*Flash-программирование*

Предисловие Гари Гроссмана, создателя ActionScript

# ActionScript

## *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 5-93286-028-6, название «ActionScript. 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

*The Definitive Guide*

*Colin Moock*

O'REILLY®

# ActionScript

## *Подробное руководство*

*Колин Мук*

*Санкт-Петербург*
*2002*

Колин Мук

# ActionScript. Подробное руководство

Перевод С. Маккавеева


| | |
|---|---|
| Главный редактор | *А. Галунов* |
| Зав. редакцией | *Н. Макарова* |
| Редактор | *Т. Маслова* |
| Корректура | *С. Журавина* |
| Верстка | *Н. Гриценко* |

*Мук К.*
ActionScript. Подробное руководство. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 792 с., ил.
ISBN 5-93286-028-6


Издание полностью посвящено описанию ActionScript – объектно-ориентированного языка программирования Flash – и адресовано как разработчикам Flash, делающим первые шаги в программировании, так и тем, кто использует свое знание JavaScript при переходе на ActionScript (оба языка основаны на стандарте ECMAScript).

С помощью этой книги новички быстро осваивают программирование на ActionScript. Опытные программисты могут при изучении сложных вопросов, специфичных для Flash, поднять уровень своих знаний по JavaScript. Помимо теории в книге есть масса практических советов и примеров, в том числе касающихся текстовых полей с прокруткой, кнопок меню, вопросников с вариантами ответов, сайтов, управляемых XML, видеоигр с использованием законов физики, многопользовательских сред реального времени и многого другого. Книга содержит описание многих недокументированных или недостаточно документированных тем. Исчерпывающий и точный «Справочник по языку» послужит незаменимым попутчиком в ежедневной работе.


**ISBN 5-93286-028-6**
**ISBN 1-56592-852-0 (англ)**



© Издательство Символ-Плюс, 2002
Authorized translation of the English edition © 2001 O'Reilly & Associates Inc. This translation is published and sold by permission of O'Reilly & Associates 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». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 31.10.2001. Формат 70x100$^{1}/_{16}$ . Бумага офсетная.
Печать офсетная. Объем 49,5 печ. л. Тираж 3000 экз. Заказ N
Отпечатано с диапозитивов в ФГУП «Печатный Двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания
и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

# Оглавление

**Предисловие** . . . . . 11

**Введение** . . . . . 15

**Часть I. Основы ActionScript** . . . . . 23

**1. Легкое введение для непрограммистов** . . . . . 25

Некоторые основные фразы . . . . . 27
Другие понятия ActionScript . . . . . 36
Создание вопросника с вариантами ответов . . . . . 45
Вперед! . . . . . 59

**2. Переменные** . . . . . 60

Создание переменных (объявление) . . . . . 61
Присваивание значений переменным . . . . . 63
Изменение и извлечение значений переменных . . . . . 64
Типы значений . . . . . 66
Область видимости переменной . . . . . 68
Некоторые практические примеры . . . . . 80
Вперед! . . . . . 81

**3. Данные и типы данных** . . . . . 82

Данные и информация . . . . . 82
Сохранение смысла данных с помощью типов данных . . . . . 83
Создание данных и задание их категорий . . . . . 84
Преобразование типов данных . . . . . 86
Элементарные и сложные типы данных . . . . . 96
Вперед! . . . . . 97

**4. Элементарные типы данных** . . . . . 98

Тип Number . . . . . 98
Целые числа и числа с плавающей точкой . . . . . 98
Числовые литералы . . . . . 99
Действия с числами . . . . . 103

Строковый тип данных . . . . . 104
Работа со строками . . . . . 109
Булев тип . . . . . 129
Тип undefined . . . . . 131
Тип null . . . . . 132
Вперед! . . . . . 132

**5. Операторы** . . . . . 133

Общие характеристики операторов . . . . . 133
Оператор присваивания . . . . . 138
Арифметические операторы . . . . . 139
Операторы равенства и неравенства . . . . . 144
Операторы сравнения . . . . . 149
Строковые операторы . . . . . 152
Логические операторы . . . . . 153
Оператор группирования . . . . . 159
Оператор запятая . . . . . 159
Оператор void . . . . . 160
Прочие операторы . . . . . 160
Вперед! . . . . . 164

**6. Предложения** . . . . . 165

Типы предложений . . . . . 166
Синтаксис предложений . . . . . 166
Предложения ActionScript . . . . . 168
Сравнение предложений с действиями . . . . . 176
Вперед! . . . . . 176

**7. Условные предложения** . . . . . 177

Предложение if . . . . . 178
Предложение else . . . . . 180
Предложение else if . . . . . 182
Эмуляция предложения switch . . . . . 183
Компактный синтаксис условных предложений . . . . . 185
Вперед! . . . . . 185

**8. Предложения цикла** . . . . . 186

Цикл while . . . . . 186
Терминология циклов . . . . . 190
Цикл do-while . . . . . 191
Цикл for . . . . . 192
Цикл for-in . . . . . 193

Досрочное завершение цикла . . . . . 195
Циклы временной диаграммы и событий клипа . . . . . 198
Вперед! . . . . . 205

**9. Функции** . . . . . 206

Создание функций . . . . . 207
Запуск функций . . . . . 207
Передача функции информации . . . . . 208
Выход из функций и возвращение значений . . . . . 211
Литералы функций . . . . . 214
Доступность и срок жизни функций . . . . . 215
Область видимости функции . . . . . 216
Еще раз о параметрах функции . . . . . 221
Рекурсивные функции . . . . . 225
Внутренние функции . . . . . 227
Функции как объекты . . . . . 228
Централизация кода . . . . . 230
Еще раз вопросник с вариантами ответов . . . . . 230
Вперед! . . . . . 234

**10. События и обработчики событий** . . . . . 235

Синхронное выполнение кода . . . . . 235
Асинхронное выполнение кода, основанное на событиях . . . . . 236
Типы событий . . . . . 236
Обработчики событий . . . . . 237
Синтаксис обработчика событий . . . . . 238
Создание обработчиков событий . . . . . 238
Область видимости обработчиков событий . . . . . 242
События кнопок . . . . . 246
Обзор событий клипов . . . . . 251
События клипов, связанные с воспроизведением фильмов . . . . . 252
События клипа, связанные с пользователем . . . . . 258
Порядок выполнения кода . . . . . 264
Копирование обработчиков событий клипа . . . . . 267
Обновление экрана с помощьюupdateAfterEvent . . . . . 267
Повторное использование кода . . . . . 269
Динамические обработчики событий клипов . . . . . 269
Применение обработчиков событий . . . . . 270
Вперед! . . . . . 272

**11. Массивы** . . . . . 273

Что такое массив? . . . . . 273
Анатомия массива . . . . . 274

Создание массивов . . . . . 276
Обращение к элементам массива . . . . . 278
Определение размера массива . . . . . 280
Именованные элементы массивов . . . . . 282
Добавление элементов в массив . . . . . 283
Удаление элементов из массива . . . . . 289
Общие средства обработки массивов . . . . . 292
Многомерные массивы . . . . . 298
Вопросник с вариантами выбора: подход № 3 . . . . . 299
Вперед! . . . . . 300

**12. Объекты и классы** . . . . . 301

Анатомия объекта . . . . . 304
Создание экземпляров объектов . . . . . 305
Свойства объектов . . . . . 305
Методы . . . . . 307
Классы и объектно-ориентированное программирование . . . . . 311
Встроенные классы и объекты ActionScript . . . . . 327
Вперед! . . . . . 329

**13. Клипы** . . . . . 330

«Объектность» клипов . . . . . 331
Типы клипов . . . . . 332
Создание клипов фильмов . . . . . 335
Порядок расположения фильмов и экземпляров в стеке . . . . . 343
Ссылки на экземпляры и главные фильмы . . . . . 348
Удаление экземпляров клипов и главных фильмов . . . . . 362
Встроенные свойства клипов . . . . . 365
Методы клипов . . . . . 367
Практические примеры клипов . . . . . 372
Последний вопросник . . . . . 376
Вперед! . . . . . 379

**14. Лексическая структура** . . . . . 380

Пробельные символы . . . . . 380
Символы, завершающие предложение (точка с запятой) . . . . . 382
Комментарии . . . . . 383
Зарезервированные слова . . . . . 385
Идентификаторы . . . . . 387
Чувствительность к регистру . . . . . 387
Вперед! . . . . . 388

**15. Более сложные темы** . . . . . 389

Копирование, сравнение и передача данных . . . . . 389
Поразрядное программирование . . . . . 392
Более сложные вопросы областей видимости функций . . . . . 404
Тип данных movieclip . . . . . 407
Вперед! . . . . . 408

**Часть II. Применение ActionScript** . . . . . 409

**16. Среда разработки ActionScript** . . . . . 411

Панель Actions . . . . . 411
Помещение сценариев в кадры . . . . . 414
Добавление кода к кнопкам . . . . . 415
Добавление сценариев в клипы . . . . . 417
Куда девался код? . . . . . 417
Производительность . . . . . 418
Сохранение кода ActionScript во внешних файлах . . . . . 419
Создание интеллектуальных клипов . . . . . 421
Вперед! . . . . . 428

**17. Формы Flash** . . . . . 429

Цикл данных форм Flash . . . . . 429
Создание заполняемой формы Flash . . . . . 432
Вперед! . . . . . 438

**18. Экранные текстовые поля** . . . . . 439

Динамические текстовые поля . . . . . 439
Текстовые поля для ввода данныхпользователем . . . . . 441
Параметры текстовых полей . . . . . 442
Свойства текстовых полей . . . . . 447
Поддержка HTML . . . . . 449
Работа с выделением текста в текстовых полях . . . . . 458
Пустые текстовые поля и предложение for-in . . . . . 458
Вперед! . . . . . 459

**19. Отладка** . . . . . 460

Средства отладки . . . . . 461
Методика отладки . . . . . 467
Вперед! . . . . . 472

**Часть III. Справочник по языку** . . . . . 475

**Справочник по языку ActionScript** . . . . . 477

Глобальные функции . . . . . 477
Глобальные свойства . . . . . 478
Встроенные классы и объекты . . . . . 479
Заголовки статей . . . . . 479
Алфавитный справочник по языку . . . . . 480

**Часть IV. Приложения** . . . . . 729

**A. Ресурсы** . . . . . 731

**B. Набор символов Latin 1 и коды клавиш** . . . . . 736

**C. Обратная совместимость** . . . . . 743

**D. Отличия от ECMA-262 и JavaScript** . . . . . 748

**Алфавитный указатель** . . . . . 751

# Предисловие

Когда летом 1998 года я прибыл в Macromedia, чтобы присоединиться к команде разработчиков Flash, небольшая и динамичная группа уже создала изумительный продукт. Flash 3 почти повсеместно был принят как стандарт для векторной анимации в приложениях Сети. Его преданные и энергичные пользователи из числа талантливых художников создавали впечатляющие произведения, с каждым днем появляющиеся на все большем количестве сайтов.

Корни ActionScript можно проследить до того пункта в списке планируемых характеристик Flash 4, который назывался «Повышение интерактивности». Flash 3 предлагал только базовый набор действий для управления клипами и кнопками Flash и обеспечения интерактивности. Однако я помню впечатление, которое произвели на меня сложность и временные затраты реализации с помощью действий Flash 3 игры в крестики-нолики – простой задачи для большинства языков программирования.

Это было до появления ActionScript. Сегодня никого не удивляют динамические веб-сайты, созданные исключительно с помощью Flash 4. А сейчас появляются сайты, использующие еще более изощренные возможности, предоставляемые ActionScript во Flash 5.

Главной задачей при создании ActionScript было обеспечить его доступность: важно, чтобы ActionScript было легко использовать не-программистам. Вместо чистого окна для редактирования сценариев мы создали во Flash 4 визуальный, легко понятный интерфейс для добавления к фильмам интерактивности. Простота Flash 4 ActionScript сделала легким его изучение и, что важно, позволила сохранить маленький размер Flash Player.

