 133, 140
символов из заданных позиций, 134
подстрок на основе
начальной позиции и длины, 137
положительных или отрицательных позиций символов, 138
индексы символов, 133
кавычки в, 123
кодирование, 122, 574
конкатенация, 127
массивы, 504
на верхнем регистре, создание, 141
на нижнем регистре, создание, 141
операторы равенства/неравенства, 129
описание объектов как, 790
перенос, поддержка HTML в текстовых полях, 1062
поиск и замена, 870, 875
поиск подстрок
первого вхождения, 134
последнего вхождения, 135
получение
кодовых позиций заданных символов, 144
количества символов, 133
преобразование
в идентификаторы, 144, 575
массивов в, 312, 313, 496
производительность извлечения подстрок, 140
пустые, 123
создание строк из кодовых позиций, 143
сравнение, 129, 167
учет регистра, 141

строковые литералы, 123
строковые операторы, 171
структура слоев в фильмах Flash, 58, 390
сценарии, 48
выполняемые удаленно, 530
добавление
в кадры, 453
в клипы, 455
в кнопки, 454
поиск (Movie Explorer), 457
создание
переменные главной временной диаграммы, 84
сценарий расчета, 70
функции в, 52
сцены, 374
(*см. также* главные фильмы)
в сравнении с псевдосценами, 596
перемещение головки воспроизведения к следующей сцене, 772
предыдущая сцена, 799
счетчики в циклах, 209, 212

Т

тангенс, вычисление, 670
тайм-ауты, эмуляция, 217, 1057
таймер обратного отсчета времени, создание, 588
теги якоря (поддержка HTML в текстовых полях), 1062
вызов функций ActionScript из, 1068
выполнение JavaScript из, 1068
не добавляются в порядок обхода, 1063
текст, быстрый ввод в текстовое поле, ошибка реализации, 915
текстовые поля
Embed Fonts, опция, 910
scroll, свойство, 896
Selection, объект, 889
заключение в кавычки значений атрибутов, 1065
нераспознаваемые теги и атрибуты HTML, 1066
отсутствие текста, 1066
получение индекса точки ввода, 809
преобразование в числа, 108
свойства, 896
ширина, задание, 947
текстовые узлы, 1000
пробельные символы в, 984
тело функции, 225
тело цикла, 210
тернарные операторы, 152
типы данных, 100
булев, 109
встроенные ActionScript, 101
определение вручную, 83, 110
переменные и, 81
преобразование, 104
Flash 4 во Flash 5, 1051
в булевы значения, 106
в строки, 106
в числа, 105, 157
операции равенства и, 164
операции сравнения и, 170
создание данных, 102
числовой, 108
элементарные, 116
в сравнении с составными, 111
равенство, 162
только для чтения и для чтения/записи, свойства, 679
точка (.), оператор, 180
доступ к свойствам клипа, 375
доступ к свойствам объекта, 324
обход иерархии экземпляров клипов, 398
точка с запятой (;), оператор
завершение инструкций с помощью, 186, 226
точки регистрации клипов, 685
_x, свойство и, 766, 945
_y, свойство и, 947
точки регистрации кнопок
_x, свойство и, 528
_y, свойство и, 529
точность представления чисел с плавающей точкой, 118
трансформации (настройка цветов), 546, 554
тригонометрия
арккосинус, вычисление, 658
арксинус, вычисление, 658
арктангенс, вычисление, 659
косинус, вычисление, 661
синус, вычисление, 669
тангенс, вычисление, 670

У

удаление локальных переменных, 93
узел, объект (XML, класс), 978, 981
узлы элементов, 1000
умножения оператор (*), 121, 159
умножения с присваиванием (*=), оператор, 154
унарные операторы, 152
унарный (–) оператор отрицания, 160

упаковка графики компонента, 435
управление циклом, 210
управляемая событиями модель исполнения, 56
уровни документов, 91
_leveln, глобальное свойство, 393, 402, 622
загрузка файлов SWF посредством loadMovie(), 393, 402
прикрепление внешних переменных к, 630
удаление с помощью unloadMovie(), 408
условные выражения, 197
блоки инструкций и, 205
условные инструкции, 50, 190
else, 199
else if, 201
switch, 202, 204
эмуляция во Flash_5, 205
в сценарии расчета, 71
условный (?:) оператор, 181, 200