Flash Player создан так, чтобы быстро загружаться даже при низкой пропускной способности соединения. Команда Flash повторяет мантру «а насколько увеличится код проигрывателя?», прежде чем добавить в него новую функцию. ActionScript не был исключением из этого правила. Задача для ActionScript, как и для любой новой функции проигрывателя, в том, чтобы получить максимум шума (богатства функций) за минимум денег (увеличение размера проигрывателя).

Мы знали, что пользователи найдут для ActionScript самые неожиданные применения, и тем более приятной неожиданностью было увидеть, чего они смогли достичь с его помощью. Через месяц после выпуска Flash 4 в Сети стали появляться изумительные сайты, использующие ActionScript: сайты электронной коммерции, комнаты для чатов, доски объявлений, аркадные игры, настольные игры и даже Flash-сайты для создания Flash-сайтов. Шлюзы были открыты и впустили новую волну анимированного, интерактивного, насыщенного графикой содержимого Сети.

Когда настало время разрабатывать Flash 5, мне в первую очередь хотелось, чтобы ActionScript развился в полноценный язык сценариев с возможностями, которыми программисты привыкли пользоваться в таких языках, как JavaScript – функциями, объектами, сложными управляющими операторами и многочисленными типами данных. Такой инструментарий позволил программистам повысить свою производительность в других языках, и я хотел, чтобы ActionScript тоже поддерживал его. Вместо проектирования языка с чистого листа я решил сделать ActionScript очень похожим на JavaScript, который фактически является стандартом для сценариев, выполняющихся в Интернете на стороне клиента. Точнее, моделью для ActionScript послужил стандарт ECMAScript (ECMA-262). Поэтому для программистов JavaScript, переходящих на Flash, ActionScript довольно близок. Кроме того, программисты ActionScript смогут применить свое знание ActionScript при программировании на JavaScript и легко использовать уже существующий код в обоих языках.

Требования к доступности и минимизации размера проигрывателя оставались в силе. JavaScript является тонким и сложным языком, и мы стремились в полной мере предоставить его мощь продвинутым пользователям, в то же время сохранив простоту использования Flash 4 ActionScript. С этой целью в новой панели Flash 5 Actions существует два режима: обычный режим (Normal Mode), улучшенная версия редактора Flash 4 ActionScript, и экспертный режим (Expert Mode), текстовый редактор для опытных пользователей. Для того чтобы минимизировать размер проигрывателя, пришлось отчасти пожертвовать совместимостью ActionScript с ECMAScript. Например, ActionScript не поддерживает компиляцию кода на этапе выполнения с помощью функции *eval( )*: для поддержки такой функции пришлось бы целиком включить в проигрыватель компилятор ActionScript, что повлекло бы недопустимое увеличение его размера. По той же причине не поддерживается поиск с помощью регулярных выражений. Обе эти характеристики очень полезны и демонстрируют, какие сложные решения пришлось принимать команде Flash, чтобы обеспечить баланс между размером проигрывателя и его функциональными возможностями.

К этим двум требованиям мы добавили третье: совместимость. Flash 5 ActionScript был спроектирован так, чтобы можно было достаточно

спокойно обновить сценарии Flash 4 до синтаксиса Flash 5. Кроме того, Flash 5 поддерживает Flash 4 ActionScript в качестве подмножества, поэтому с помощью Flash 5 можно отлично создавать фильмы Flash 4. Колин Мук осветил вопросы обратной совместимости, а также основные различия между ActionScript и JavaScript (часто обусловленные соображениями совместимости) в приложении C «Обратная совместимость» и приложении D «Отличия от ECMA-262 и JavaScript».

В течение всего процесса разработки команда Flash получала бесценные данные от сообщества пользователей Flash – крепко спаянной группы, обладающей собственным голосом, огромными талантами и увлеченностью. Сообщество Flash сыграло большую руководящую роль в формировании тех функций, которые вошли в продукт. Задачей Macromedia является создание программного обеспечения, которое удовлетворяло бы потребностям ее клиентов; для этого она прислушивается к клиентам и изучает методы их работы.

В конце концов, история Flash не закончена. Это живая работа, которая постоянно ведется с целью удовлетворения ваших потребностей. Flash-разработчики – это художники века информации, и задача команды Flash состоит в том, чтобы изготовить для них самые лучшие кисти и резцы. Эта книга является первым исчерпывающим учебником и справочником, посвященным исключительно языку ActionScript. В этом качестве она знаменует поворотную точку в эволюции ActionScript: теперь ActionScript стал достаточно сложен, чтобы заслужить такую превосходную книгу, насыщенную новейшим материалом и не обошедшую вниманием ни одну возможность языка.

Наслаждайтесь этой книгой и наслаждайтесь Flash 5 ActionScript. Мы все мечтаем увидеть ваши достижения!

— Гари Гроссман
Ведущий разработчик Macromedia Flash
Март 2001

# Введение

В этой книге рассказывается как об основах языка ActionScript, так и о его продвинутом применении. Около 800 содержательных страниц посвящены подробнейшему исследованию языка ActionScript – от базовых понятий переменных и управления клипами до таких глубоких тем, как объекты и классы, связь с сервером и XML. В конечном итоге будет охвачено все, что относится к Flash-программированию.

Книга написана не только для программистов. Изложение ведется достаточно быстро, но для чтения не требуется предварительного знания программирования. Все, что требуется, – это опыт работы с той частью Flash, которая не связана с использованием ActionScript, и желание учиться. Конечно, если вы программист, это еще лучше; вы сразу сможете применить в ActionScript свое мастерство написания программ.

Данное издание содержит подробно документированный материал, который не изложен или недостаточно полно изложен в документации Macromedia или в книгах других авторов. Известно, что во Flash есть функции и технологии, о которых знают лишь посвященные, и сведения о них передаются изустно. Каким образом слои, клипы и загружаемые фильмы помещаются в стек Flash Player? (см. главу 13 «Клипы»). Как определяется порядок выполнения кода для каждого конкретного кадра? (см. главу 13). Обладают ли обработчики событий локальной областью видимости? (см. главу 10 «События и обработчики событий»). Почему число 90 иногда выводится как 89.9999999997? (см. главу 4 «Элементарные типы данных»). Я счел своей особой задачей нанести на карту эти неизведанные области. Конечно, я рассказываю и об основных приемах программирования, которые требуются в любом языке, например, о том, как обеспечить многократное выполнение участка кода (см. главу 8 «Предложения цикла»).

Эта книга задумана так, чтобы стать настольной, а не пылиться на полке. Часть III «Справочник по языку» исчерпывающим образом описывает все объекты, классы, свойства, методы и обработчики событий, имеющиеся в ActionScript. Этим справочником нужно регулярно пользоваться, чтобы узнавать новое и вспоминать вещи, которые постоянно забываются.

Прежде всего эта книга является подробным руководством. Это результат многолетних исследований, обработки тысяч электронных писем в Macromedia и откликов, полученных от пользователей всех уровней. Очевидно, книга носит отпечаток как моего страстного отношения к предмету, так и мучительного практического опыта, которым вы можете непосредственно воспользоваться. Она исчерпывающе авторитетно описывает ActionScript, а точность ее беспрецедентна благодаря технической проверке Гари Гроссмана (Gary Grossman), создателя ActionScript.

# Что может ActionScript?

Честно говоря, полноценные языки типа Flash 5 ActionScript не имеют практических границ применения. Взглянем на некоторые специфические возможности ActionScript, что позволит почувствовать, какого рода темы мы будем освещать на протяжении всей книги. Постарайтесь представить себе, как можно сочетать эти приемы для достижения ваших конкретных целей.

## Управление временной диаграммой

Фильмы Flash состоят из кадров, располагающихся в линейной последовательности, называемой *временной диаграммой* (*timeline*). С помощью ActionScript можно управлять воспроизведением временной диаграммы фильма, проигрывать отрезки фильма, выводить конкретный кадр, останавливать воспроизведение фильма, зацикливать анимации и синхронизировать анимационное содержание.

## 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 мышью (т. е. классические кнопки навигации)
- Содержимое, анимируемое движениями мыши (например, след от мыши)
- Объекты, которые можно перемещать с помощью мыши или клавиатуры (например, автомобиль в игре вождения)
- Текстовые поля, разрешающие пользователю 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 моделировать такие необычные эффекты, как движение по законам физики или обнаружение столкновений.

## Программная генерация содержимого

С помощью ActionScript можно генерировать визуальное и звуковое содержимое непосредственно из библиотеки фильма или путем копирования содержимого, уже имеющегося на рабочем столе. Содержимое, генерируемое программным образом, может служить строго статичным элементом, например, случайным узором, или интерактивным элементом, таким как вражеский космический корабль в видеоигре или опция в выпадающем меню.

## Обмен данными с сервером

ActionScript предоставляет большое разнообразие средств для отправки информации на сервер и получения ее с сервера. Во всех перечисленных ниже приложениях используется обмен данными с сервером:

- Ссылка на сайт
- Гостевая книга
- Приложение чата
- Сетевая игра с несколькими участниками
- Операция электронной торговли
- Персональный сайт с регистрацией и авторизацией пользователей

Конечно, эти примеры дают лишь ограниченное представление о возможных приложениях ActionScript. Задача книги в том, чтобы дать базовые навыки, которые позволят вам самостоятельно исследовать великое множество других возможностей. Это не сборник рецептов, это урок по приготовлению кода, а что будет в меню, решайте сами.

# Хранилище кода

В последующих главах встретятся десятки примеров кода. Чтобы получить соответствующие исходные файлы, а также многие другие обучающие файлы, не включенные в книгу, следует посетить сетевое хранилище кода (Code Depot), находящееся по адресу:

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

Хранилище кода – это развивающийся ресурс, содержащий практические приложения ActionScript и базовые коды. Вот список некоторых примеров, которые можно в нем найти (они скачиваются как по отдельности, так и единым zip-файлом):

- Вопросник с вариантами ответов
- Приложение для чата, использующее XML
- Приложение гостевой книги
- 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, как и на сайт данной книги, помещаются новые сведения о книге, уточнения, технические замечания и выявленные ошибки.

# Витрина

Практически в каждом имеющемся сайте, использующем Flash, есть хоть капелька ActionScript, а на некоторых сайтах даже довольно много. В следующей таблице представлен ряд адресов, на которых можно получить вдохновение для собственной работы. Посмотрите также сайты, перечисленные в приложении А «Ресурсы».

*Таблица. Витрина ActionScript*

| Тема | URL |
|---|---|
| Экспериментирование в дизайне, интерактивности и сценариях | *http://www.yugop.com*<br>*http://www.praystation.com**<br>*http://www.presstube.com*<br>*http://www.pitaru.com*<br>*http://www.flight404.com*<br>*http://www.bzort-12.com*<br>*http://kaluzhny.nm.ru/3D.html**<br>*http://www.protocol7.com**<br>*http://www.uncontrol.com** |

*(продолжение)*

| Тема | URL |
|---|---|
| | *http://www.digitalnotions.com/dev/flash5** |
| | *http://flash.onego.ru** |
| | *http://www.figleaf.com/development/flash5** |
| Игры | *http://www.gigablast.com* |
| | *http://www.sadisticboxing.com* |
| | *http://www.flashkit.com/arcade** |
| | *http://www.huihui.de* |
| Интерфейс и динамическое содержимое | *http://www.mnh.si.edu/africanvoices* |
| | *http://www.curiousmedia.com* |

* Имеются загружаемые файлы *.fla*. В остальных случаях доступны только файлы *.swf*.

# Типографские обозначения

Для выделения различных синтаксических составляющих ActionScript в книге использованы следующие обозначения:

- Моноширинный шрифт используется в примерах кода, именах экземпляров клипов, метках кадров, именах свойств и именах переменных.
- *Курсив* используется в названиях функций, методов, классов, слоев, именах и расширениях файлов, например *.swf*.
- **Моноширинный полужирный** шрифт используется для кода, который нужно вводить с клавиатуры при поэтапном выполнении процедуры.
- *Моноширинный курсив* используется для кода, который нужно заменить соответствующим значением (например, *здесь должно быть ваше имя*), или для имен переменных и свойств, упоминаемых в комментарии кода.
- За именами функций и методов следуют круглые скобки.

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

Это совет. В нем содержится полезная дополнительная информация по рассматриваемой теме.

Это предостережение, которое поможет вам решить или избежать неприятностей.

# Сообщите нам свое мнение

Информация, приведенная в данной книге, была протестирована и проверена со всей тщательностью, но вы можете обнаружить, что некоторые функции изменились (или даже найти ошибку!). Пожалуйста, пишите о найденных ошибках, а также ваши предложения для будущих изданий по адресу:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (in the U.S. or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

У этой книги есть сайт, на котором сообщается об ошибках, приводятся примеры и содержится различная дополнительная информация. Он находится по адресу:

*http://www.oreilly.com/catalog/actscript*

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 сайте:

*http://www.oreilly.com*

# Благодарности

Я глубоко обязан следующим людям, с которыми имел честь быть знакомым и совместно работать:

- Чрезвычайно талантливой команде Flash в Macromedia, находящейся в постоянном поиске нового и уважающей людей, сидящих за терминалами, которые и образуют Сеть. Сочетание профессионализма, исследовательского духа и личной увлеченности, свойственное Macromedia, редко встречается в структурах корпораций.
- Непревзойденным профессионалам в O'Reilly: Тиму O'Рейли (Tim O'Reilly), Трою Мотту (Troy Mott), Майку Сьерра (Mike Sierra), Робу Романо (Rob Romano), Эди Фридману (Edie Freedman) и многим редакторам, составителям указателей, корректорам, а также сотрудникам служб продаж и маркетинга, благодаря которым эта книга появилась на полках.
- Дереку Клейтону (Derek Clayton), моему личному наставнику в программировании и другу. Помимо почти каждодневного консультирования по коду Дерек написал код Perl для главы 17 «Фор-

мы Flash» и очень часто тренировал меня в Quake. Он также написал на Java сервер *XMLSocket* и на Perl – систему базы данных с плоскими файлами; то и другое можно загрузить из Хранилища кода.

- Уэнди Шаффер (Wendy Schaffer), которая, помимо корректирования чернового варианта рукописи, смогла поддерживать меня энергией и любовью во время всепоглощающего труда по написанию этой книги.
- Брюсу Эпштейну (Bruce Epstein), который выступал как развивающий редактор, помогая улучшить почти каждое предложение в рукописи и часто предлагая варианты завершения темы. Его проницательная редакторская работа и руководство неоценимы.
- Гари Гросману (Gary Grossman), создателю ActionScript в Macromedia, каким-то образом всегда находившему время для ответов на вопросы, разъяснения тонкостей и даже поддержки дискуссий во время разработки ActionScript. Гари выступал ведущим техническим редактором книги, помогая прояснить важные понятия и детали. Точность этого текста во многом является прямым результатом его участия.
- Славику Лозбену (Slavik Lozben), специалисту Macromedia Flash, которого я не могу в достаточной мере отблагодарить за создание событий в клипах и *swapDepths*. Без помощи интеллекта Славика и его готовности участвовать в обсуждении я бы до сих пор писал главу о событиях и их обработчиках. Славик внес также большой вклад в качестве технического редактора.
- Эрике Нортон (Erica Norton), специалисту по качеству Macromedia ActionScript, которая с готовностью отвечала на вопросы и исследовала их один за другим. Помимо ведения регулярного обсуждения Эрика выкраивала в своем напряженном графике работы время для того, чтобы осуществлять техническое редактирование.
- Джереми Кларку (Jeremy Clark), менеджеру продукта Macromedia Flash, который оказал активную поддержку книге своими мыслями, советами, дружбой и ответами на мои бесконечные вопросы. Эрику Витману (Eric Wittman), главному менеджеру продукта Macromedia Flash, чье прозорливое руководство на годы определило развитие Flash. Дженис Пирс (Janice Pearce) из группы контроля качества Macromedia Flash, которая разъяснила различные вопросы выпуска Flash и любезно предоставила первые версии Flash 5. Мэтту Вобенсмиту (Matt Wobensmith), менеджеру сообщества Macromedia Flash и постоянному ценному источнику информации. Трою Эвансу (Troy Evans), менеджеру продукта Macromedia Flash Player, который руководит Flash Player и борется за него. Бентли Вольфу (Bentley Wolfe) из группы технической поддержки Macromedia, который, кажется, никогда не расстается с клавиатурой.
- Ричарду Коману (Richard Koman) из O'Reilly & Associates, обеспечившему руководство редакторской работой на этапах заявки и первых черновых версий.

- Дэвиду Фугейту (David Fugate), моему литературному агенту из Waterside Productions, чье усердие и доверие сделали деловые вопросы практически незаметными.
- Д. Джо Дуонгу (D. Joe Duong), который знает слишком много для своего возраста, несмотря на то, что много времени посвящает обучению других. Мне повезло быть одним из них. Майку Линковичу (Mike Linkovich), философу программирования, который вдохновляет в такой же мере, как поучает. Джеймсу Портеру (James Porter) и Эндрю Мерфи (Andrew Murphy) за чтение и тестирование, снова чтение и тестирование и еще раз чтение и тестирование. И, конечно, Грэму Бартону (Graham Barton).
- Дугу Кили (Doug Keeley), Терри Маквайру (Terry Maguire) и Джону Николзу (Jon Nicholls), создавшим ICE – компанию, в которой позволено сосуществовать профессии и страсти.
- Профессору Полю Биму (Paul Beam), увидевшему связь между литературой, коммуникациями и компьютерами в невероятных рамках студенческой программы по английскому языку в Университете Ватерлоо.
- Профессору Джеку Грею (Jack Gray) за мудрость и дружбу.
- Эндрю Харрису (Andrew Harris), Дэвиду Люкстону (David Luxton), Майклу Каванагу (Michael Kavanagh), Стивену Бурку (Stephen Burke), Шерил Гула (Cheryl Gula), Кристине Нишино (Christine Nishino), Стивену Мумбу (Stephen Mumby), Карин Трговак (Karin Trgovac) и Юдит Зисман (Judith Zissman), которых я ценю за их искусство, идеи и дружбу.
- Сообществу Flash, из которого я черпал вдохновение и понимание, включая Джеймса Патерсона (James Patterson), Юго Накамура (Yugo Nakamura), Наоки Мицузе (Naoki Mitsuse), Джошуа Дэвису (Joshua Davis), Джеймсу Бейкеру (James Baker), Марселю Марсу (Marcell Mars), Филлипу Торроне (Phillip Torrone), Роберту Рейнхардту (Robert Reinhardt), Марку Феннелу (Mark Fennell), Брандену Холлу (Branden Hall), Джошу Ульму (Josh Ulm), Даррелу Планту (Darrel Plant), Тодду Пургасону (Todd Purgason), Джону Наку (John Nack), Джейсону Крогу (Jason Krogh), Хилману Куртису (Hillman Curtis), Гленну Томасу (Glenn Thomas) и всем остальным, кого я неизбежно пропустил.
- Семейству Мук (Moocks) – Маргарет, Майклу, Джейн и Биз, которые научили меня думать, мечтать, исследовать и любить.
- Семейству Шаффер (Schaffers), за годы семейной жизни и дружбы.

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


Колин Мук
Торонто, Канада
Апрель 2001

# I

# Основы ActionScript

В этой части рассказывается о базовом синтаксисе и грамматике языка ActionScript: переменных, данных, предложениях, функциях, обработчиках событий, массивах, объектах и клипах. По окончании изучения части I вы будете знать все необходимое 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 «Предложения»
- Глава 7 «Условные предложения»
- Глава 8 «Предложения цикла»
- Глава 9 «Функции»
- Глава 10 «События и обработчики событий»
- Глава 11 «Массивы»
- Глава 12 «Объекты и классы»
- Глава 13 «Клипы»
- Глава 14 «Лексическая структура»
- Глава 15 «Более сложные темы»

# 1

# Легкое введение для непрограммистов

Я буду учить вас разговаривать с Flash.

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

Языки программирования используются для отправки информации компьютерам и получения информации от них. Язык программирования, так же как и естественный язык, представляет собой совокупность словаря и грамматики, используемую для общения. Пользуясь языком 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 могли подумать, 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 бы было, если бы Flash мог разговаривать по-английски, т. е. для общения с ним не требовалось 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 нужно его поместить? В каком направлении он должен начать перемещение? 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 того, 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 в виде последовательности шагов:

1. Шарик – это символ круглого клипа с именем `ball`.
2. Квадрат – это символ клипа с четырьмя сторонами и именем `square`.
3. Создать новый зеленый шарик диаметром 50 пикселов.
4. Дать новому шарику имя `ball_one`.
5. Создать новый черный квадрат шириной 300 пикселов и поместить его в центр стола.
6. Поместить `ball_one` где-нибудь наверху 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 свои команды на естественном языке, нам все же пришлось пройти всю логику управления нашим прыгающим мячиком, чтобы Flash смог нас понять. Очевидно, программирование не ограничивается лишь знанием синтаксиса языков программирования. Аналогично этому, 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 формулировке логических шагов.
- Прежде чем сказать что-либо на языке компьютера, обычно полезно сказать это на человеческом языке.

- Разговор, ведущийся на одном языке, будучи переведен на другой язык, по-прежнему состоит из тех же основных утверждений.
- Компьютеры не очень догадливы. Кроме того, их словарь весьма ограничен.

По большей части программирование никак не связано с написанием кода. Прежде чем написать хоть одну строчку ActionScript, точно продумайте, что вы хотите сделать, и запишите функции своей системы в виде блок-схемы или проекта. Когда ваша программа станет достаточно хорошо описана на уровне понятий, можете перевести ее на ActionScript.

В программировании, так же как в любви, политике и бизнесе, эффективное общение является ключом к успеху. 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, нужно ссылаться только на тот мир, который ему известен, и использовать понятные ему термины. То, что может быть очевидным для вас, не всегда очевидно компьютеру. Программирование на компьютере можно уподобить разговору с ребенком: ничто не следует считать само собой разумеющимся, нужно быть точным в каждой детали и перечислять все шаги, необходимые для выполнения задачи. Но помните, что, в отличие от детей, Flash точно сделает то, что вы ему скажете, и ничего сверх того.

# Некоторые основные фразы

В первый же день занятий любым иностранным языком вы, скорее всего, выучите несколько основных фраз («Добрый день», «Как вы 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 использовать 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 (Действия). Все инструк-

ции, которые мы добавляем в панель Actions, выполняются Flash при воспроизведении фильма. Откройте теперь панель Actions с помощью следующих шагов:

1. Запустите Flash с новым пустым документом.
2. На главной временной диаграмме выберите кадр 1 слоя 1.
3. Выберите меню Window → Actions.

Панель Actions разделена на две части: окно Script (справа) и окно Toolbox (слева). В окне Script (Сценарии) размещается весь наш код. Окно Toolbox (Инструменты) предоставляет быстрый доступ к процедурам, операторам, функциям, свойствам и объектам (Actions, Operators, Functions, Properties и Objects) ActionScript. Вероятно, вы узнаете показанные на рис. 1.1 основные действия по прежним версиям Flash.

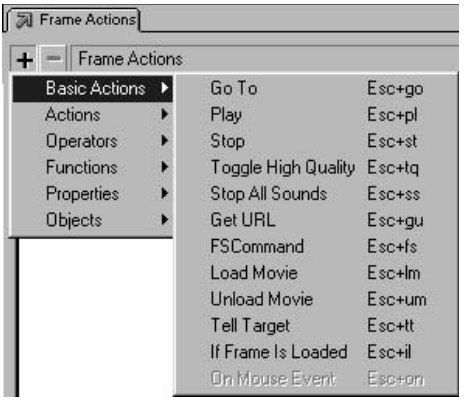


***Рис. 1.1.*** *Основные действия Flash 5*

Но в окне Toolbox можно обнаружить и многое другое. На рис. 1.2 отражены все имеющиеся действия, в том числе есть несколько старых друзей из Flash 2, 3 и 4. Продолжив изучение окна Toolbox, вы даже найдете такие вещи, как Sound (звук), Array (массив) и XML. В данной книге мы расскажем обо всем этом.

С помощью меню окна Toolbox можно создавать код ActionScript. Однако, чтобы изучить синтаксис, принципы и структурный состав ActionScript, мы будем вводить весь код вручную.