Ф

факториалы, вычисление с/без рекурсии, 247
файлы с исходным кодом учебных примеров, 28
файлы сценариев, импорт текста из внешних файлов, 458
фильмы, использование компонентов, 467
формы Flash
данные
ввод клиентом, 470
обработка и результаты, 471
отправка серверу, 470
получение результатов с сервера, 472, 478
проверка, 470
добавление кнопки Reset, 479
проблемы взаимодействия клиент/сервер, 470
создание заполняемых, 472
цикл данных, 469
функции, 52, 224
arguments, объект, 243
в сравнении с действиями, 252
вложенные, 248
возврат значений из, 232
возврат функций из функций, 255
встроенные (*см.* встроенные функции)
функции
вызов, 225, 235, 326
для инициализации, 62
операторы, 326
с параметрами, 228
динамические, 234
доступность, 235
завершение, 231
запуск, 225
инициализация, 62
как объекты, 252
обеспечение сохранности, 236
области видимости
лексическая и динамическая, 239
локальная, 237
обратного вызова, 67
объявление, 225
организация кода в, 257
параметры, 242
передача по значению и по ссылке, 245
передача
информации в, 226
нескольких аргументов, 229
рекурсивные, 246
свойства, прикрепление к, 254
создание, 225
с параметрами, 227
сравнения, создание, 310
срок жизни, 236
централизация кода, 256
цепочка областей видимости и, 240
элементарные типы данных в сравнении с составными, 115

Ц

цвет
альфа-составляющая, 545
генерирование случайного, 548
задание для клипа, 554
значения смещений, 545, 554
изменение яркости, 548
процентные отношения, 545, 554
трансформации, 546, 554
установка для клипа, 551
целочисленные литералы, 117
целые числа, 116
цепочка областей видимости, 95, 238
цепочка прототипов, 348, 359
обход, 344
циклы, 190, 206
break, инструкция в, 215
continue, инструкция в, 216

циклы
for, 212, 213
for-in, 213
while, 206
бесконечные, 207, 210
способы избежать, 216
вложенные, 210
временной диаграммы, 217
альтернативы, 223
в сравнении с обычными циклами, 218
влияние частоты кадров на, 221
пустого клипа, создание, 219
обрабатывающего клипа, создание, 378
досрочное завершение, 215
индексные переменные, 212
инициализация, 207, 209
итераторы, 212
итерации в, 51, 210
проверочные инструкции в, 210
событий клипов (*см.* события клипов, циклы), 220
счетчики в, 209, 212
терминология, 209
инструкции обновления в, 207, 210

Ч

частное, 159
частота кадров, инструмент для измерения, 222
часы в аналоговом стиле, создание, 419
числа
большее из двух, определение, 665
возведение в степень, 118, 666
вычисление абсолютного значения, 657
квадратный корень, вычисление, 670
меньшее из двух, определение, 665
натуральные логарифмы, вычисление, 664
округление, 668
до ближайшего большего целого, 661
до ближайшего меньшего целого, 121, 663
определение угла по точке, 659
отбрасывание дробной части, 607
с плавающей точкой, 116
parseFloat() и, 109
литералы, 117
точность представления, 118
случайные, генерирование, 121, 667
сравнение, 167
числитель, 159
число месяца в дате, получение, 561
числовые литералы, 116
чувствительность к регистру в ActionScript, 447

Ш

шестнадцатеричные (по основанию 16)
escape-последовательности, 125
значения цветов, 549
ширина клипов/фильмов
в пикселах, 765
в процентном отношении, 768
шрифты, встроенные, 898

Э

экземпляры клипов, 55
создание пустых новых , 384, 385
элемент массива/свойство объекта, оператор, 180, 293, 325, 404
элементарные
данные в сравнении с составными, 111
значения, булевы, 506
типы данных, 116
равенство, 162
элементы массива, именованные, 298

Я

явное и неявное объявление переменных, 75
явное преобразование типа данных, 107
яркость, установка с помощью объекта Color, 548

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-068-5, название «ActionScript для Flash MX. Подробное руководство» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.