Так называемые *Actions (Действия)* являются не просто действиями: они содержат различные базовые средства языка программирования – переменные, условные операторы, циклы, комментарии, вызовы функций и т. д. Они смешаны в кучу в одном меню, и общее название *Actions (Действия)* затемняет значение программных структур.

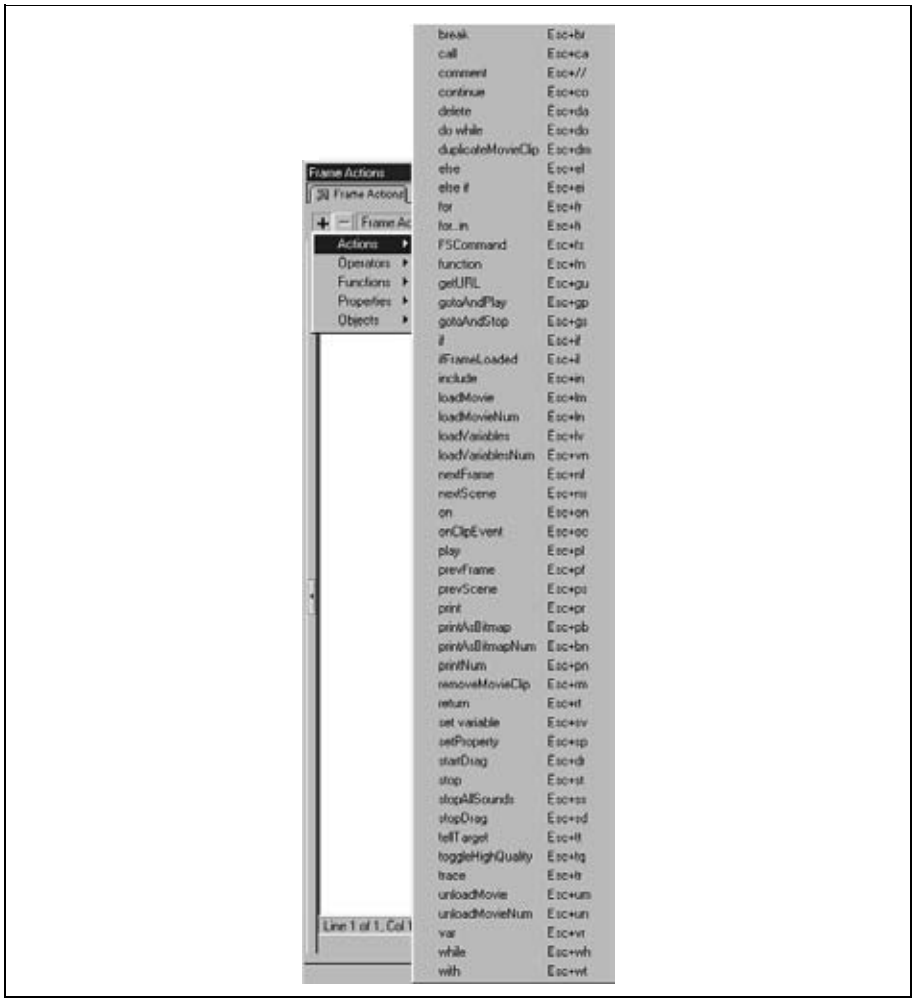


***Рис. 1.2.*** *Расширенный список действий*

Мы будем раскрывать смысл Actions, чтобы взглянуть на них с точки зрения программиста. По всей книге для описания каждого рассматриваемого действия я использую подходящий термин программирования. Например, вместо «добавить действие *while*» я буду писать «создать цикл *while*». Вместо слов «добавить действие *if*» я буду писать «создать новый условный оператор». 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, нужно отключить автопилот ActionScript следующим образом:

1. Выбрать меню Edit → Preferences.
2. На вкладке General выбрать Actions Panel → Mode → Expert Mode.
3. Expert Mode (Экспертный режим) можно выбрать и через всплывающее меню, доступное по стрелке на правом краю панели Actions, хотя при этом режим устанавливается только для текущего кадра. См. главу 16 «Среда разработки ActionScript».

Как вам это нравится? Вы уже эксперт. При входе в режим Expert Mode окно Parameters внизу на панели Actions исчезает. Пусть вас это не тревожит: мы не программируем с помощью меню, и поэтому оно нам не понадобится.

Теперь выберите кадр 1 в слое 1. Ваш ActionScript (называемый также *кодом*) всегда должен быть прикреплен к кадру, клипу или кнопке; выбор кадра 1 влечет прикрепление к нему создаваемого вслед за этим кода. 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 *предложением (statement)*. Под этой строкой введите вторую и третью строки кода, показанные после этого абзаца. Замените *здесь ваше имя* на свое имя (всюду в этой книге, где вы увидите *код, набранный курсивом*, нужно заменить эту часть кода собственным содержимым):

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

Хм. Пока ничего не произошло. Это связано с тем, что наш код не может ничего выполнить, пока мы не экспортируем файл *.swf* и не воспроизведем наш фильм. Прежде чем сделать это, давайте попросим Flash ответить на наше приветствие. Введите под уже имеющимися строками четвертую строку кода (ну, у нас тут полоса удачи ...):

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

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

Ловко, да? Давайте разберемся, как все это произошло.

***Рис. 1.3.*** *Flash становится дружелюбным*

## 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!";
```

на самом деле вы 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 дружественно, 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 словом `firstName`. Flash запомнил ваше имя и вывел его в окне Output, когда вы тестировали свой фильм.

Способность Flash кое-что запоминать является для нас решающим обстоятельством в программировании на нем. 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 позже.

### 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 кавычки для обозначения того, что это *строка* текста, а не число или другой *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. Формально существование переменной начинается с ее *объявления* с помощью особого ключевого слова *var*, которое вы использовали ранее.

Поэтому, для того чтобы проинструктировать вас, как создать первую строку кода, можно было воспользоваться более формальным языком: объявить новую переменную с именем `message` и присвоить ей начальное значение «Hi there, Flash!». Тогда вы должны были бы написать:

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

## Волшебник за шторой (интерпретатор)

Вспомните первые две строки кода:

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

В каждом из этих предложений 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.");
```

Эти предложения используют команду *trace( )*. Вы уже видели действие этой команды – она заставила Flash вывести ваш 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( )* заставила указанные данные появиться в окне Output (что удобно для слежения за происходящим во время выполнения программы).

Возникает вопрос: 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 инструкций, выполняет все команды ActionScript и выполняет предложения, запоминает ваши данные, посылает вам информацию, вычисляет значения и даже запускает основную среду программирования, когда фильм загружается в проигрыватель Flash.

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 попытаться 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. Когда интерпретатору не понятна команда, он посылает сообщение об ошибке. Интерпретатор, таким образом, действует как коммутатор для ActionScript: это аудитория, к которой вы обращаетесь в своем коде, и посредник, который возвращает вам сообщения из Flash.

Рассмотрим 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( )* используется для отображения текста в окне Output, поэтому он ждет указаний о том, какой текст выводить. Дальше он находит текст «Nice night to learn ActionScript.» между круглыми скобками, следующими за словом *trace*, и думает: «Ага! Это как раз то, что мне нужно. Я должен немедленно послать это в окно Output!».

Заметьте, что команда заканчивается точкой с запятой (;). Точка с запятой действует как точка в конце предложения. За редким исключением все предложения ActionScript должны заканчиваться точкой с запятой. Если предложение понято 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 того, как работают процессор компьютера и интерпретатор, но оно иллюстрирует следующие пункты:

- Интерпретатор всегда ждет ваши инструкции.
- Интерпретатор должен прочесть ваш код буква за буквой и постараться понять его. Это похоже на то, как человек старается прочесть и понять предложение в книге.
- Интерпретатор читает ваш ActionScript, применяя строгие правила: если бы, к примеру, отсутствовали круглые скобки в предложении *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 любовники: начнутся ссоры и крики «почему ты меня не слушаешь?!», и чудесные мгновения, когда вы понимаете друг друга в совершенстве. Мой отец всегда говорил мне, что лучший способ выучить новый язык – это найти возлюбленную, которая 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 обычно говорить «интерпретатор», а не «Flash».

## Дополнительные нужные 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 нужно заключить в круглые скобки:

```
command(argument);
```

Чтобы снабдить команду несколькими аргументами, их следует разделить запятыми:

```
command(argument1, argument2, argument3);
```

Посылка команде аргументов называется *передачей* аргументов. Например, в коде `gotoAndPlay(5)` именем команды является *gotoAndPlay*, а 5 является передаваемым аргументом (в данном случае, номером кадра). Некоторые команды, например *stop( )*, требуют после себя скобок, но не принимают аргументы. Почему так происходит, мы узнаем в главе 9 «Функции».

## Связующие элементы ActionScript (операторы)

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

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

Видите знаки плюс (+)? Они используются для соединения (*конкатенации*) нашего текста и являются одним из многих имеющихся *операторов*. Операторы в языке программирования сродни союзам («и», «или», «но» и т. д.) в естественных языках. Это средства, используемые 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 вместе, при этом обрабатывая их. Фразы могут быть текстом, числами или иметь другой тип данных, и оператор, как правило, производит некоторое преобразование. Очень часто операторы соединяют вместе два элемента, как это делает оператор «плюс». А другие операторы сравнивают значения, присваи-

вают значения, участвуют в принятии логических решений, определяют типы данных, создают новые объекты и предоставляют различные 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 чем* проверяет, какое из двух чисел является меньшим 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 типа данные хранятся в переменной. Поэтому, если создать переменную x и присвоить ей значение 4, 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 действовать, но при этом не используются 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 x, так похожий на аргумент, принято называть *операндом*. Операнд – это то, над чем действует оператор. Например, в выражении 4 + 9 числа 4 и 9 являются операндами оператора +.

В главе 5 «Операторы» подробно рассказано обо всех операторах ActionScript. Сейчас нужно просто запомнить, что операторы связывают фразы кода в некоторых преобразованиях.

## 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` является именем нашей переменной. Знак равенства является оператором, присваивающим текстовую строку («Hi there, Flash!») переменной `message`. Текст «Hi there, Flash!» становится в результате значением переменной `message`. Наконец, точка с запятой (`;`) сообщает интерпретатору, что мы завершили свое первое предложение.

Вторая строка почти такая же, как первая:

```
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 в третьей и четвертой строках мы используем переменные `message` и `firstName`:

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

Ключевое слово *trace* сообщает интерпретатору, что нужно вывести в окне Output некоторый текст. Текст, который нужно вывести, передается как аргумент. Открывающая круглая скобка обозначает начало аргумента. В четвертой строке в сам аргумент входят две *операции*, каждая из которых использует *оператор* «плюс». Первая операция присоединяет свой первый *операнд*, `"Hi there, "`, к значению второго операнда, `firstName`. Вторая операция присоединяет `", nice to meet you."` к результату первой. Закрывающая круглая скобка обозначает конец аргумента, а точка с запятой снова указывает на конец предложения.

Вот и готова ваша первая программа ActionScript! Это хорошее предзнаменование и важная веха.

# Другие понятия ActionScript

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

## Программы Flash

Для большинства пользователей компьютеров слово *программа* (*program*) является синонимом *приложения* (*application*), например Adobe Photoshop или Macromedia Dreamweaver. Очевидно, это не то, что мы создаем, программируя на Flash. С другой стороны, программисты определяют программу как совокупность строк кода («последовательность предложений»), 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 и кнопки. Мы прикрепляем свой код к этим элементам, чтобы он мог с ними взаимодействовать.

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, является не программой, а законченным фильмом (содержащим код, временную диаграмму, зрительные и звуковые элементы и прочие возможности).

Наши сценарии содержат большую часть того, что можно увидеть в обычных программах, но без всех этих вещей, что пишут на языках типа C++ или Java на уровне операционной системы, чтобы вывести на экран графику или воспроизвести звуки. Мы избавились от необходимости заниматься техническими деталями программирования графики и звука, что позволяет сосредоточиться на разработке поведения наших фильмов.

## Выражения

Предложения сценария, как мы выяснили, содержат инструкции сценария. Но инструкции обычно бесполезны при отсутствии данных. Когда мы, например, задаем 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 являются содержимым, которое мы обрабатываем в своем коде ActionScript. В своих сценариях вы будете извлекать, передавать, запоминать и вообще раскидывать всюду массу данных.

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

Литералы – лишь один из типов выражений. Переменная тоже может быть выражением (переменные считаются выражениями, поскольку замещаются данными). Выражения становятся еще более интересными, когда объединяются между собой с помощью операторов. Например, выражение 4 + 5 имеет два операнда, 4 и 5, но из-за оператора сложения все выражение принимает одиночное значение 9. Сложные выражения сами могут содержать другие, более короткие, если кодовая конструкция в целом может быть преобразована в одиночное значение.

Ниже показана переменная `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?"
```

что во время выполнения программы превратится в «Hi there, Flash! How are you?». При работе с арифметикой вы заметите, что 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 бы написать: «Чтобы присвоить значение переменной, введите ее имя, затем знак равенства и произвольное выражение».

## Два важных типа предложений: условные и циклические

Почти во всех программах используются *условные предложения* (*conditionals*) для внесения логики в программы и *циклы* (*loops*) для выполнения повторяющихся задач.

### Осуществление выбора с помощью условных предложений

Одной из действительно стоящих черт программирования Flash 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 эле-

ментарную логику мы используем при создании интерактивных фильмов Flash.

Вот несколько примеров логики в фильмах Flash:

- Пусть в нашем фильме есть 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 зоны с ограниченным доступом, 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 использования особого типа предложений, называемых *условными предложениями*. Условные предложения позволяют задать условия, при которых некоторый раздел кода должен (или не должен) выполняться. Вот пример условного предложения:

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

Общая структура условного предложения такова:

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

Подробнее о синтаксисе вы узнаете из главы 7 «Условные предложения». 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 делали за нас утомительные повторяющиеся задачи. Т. е. пока они не захватили весь мир и не поработили нас, начав выращивать в маленьких энергетических стручках, как … Стоп… забудьте то, что я сейчас сказал… Допустим, вы хотите вывести в окне Output последовательность из пяти чисел, начинающуюся с определенного числа. Если начальное число 10, можно вывести последовательность так:

```
trace (10);
trace (11);
trace (12);
```

```
trace (13);
trace (14);
```

Но если нужно начать последовательность с 513, то придется заново ввести все числа:

```
trace (513);
trace (514);
trace (515);
trace (516);
trace (517);
```

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 первой строке мы присваиваем переменной x значение 1. Затем во второй строке 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 повторяется еще три раза. В результате в окно Output оказывается выведена последовательность из пяти чисел. Прелесть в том, что если теперь нужно изменить начальное число последовательности, достаточно лишь изменить начальное значение x. Поскольку весь последующий код использует x, при выполнении программы меняется вся последовательность.

Это усовершенствование нашего первоначального подхода работает достаточно успешно, когда нужно вывести лишь пять чисел, но оказывается непригодным, если чисел становится 500. Для выполнения многократно повторяющихся задач используется *цикл* – предложение, заставляющее блок кода повторяться произвольное количество раз. Существует несколько типов циклов, каждый из которых имеет собственный синтаксис. Один из самых распространенных – цикл типа *while*. Вот как может выглядеть наш пример с использованием цикла *while* вместо ряда повторяющихся предложений:

```
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 предложения *trace (x);* и *x = x + 1;* выполняются при каждом повторении (или *итерации*) цикла. В данном случае цикл экономит нам только пять строк кода, но он мог бы сэкономить сотни строк, если бы мы считали до чисел более высокого порядка. И кроме того, наш цикл гибок. Чтобы сосчитать до 500, нужно лишь изменить выражение (*x <= 5*) на (*x <= 500*):

```
var x = 1;
while (x <= 500) {
  trace (x);
  x = x + 1;
}
```

Как и условные предложения, циклы являются одним из наиболее часто используемых и важных типов предложений в программировании.

## Модульный код (функции)

До сих пор ваш самый длинный сценарий состоял из четырех строчек кода. Но пройдет немного времени, и эти 4 строки превратятся в 400, а то и 4000 строк. Рано или поздно вы станете искать способы, как управлять кодом, сократить объем работы и облегчить применение вашего кода в различных сценариях. Тогда вы впервые действительно полюбите *функции*. Функция – это объединенная в один пакет последовательность предложений. На практике функции обычно являются многократно используемыми блоками 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 вам нужно 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;
```

Так как мы многократно повторяем расчет площади, лучше один раз поместить его в функцию и затем выполнять ее много раз:

```
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 функцию расчета площади с помощью предложения *function*, которое определяет (объявляет) функцию, подобно тому как слово *var* определяет переменную. Затем мы дали нашей функции имя `area`, подобно тому как мы даем имена переменным. Между круглыми скобками мы перечислили аргументы, которые получает наша функция при каждом своем использовании: `height` и `width`. А в фигурные скобки ({ }) заключены команды, которые должна выполнять наша функция:

```
return height * width;
```

Создав функцию, мы можем выполнять код, который она содержит, из любого места нашего фильма, указывая там 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 нам, и мы записываем их в переменные от `area1` до `area5`. Мило и точно, и здесь значительно меньше работы, чем в бесфункциональной версии кода.

Не волнуйтесь, если у вас возникли вопросы в связи с этим примером функции: мы узнаем о функциях больше в главе 9. 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 процедура *trace( )*. Вызов функции *area(4, 5);* очень

похож на подачу команды *trace( )*, например *trace (x);*. Это сходство не является просто совпадением. Как отмечалось выше, многие действия (Actions), в том числе *trace( )*, фактически являются функциями. Но это особый тип функций, которые встроены в ActionScript (в противоположность функциям, определяемым пользователем, таким как наша *area( )*). Поэтому допустимо (и технически более точно) говорить «вызвать функцию *gotoAndStop( )*», а не «выполнить Action *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 III «Справочник по языку». Мы также столкнемся со многими из них, изучая основы ActionScript.

## Экземпляры клипов фильмов

Я надеюсь, что за всеми этими разговорами об основах программирования вы не забыли об основах Flash. Одним из ключевых понятий визуального программирования на Flash являются *экземпляры* (*instances*) клипов. Как проектировщик или разработчик для Flash вы должны уже быть знакомы с клипами, но, возможно, не воспринимаете их как инструменты программирования.

Для каждого клипа в библиотеке фильма Flash имеется определение символа. Можно добавлять несколько копий, или *экземпляров*, одного символа клипа в фильм Flash, перетаскивая клип из библиотеки на рабочий стол. В значительной мере развитое программирование Flash заключается просто в управлении экземпляром клипа. Например, прыгающий мячик – это лишь экземпляр клипа, положение которого на столе изменяется периодически. С помощью ActionScript можно 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 Flash или справочные файлы.

## Модель выполнения, управляемого событиями

Последняя тема, которую мы должны рассмотреть в нашем обзоре основ ActionScript, – это модель выполнения, 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 совер-

шить 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, как пакетные программы. Они выполняются непрерывно (в *цикле событий, event loop*), ожидая, когда что-нибудь произойдет (*событие, event*), и выполняя фрагменты кода в ответ на эти события. В языках, предназначенных для использования в интерактивной визуальной среде (например, ActionScript или JavaScript), событиями обычно являются действия пользователя, такие как щелчок мышью или нажатие клавиши.

Когда происходит событие, например, щелчок мышью, интерпретатор подает сигнал тревоги. 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 (классическая навигация) или передаст переменные в базу данных (классическая подача формы).

Но программы будут реагировать на события только тогда, когда мы создадим *обработчики событий* (*event handlers*). Вот некий псевдокод, который в общем виде показывает, как задается обработчик события:

```
когда (произойдет это событие) {
  выполнить эти строки кода
}
```

Обычно это записывается в общей форме:

```
on (событие) {
  предложения
}
```

На практике обработчик события для кнопки, перемещающей воспроизводящую головку на кадр 200, будет выглядеть так:

```
on (press) {
  gotoAndStop(200);
}
```

Поскольку управляемые событиями программы всегда выполняют цикл событий в готовности реагировать на очередное событие, они подобны живым системам. События составляют решающую часть фильмов Flash. Без событий наши сценарии ничего бы не выполняли, за исключением одного: Flash выполняет любой код в кадре, когда воспроизводящая головка входит в этот кадр. Неявное событие состоит во

входе головки в конкретный кадр, и это настолько свойственно Flash, что не требует явного обработчика события.

События буквально являются движущей силой всего, поэтому они и появились в конце первого дня обучения в школе ActionScript. Вы теперь знаете, в чем состоит написание сценариев и что управляет временем выполнения этих сценариев (события). Можно сказать, 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 прикладное изучение программирования Flash с создания вопросника с вариантами ответов, используя очень простые приемы программирования, по большей части вам уже известные. В последующих главах мы будем возвращаться к нашему вопроснику, чтобы показать, как можно его усовершенствовать за счет более развитых концепций программирования, которые мы там 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. 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 демонстрирует практические приложения следующих аспектов программирования на ActionScript:

- Переменные
- Управление воспроизводящей головкой фильма с помощью функций
- Обработчики событий кнопок
- Простые условные выражения
- 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 варианта ответа. Пользователь вводит свои ответы, щелкая мышью по кнопкам, соответствующим выбранным вариантам. Результаты выбора, сделанного пользователем, записываются в переменную, чтобы получить результат пользователя. Когда даны ответы на все вопросы, подсчитывается число правильных ответов и выводится балл, набранный пользователем.

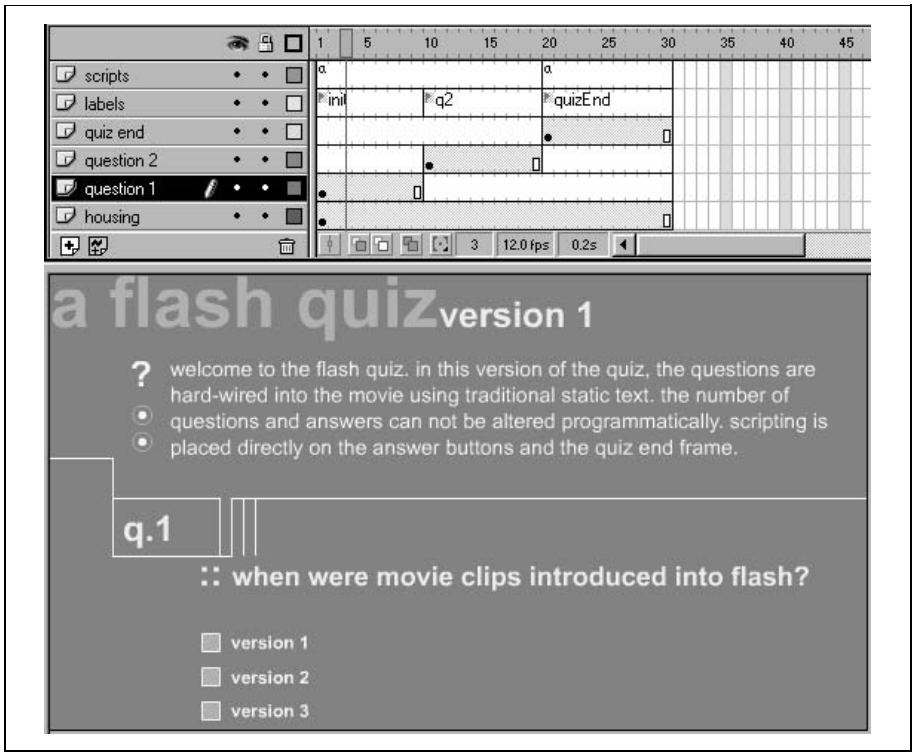


***Рис. 1.4.*** *Вопросник Flash*

## Создание структуры слоев

При создании фильмов Flash важно разделить содержимое на контролируемые части путем размещения различных его элементов на отдельных слоях. Размещение содержимого по слоям вообще является хорошим технологическим приемом, но для программирования Flash оно особенно существенно. 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 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, на которых мы распределим свои активы содержимого.

Начните построение вопросника с создания и именования следующих слоев, располагаемых в указанном порядке:

*scripts*

*labels*
*quiz end*
*question 2*
*question 1*
*housing*

Теперь добавьте по 30 кадров в каждый из этих слоев. Ваша временная диаграмма должна иметь вид, соответствующий рис. 1.5.

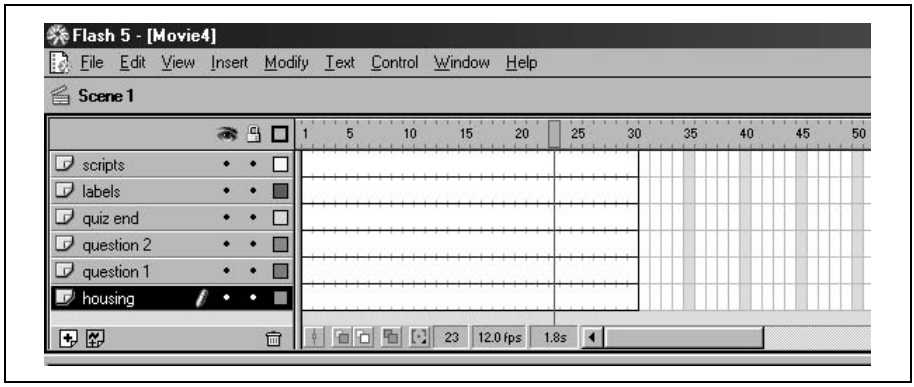


***Рис. 1.5.*** *Начальная настройка временной диаграммы вопросника*

## Создание интерфейса и вопросов

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

Вот шаги, которые нужно выполнить:

1. Выбрав кадр 1 в слое *housing*, с помощью инструмента Text введите название вопросника прямо на столе.
2. В кадре 1 слоя *question 1* добавьте номер вопроса «1» и текст вопроса: «When were movie clips introduced into Flash?» (Когда впервые во Flash появились клипы?). Оставьте ниже вопроса место для текста ответа и кнопок.
3. Создайте простую кнопку типа флажка или переключателя размером не выше строки текста (рис. 1.6).
4. Под текстом вопроса (по-прежнему в слое *question 1*) добавьте тексты трех ответов: «Version 1», «Version 2» и «Version 3», причем каждый на отдельной строке.
5. Рядом с каждым из этих трех ответов поместите экземпляр кнопки-флажка.

6. Question 1 будет использоваться в качестве шаблона для Question 2. Выделите первый кадр слоя *question 1* и выберите меню Edit → Copy Frames.
7. Выделите кадр 10 в слое *question 2* и выберите меню Edit → Paste Frames. В слое *question 2* в кадре 10 появится копия вашего вопроса 1.
8. 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 была введена поддержка аудиоформата MP3?). Ответы поменяйте на «Version 3», «Version 4» и «Version 5».
9. Наконец, чтобы под вопросом 2 не появился вопрос 1, добавьте пустой ключевой кадр в кадре 10 слоя *question 1*.

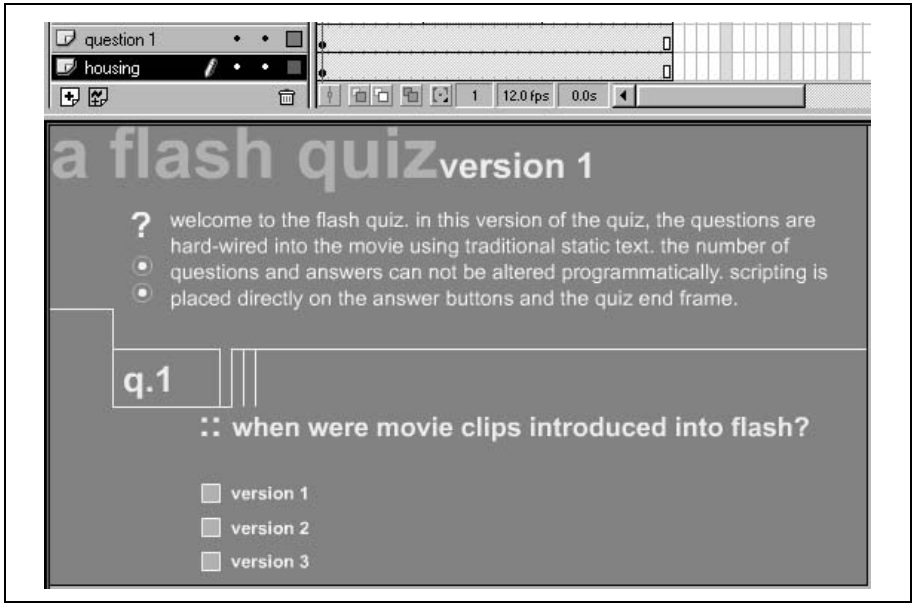


***Рис. 1.6.*** *Заголовок вопросника и Вопрос 1*

Рис. 1.6 демонстрирует фильм Flash после добавления в вопросник первого вопроса. На рис. 1.7 показано, как будет выглядеть временная диаграмма после помещения в вопросник двух вопросов.

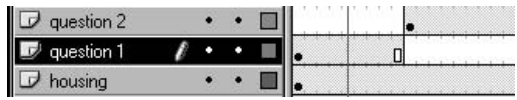


***Рис. 1.7.*** *Временная диаграмма вопросника с двумя вопросами*

## 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 нужно инициализировать свои переменные, прежде чем начнут 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 фильмах можно также задавать начальные условия путем вызова функций и присваивания значений переменным, осуществляя подготовку оставшейся части фильма. Этот этап называется *инициализацией*. Функции, запускающие в движение процессы или определяющие начальные условия действия системы, часто называют *инициирующими* (*init*).

Инициирующий код нашего вопросника, показанный в примере 1.1, прикреплен к кадру 1 слоя *scripts* нашего фильма.

*Пример 1.1. Инициирующий код вопросника*

```
// Инициировать переменные главной временной диаграммы
var q1answer;          // Ответ пользователя на вопрос 1
var q2answer;          // Ответ пользователя на вопрос 2
var totalCorrect = 0;  // Счетчик правильных ответов
var displayTotal;      // Текстовое поле для вывода набранных
                       // пользователем баллов

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

Строка 1 нашей инициирующей последовательности является *комментарием в коде*. Комментарии – это заметки, которые добавляются в код для пояснения производимых действий. Однострочный комментарий начинается с двух символов косой черты и пробела, за которыми следует строка текста:

```
// Это комментарий
```

Заметьте, что комментарии можно помещать в одной строке с кодом, например:

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

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

```
var favoriteColor;
```

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

- `q1answer` и `q2answer` будут содержать значения ответов пользователя (1, 2 или 3, указывая на номер варианта, выбранный для каждого вопроса). Эти значения будут использоваться для проверки правильности ответов пользователя.
- `totalCorrect` будет использоваться в конце опроса для подсчета количества правильных ответов.
- `displayTotal` – имя текстового поля, в котором будет выводиться на экран значение `totalCorrect`.

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` по умолчанию был 0 на случай, если 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.

После определения переменных мы вызываем функцию *stop( )*, останавливающую воспроизведение фильма на кадре 1, в котором начинается вопросник:

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

Функция *stop( )* оказывает такое же действие, как и операция *stop*, которой вы могли пользоваться во Flash 4 или более ранних версиях (она останавливает воспроизводящую головку в текущем кадре).

Посмотрите снова на использование комментария перед вызовом функции *stop( )*. Этот комментарий объясняет предназначение последующего кода. Комментарии не являются обязательными, но они помогают прояснить код, когда мы некоторое время не работаем с ним, а затем снова возвращаемся к нему и должны его вспомнить, либо если передаем код другому разработчику. Кроме того, комментарии облегчают просмотр кода, что важно во время отладки.

Теперь, когда вы разобрались с тем, что делает инициирующий код, добавим его в фильм вопросника:

1. Выделите кадр 1 слоя *scripts*.

2. Выберите меню Window → Actions. Появится панель Frame Actions.
3. Проверьте, что вы находитесь в режиме Expert Mode, который можно сделать постоянной настройкой с помощью меню Edit → Preferences.
4. В правой части панели Frame Actions введите инициирующий код, показанный выше в примере 1.1.

### Стиль именования переменных

Вы уже видели достаточно много имен переменных, и у вас мог возникнуть вопрос об использовании прописных букв. Если вы никогда ранее не занимались программированием, то прописная буква в середине слова, как в `firstName` или `totalCorrect`, может показаться вам необычной. Такое выделение первой буквы второго и последующих слов в имени переменной позволяет зрительно разделить слова, образующие имя. Этим приемом пользуются потому, что пробелы и дефисы использовать в именах переменных нельзя. Но не делайте прописной первую букву имени переменной: начальная прописная буква обычно используется в именах классов объектов, а не в именах переменных.

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

Всегда давайте переменным и функциям осмысленные имена, которые напоминают о том, для чего они используются. Избегайте бессмысленных имен типа «foo», а однобуквенные переменные типа «x» или «i» используйте только для простых вещей, таких как индекс (т. е. счетчик) в цикле.

## Добавление меток кадров

Мы сделали инициирующий сценарий для нашего вопросника и создали вопросы. Теперь нужно сделать метки для некоторых кадров, чтобы можно было управлять воспроизведением нашего вопросника.

Для того чтобы пользователь в каждый момент времени работал только с одним вопросом, мы разделили содержимое вопросов 1 и 2 на кадры 1 и 10. Перемещая воспроизводящую головку на эти ключевые кадры, мы вызываем показ слайдов, причем каждый слайд содержит свой вопрос. Мы знаем, что вопрос 2 находится в кадре 10, поэтому 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. Чтобы наш код мог работать даже в том случае, если кадры во временной диаграмме окажутся сдвинутыми, 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 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. Выберите меню Modify → Frame. Появится панель Frame.
3. В текстовом поле Label введите `init`.
4. К кадру 10 слоя *labels* добавьте пустой ключевой кадр.
5. В текстовом поле Label панели Frame введите `q2`.
6. К кадру 20 слоя *labels* добавьте пустой ключевой кадр.
7. В текстовом поле Label панели Frame введите `quizEnd`.

## Сценарии для кнопок ответов

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

Для удобства назовем кнопки выбора: кнопка 1, кнопка 2 и кнопка 3, как показано на рис. 1.8.

Сценарии для наших трех кнопок будут очень похожи между собой. В примерах 1.2–1.4 показан код для каждой кнопки.

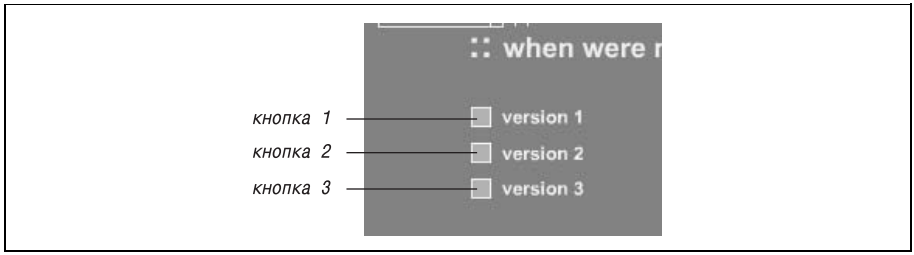


***Рис. 1.8.*** *Кнопки ответов*

*Пример 1.2. Код для кнопки 1 вопроса 1*

```
on (release) {
  q1answer = 1;
  gotoAndStop ("q2");
}
```

*Пример 1.3. Код для кнопки 2 вопроса 1*

```
on (release) {
  q1answer = 2;
  gotoAndStop ("q2");
}
```

*Пример 1.4. Код для кнопки 3 вопроса 1*

```
on (release) {
  q1answer = 3;
  gotoAndStop ("q2");
}
```

Код кнопки состоит из двух предложений (строки 2 и 3), которые выполняются только при обнаружении щелчка мышью. 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». Вот как это работает:

Строка 1 служит началом *обработчика события*:

```
on (release) {
```

Обработчик события терпеливо ожидает, пока пользователь не щелкнет по кнопке 1. Вспомните, что обработчик событий прислушивается, не произойдет ли что-нибудь (например, щелчок мышью) во время воспроизведения фильма. Когда происходит событие, выполняется код, содержащийся в соответствующем обработчике.

Разберем по частям обработчик события, который начинается в строке 1. Ключевое слово *on* сообщает о начале обработчика событий. (Если слово *on* вас смущает, воспринимайте его как *when*, пока не привыкните.) Ключевое слово *release*, заключенное в круглые скобки, указывает на

*тип события*, ожидаемого обработчиком. В данном случае мы ждем событие *release*, которое происходит, когда пользователь нажимает, а затем отпускает кнопку мыши над нашей кнопкой. Открывающая фигурная скобка ({) обозначает начало блока предложений, которые должны быть выполнены при возникновении события *release*. Конец блока кода помечен закрывающей фигурной скобкой (}) в строке 4, где и заканчивается обработчик события.

В строке 2 находится первое из предложений, которые будут выполнены, когда произойдет событие *release*. Код в строке 2 должен уже быть вам знаком:

```
q1answer = 1;
```

Он присваивает переменной `q1answer` значение 1 (другие кнопки ответа присваивают ей 2 или 3). Переменная `q1answer` запоминает ответ пользователя на первый вопрос. Запомнив ответ пользователя на первый вопрос, мы переходим ко второму вопросу с помощью кода, находящегося в строке 3:

```
gotoAndStop ("q2");
```

Строка 3 вызывает функцию *gotoAndStop( )*, передавая ей в качестве аргумента метку кадра «q2», в результате чего воспроизводящая головка перемещается в кадр `q2`, в котором оказывается вопрос 2.

Теперь, когда вы знаете, как работает код кнопки, давайте добавим его к кнопкам вопроса 1:

1. Открыв панель Actions, выберите на рабочем столе кнопку 1. Заголовок Frame Actions меняется на Object Actions. Любой код, который вы сейчас добавите, будет прикреплен к кнопке 1 (объекту, выделенному на рабочем столе).
2. В правой части панели Actions введите код примера 1.2.
3. Повторите шаги 1 и 2 для добавления кода к кнопкам 2 и 3. Для кнопки 2 присвойте `q1answer` значение 2; для кнопки 3 присвойте `q1answer` значение 3, как показано в примерах 1.3 и 1.4.

Код для кнопок вопроса 2 структурно идентичен коду для кнопок вопроса 1 (меняется только имя переменной ответа и адрес перехода в вызове *gotoAndStop( )*). Пример 1.5 показывает код для кнопки 1 вопроса 2.

*Пример 1.5. Код для кнопки 1 вопроса 2*

```
on (release) {
  q2answer = 1;
  gotoAndStop ("quizEnd");
}
```

Мы используем переменную `q2answer` вместо `q1answer`, поскольку хотим, чтобы кнопки отслеживали ответ пользователя на вопрос 2 . В качестве аргумента функции *gotoAndStop( )* указываем «quizEnd», что-

бы переместить воспроизводящую головку в конец вопросника (т. е. в кадр с меткой `quizEnd`) после ответа пользователя на вопрос 2.

Добавим код в кнопки вопроса 2:

1. Щелкните по кадру 10 слоя *question 2*.
2. Щелкните по кнопке 1.
3. В панели Actions введите код примера 1.5.
4. Повторите шаги 2 и 3, чтобы ввести код для кнопок 2 и 3. Для кнопки 2 присвойте `q2answer` значение 2; для кнопки 3 присвойте `q2answer` значение 3.

Добавив этот код для шести кнопок, вы, несомненно, обратите внимание на то, что в нем много повторений. Код одной кнопки отличается от кода другой лишь несколькими символами. Такое программирование не очень эффективно. Наш код для кнопок уже настоятельно требует наличия некоторого централизованного объекта, записывающего ответ и переходящего к следующему экрану вопросника. В главе 9 мы узнаем, как централизовать наш код с помощью *функций*.

## Создание концовки вопросника

Вопросник уже почти готов. У нас теперь есть два вопроса, действующих со сценарием, запоминающим ответы пользователя и позволяющим ему перемещаться по вопроснику. Но нам еще нужен заключительный экран с выводом результатов опроса.

Чтобы создать завершающий экран нашего вопросника, нужно еще немного поработать с основами Flash и сценарием. Сначала поработаем во Flash:

1. К кадру 20 слоя *question 2* добавим пустой ключевой кадр. Это помешает содержимому вопроса 2 появиться под содержимым завершающего экрана.
2. К кадру 20 слоя *quiz end* добавим пустой ключевой кадр.
3. Оставаясь в этом кадре, поместите на рабочий стол такой текст: «Thank you for taking the quiz. Your final score is: /2.» (Спасибо за участие в опросе; ваш итоговый результат: /2). Постарайтесь оставить достаточно места между «is:» и «/2.». Мы поместим туда набранные пользователем очки.
4. 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 выполнялся сценарий расче-

та. Так как любой сценарий, помещенный в ключевой кадр на временной диаграмме, автоматически выполняется при входе воспроизводящей головки в этот кадр, можно просто прикрепить наш сценарий расчета к ключевому кадру, который мы добавили в кадр 20 слоя *scripts*.

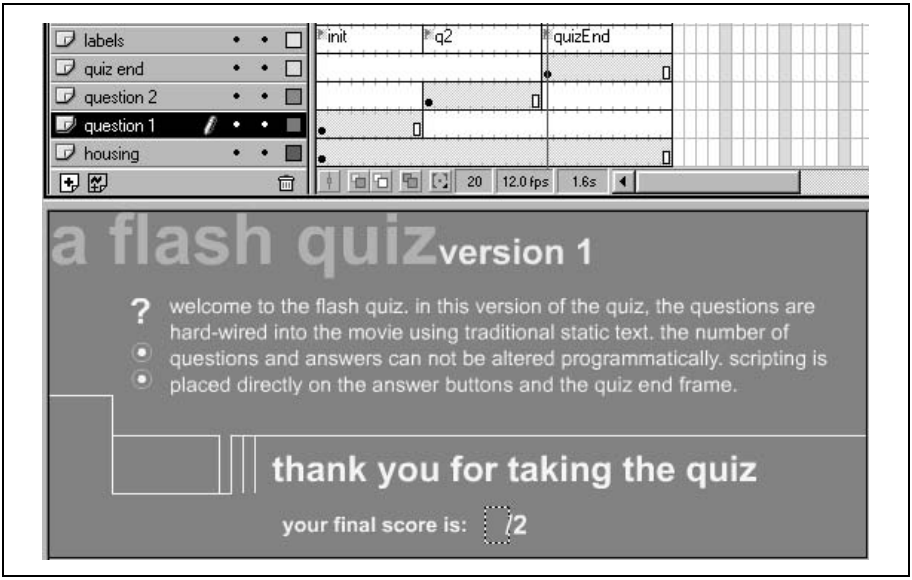


***Рис. 1.9.*** *Судный день*

В сценарии для расчетов сначала определим количество очков, набранных пользователем, а затем выведем его результат на экран:

```
// Подсчитаем правильные ответы пользователя
if (q1answer == 3){
  totalCorrect = totalCorrect + 1;
}
if (q2answer == 2){
  totalCorrect++;
}
// Покажем результат пользователя в текстовом поле на экране
displayTotal = totalCorrect;
```

Строки 1 и 8 являются комментариями, кратко описывающими назначение двух частей, имеющихся в сценарии. В строке 2 начинается первое из двух условных предложений в этом сценарии. В нем мы видим применение нашей переменной `q1answer`:

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

Ключевое слово *if* сообщает интерпретатору, что мы собираемся передать ему список предложений, которые должны быть выполнены,

только если удовлетворено некоторое условие. Суть этого условия помещена в круглые скобки за ключевым словом *if*: (*q1answer == 3*), а открывающая фигурная скобка начинает список условно выполняемых предложений. Поэтому строка 2 означает: «Если значение `q1answer` равно 3, выполнить предложения, 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 сохранен 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 предложения в фигурных скобках. Если нет, то условие не выполнено, и предложения в фигурных скобках пропускаются.

Flash не может знать о том, какие ответы на наши вопросы являются правильными. Проверка того, что `q1answer` равна 3, служит способом сообщить Flash, что нужно проверить правильность ответа пользователя на вопрос 1. Если ответ правильный, 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» (т. е. *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 мы завершаем блок операторов, которые нужно выполнить, если удовлетворено наше первое условие:

```
}
```

Строки с 5 по 7 являются другим условным предложением:

```
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 было только два вопроса, 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. Последнее, что осталось – это показать пользователю его результат, что делается в 9-й и последней строке нашего сценария завершения опроса:

```
displayTotal = totalCorrect;
```

Вы уже достаточно знаете о переменных, чтобы догадаться, что в строке 9 значение переменной `totalCorrect` присваивается другой переменной `displayTotal`. Но как при этом результат пользователя появляется на экране? А он пока и не появляется. Чтобы вывести его на экран, нужно создать переменную особого типа, называемую *текстовым полем*, которая имеет физическое представление на экране. Давайте создадим такую переменную и посмотрим, как она работает:

1. Выберите инструмент Text.
2. В слое *quiz end* щелкните по кадру 20.
3. Поместите указатель перед текстом «/2», который вы создали раньше, затем щелкните по рабочему столу.
4. Растяните текстовое окно так, чтобы его размер был достаточен для одного числа.
5. Выберите меню Text → Options.
6. В панели Text Options замените Static Text на Dynamic Text.
7. В текстовом поле Variable введите `displayTotal`.

У переменной `displayTotal` теперь есть экранное представление. Если в сценарии мы изменим значение `displayTotal`, соответствующее текстовое поле на экране будет обновлено.

## Тестирование нашего вопросника

Ну, вот и все. Наш вопросник завершен. Теперь вы можете проверить, работает ли он, с помощью меню Control → Test Movie. Пощелкайте по ответам в разных сочетаниях, чтобы убедиться в правильности подсчета очков. Можно даже создать кнопку перезапуска, прикрепив к новой кнопке следующий код:

```
on (release) {
  gotoAndStop("init");
}
```

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

Если вы обнаружите, что ваш вопросник не работает, попробуйте сравнить его с образцом вопросника, имеющимся в сетевом Хранилище кода. Может быть, вам также захочется изучить приемы устранения ошибок, описанные в главе 19 «Отладка».

# 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 вы могли заметить, в ActionScript нужно многое изучить, но и при минимуме знаний можно сделать довольно много. Даже то, что вам известно сейчас, предоставляет массу возможностей. В оставшейся части книги мы упрочим наши знания основ, изучая их более глубоко и показывая в сочетании с практическими примерами. Конечно, мы также обсудим некоторые темы, о которых еще и речи не шло.

Запомните главное: общение, сотрудничество и ясность мыслей. Если вы обнаружите, что делаете что-то чрезвычайно интересное, и вам хочется поделиться этим с другими, посылайте это мне на адрес *http://ww.moock.org/webdesign/flash/contact.html*.

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 необходимо 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 похоже на открытие 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.

Как только переменная создана, можно помещать в нее новые данные каждый раз, когда это требуется, – как деньги на счет. Или можно узнать, что находится в переменной, используя ее имя, – это напоминает проверку состояния счета. Если переменная больше не нужна, можно «закрыть счет», удалив переменную.

Главное, что нужно отметить, – это возможность обращения с помощью переменных к данным, которые изменяются или вычисляются 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 при условии изменения содержащихся в ней данных. Используя такую постоянную ссылку для доступа к переменному содержимому, можно осуществлять 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 рассмотрения того, как они создаются.

# Создание переменных (объявление)

Создание переменной называется ее *объявлением* (*declaration*). Объявление – это, пользуясь нашей банковской метафорой, этап «открытия счета», т. е. этап формального создания переменной. Когда впервые объявляется переменная, она пуста – чистая страница, ожидающая заполнения. В таком состоянии переменная содержит особое значение, называемое `undefined` (указывающее на отсутствие данных).

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

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

Слово *var* сообщает 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.

С помощью одного предложения *var* можно объявить несколько переменных, например:

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

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

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

## Автоматическое создание переменной

Многие языки программирования требуют, чтобы переменные были объявлены, прежде чем в них могут быть помещены данные; в противном случае возникает ошибка. ActionScript не так строг. Если мы присваиваем значение несуществующей переменной, интерпретатор создаст новую переменную сам. Продолжая аналогию, банк автоматически откроет вам счет при попытке сделать первый вклад.

Такое удобство обходится, однако, не бесплатно. Если мы не объявляем переменные сами, то не сможем сверить их с центральным переч-

нем при изучении кода. Более того, явное объявление переменной с помощью предложения *var* может иногда давать результаты, отличные от тех, которые получаются при *неявном* (т. е. автоматическом) объявлении переменной. Надежнее всего сначала объявить переменную, а затем ее использовать (т. е. *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 и символов подчеркивания. (Пробелы, дефисы и символы пунктуации не допустимы.)
- Должны начинаться с буквы или символа подчеркивания.
- Не должны превышать в длину 255 символов. (Ладно, это неправда, но пересмотрите свою систему именования переменных, если они превосходят в длину 255 символов.)
- Являются нечувствительными к регистру (верхний и нижний регистр воспринимаются одинаково, но, тем не менее, следует соблюдать единообразие).

Вот допустимые имена переменных:

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

А это 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
```

### Создание переменных с динамическими именами

Хотя вы едва ли будете часто (если вообще будете) использовать динамически создаваемые имена переменных, следует знать, что существует возможность программной генерации имен переменных. Для создания имени переменной из любого выражения используйте предложение *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, предпочтительнее использовать вместо переменных с динамическими именами, поскольку они предоставляют более мощные средства отслеживания данных с динамическими именами.

## Объявление переменных вначале

Хорошим стилем является объявление переменных в начале главной области сценариев каждого фильма, которая обычно начинается в первом ключевом кадре, следующим сразу за предзагрузчиком фильма. Непременно добавьте комментарии, объясняющие назначение каждой переменной, чтобы облегчить ее дальнейшую идентификацию. 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 присвоить им начальные значения, например:

```
var ballSpeed = 5;   // Скорость шарика, максимум 10
var score = 0;       // Текущий счет игрока
var hiScore = 0;     // Высший результат (не сохраняемый между сеансами)
```

# 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 наш счет». Для присваивания значения переменной запишем:

```
variableName = value;
```

где *variableName* – имя переменной, а *value* – данные, которые присваиваются этой переменной. Приведем пример применения:

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

Слева от знака равенства находится слово `bookTitle`, являющееся *именем* переменной (ее *идентификатором*). Справа от знака равенства – фраза «ActionScript: The Definitive Guide», представляющая собой *значение переменной* (т. е. депонируемые данные). Сам знак равенства называется оператором *присваивания* (*assignment*). Он сообщает Flash, что то, что находится справа от знака равенства (т. е. вклад), вы хотите присвоить переменной, указанной слева. Если переменная слева еще не существует, Flash создаст ее (хотя использовать интерпретатор для неявного создания переменных не рекомендуется).

Вот еще два примера присваивания значений переменным:

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

В первом примере переменной speed присваивается целое число 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 двойные кавычки (" ").

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

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

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

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

И снова вычисляется выражение справа от знака равенства, и 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 (чисел, текста или данных любого другого типа) переменной одинаков для всех типов данных. Например, мы еще не изучали массивы, но вы должны уже определить, что следующий код является предложением присваивания значения переменной:

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

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

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. Предыдущее предложение присваивает z значение 10, затем присваивает y значение z и, наконец, присваивает x значение y.

# Изменение и извлечение значений переменных

После того как переменная создана, 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 нужного 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 эти значения нельзя извлечь позже. Чтобы извлечь значение переменной, просто укажите имя переменной там, где вы хотите использовать ее значение. Всюду, где появляется имя переменной (за исключением объявления или левой части оператора присваивания), оно заменяется ее значением. Вот несколько примеров:

```
newX = oldX + 5;  // Присвоить newX значение oldX плюс 5
ball._x = newX;   // Присвоить newX горизонтальное положение клипа ball
trace(firstName); // Вывести значение firstName в окне Output
```

Обратите внимание, что `ball` в выражении `ball._x` является именем клипа, а `._x` задает свойство его x-координаты (т. е. горизонтального положения на рабочем столе). Подробнее о свойствах мы узнаем позже. Последняя строка, `trace(firstName)`, выводит значение переменной 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 ее значение с ключевым словом `undefined`. Например:

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

Обратите внимание на использование *оператора неравенства* `!=`, который определяет, что два значения *не* равны между собой.

# Типы значений

Используемые в программировании ActionScript данные могут иметь разные типы. До сих пор мы видели числа и текст, а другие типы включают булевы значения, массивы, функции и объекты. Прежде чем подробно рассказывать о каждом типе данных, обсудим некоторые аспекты типов данных, имеющие 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. Такие языки, как C++ и Java, используют *типизированные* переменные; каждая переменная может принимать данные только одного типа, который должен быть указан при объявлении переменной. Переменные ActionScript получают тип *автоматически* – интерпретатор назначает тип переменной при присваивании ей данных.

Переменные ActionScript не только могут содержать данные любого типа, но также могут динамически *изменять* типы данных. Когда переменной присваивается значение иного типа, чем предыдущеее ее значение, тип переменной автоматически изменяется. Поэтому в ActionScript допустим такой код:

```
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 таких языках, как C++ или Java, которые не поддерживают автоматическое изменение типа, данные неверного типа должны быть преобразованы в тому типу, который имеется у переменной (либо невозможность преобразования приводит к ошибке). Автоматическая и динамическая типизации имеют некоторые важные разновидности, которые мы рассмотрим в следующих разделах.

## Автоматическое преобразование значения

В некоторых контекстах ActionScript ожидает данных конкретного типа. Если мы используем переменную, чье значение имеет тип, отличающийся от ожидаемого, интерпретатор пытается преобразовать данные. 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 переменной `z` присваивается значение 2. Почему? Поскольку оператор вычитания ждет число, значение `y` преобразуется

из строки «4» в число 4, которое вычитается из 6 (значения x), что и дает в результате 2.

*Пример 2.2. Автоматическое преобразование строки в число*

```
x = 6;       // x - число 6
y = "4";     // y является строкой "4"
z = x - y;   // Присваивает z значение 2
```

Наоборот, если числовая переменная используется там, где ожидается строка, то интерпретатор пытается преобразовать число в строку. В примере 2.3 переменная z получает в качестве значения строку «64», а не число 10. Почему? Потому что второй операнд в выражении *x + y* является строкой. Поэтому (+) 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;   // Присваивает z строку "64"
```

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

Обратите внимание, что оператор в строке 3 («-» в примере 2.2 и «+» в примере 2.3) существенно влияет на значение, присваиваемое z. В примере 2.2 строка «4» становится числом 4, в то время как в примере 2.3 происходит противоположное (число 6 становится строкой «6»), поскольку правила преобразования типов различны для операторов «+» и «-». Мы рассмотрим правила преобразования данных в главе 3 «Данные и типы данных», а операторы – в главе 5 «Операторы».

## Определение типа данных вручную

Автоматическая типизация и преобразование, возможно, и удобны, но, как показывают примеры 2.2 и 2.3, могут 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 переменной, можно воспользоваться условными предложениями. 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");  // Вывести сообщение об ошибке
}
```

Полное описание оператора *typeof* смотрите в главе 5.

# Область видимости переменной

Ранее мы узнали, как создавать переменные и извлекать их значения при помощи переменных, прикрепленные к отдельному кадру главной временной диаграммы документа Flash. Если документ содержит несколько кадров и несколько временных диаграмм клипов, создание переменных и извлечение их значений становится несколько более сложным.

Чтобы проиллюстрировать причины, рассмотрим несколько сценариев.

## Сценарий 1

Предположим, нам нужно создать переменную x в кадре 1 главной временной диаграммы. После создания переменной x мы присваиваем ей значение 10:

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

Затем в следующем кадре (кадр 2) мы прикрепляем такой код:

```
trace(x);
```

Если мы проиграем наш 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.

При определении переменной на временной диаграмме она доступна из всех других кадров этой временной диаграммы.

## Сценарий 2

Допустим, что мы создаем переменную x и присваиваем ей значение, как в сценарии 1, но код для присваивания помещаем не непосредственно в кадр 1, а на кнопку в кадре 1. Затем к кадру 2 мы прикрепляем тот же код, что и раньше:

```
trace(x);
```

Будет ли сценарий 2 тоже работать? Да. Поскольку переменная x прикреплена к нашей кнопке, а кнопка прикреплена к главной временной диаграмме, то наша переменная косвенно прикреплена к главной временной диаграмме. Поэтому мы можем обращаться к переменной из кадра 2, как и раньше.

### Сценарий 3

Предположим, что мы создаем переменную с именем secretPassword в кадре 1 главной временной диаграммы. При воспроизведении фильма пользователь должен угадать пароль, чтобы получить доступ к особой части фильма.

Помимо объявления secretPassword в кадре 1, мы создаем функцию, которая сравнивает настоящий пароль с введенным пользователем. Вот наш код:

```
var secretPassword;
secretPassword = "yppah";

function checkPassword() {
  if (userPassword == secretPassword) {
    gotoAndStop("accessGranted");
  } else {
    gotoAndStop("accessDenied");
  }
}
```

Допустим, что мы просим пользователя ввести свой пароль в кадре 30. Он вводит свой пароль в переменную текстового поля ввода с именем userPassword, которую мы сравниваем со своей переменной secretPassword с помощью функции *checkPassword( )* в кадре 1. Если наш код проверки пароля определен в кадре 1, а переменная userPassword определена только в кадре 30, будет ли userPassword существовать, когда мы вызываем свою функцию *checkPassword( )*?

Ответ опять положительный. Хотя переменная userPassword определена в более позднем кадре, чем функция *checkPassword( )*, она является частью той же временной диаграммы.

Любая переменная, объявленная на временной диаграмме, доступна всем сценариям своей временной диаграммы до тех пор, пока существует эта временная диаграмма.

## Доступность переменных (область видимости)

Представленные выше три сценария исследуют проблемы *области видимости* (*scope*). Область видимости переменной описывает, когда и где эта переменная может обрабатываться кодом в фильме. Область

видимости переменной определяет продолжительность ее существования и доступность из других блоков кода в наших сценариях. Чтобы определить область видимости переменной, мы должны ответить на два вопроса: a) в течение какого времени существует переменная? и b) в каких частях нашего кода значения переменной могут присваиваться или извлекаться?

В традиционном программировании переменные часто делят на две общие категории видимости: *глобальные* и *локальные*. Переменные, которые доступны на протяжении всей программы, называются *глобальными переменными*. Переменные, доступные только в ограниченных разделах программы, называются *локальными переменными*. Хотя Flash поддерживает обычные локальные переменные, он не поддерживает подлинно глобальные переменные. Давайте выясним, почему.

## Переменные клипов

Как мы видели в трех предшествующих сценариях, переменная, определенная на временной диаграмме, доступна во всех сценариях на этой временной диаграмме (от первого до последнего кадра) независимо от того, объявлена ли она в кадре или в кнопке. А если в фильме не одна временная диаграмма, как в сценарии 4, что тогда?

### Сценарий 4

Предположим, есть две основные геометрические фигуры – квадрат и круг, определенные как символы клипа.

В кадре 1 символа клипа квадрата присвоим переменной `x` значение 3:

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

В кадре 1 символа клипа круга присвоим переменной `y` значение 4:

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

Экземпляры этих клипов мы поместим в кадр 1 слоя 1 главной временной диаграммы нашего фильма 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? Вот этот код:

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

Ответ: Ничего в окне Output не появится. Переменные `x` и `y` определены во временных диаграммах наших клипов, а *не* в главной временной диаграмме.

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

Второй вопрос: Если бы мы поместили предложения `trace(x)` и `trace(y)` в кадр 1 нашего клипа `square`, а не в кадр 1 главной временной диаграммы, что появилось бы в окне Output?

Ответ: Значение `x`, которое равно 3, и ничего больше. Значение `x` отображается, поскольку `x` определена на временной диаграмме клипа `square` и поэтому доступна команде *trace( )*, которая также расположена на этой временной диаграмме. Но значение `y`, равное 4, не появляется в окне Output, так как `y` определена в клипе `circle`, а это отдельная временная диаграмма.

Теперь вам должно быть понятно, почему я сказал, что ActionScript не поддерживает подлинно глобальные переменные. Глобальные переменные – это переменные, которые доступны на протяжении всей программы, но во Flash переменная, прикрепленная к отдельной временной диаграмме, непосредственно доступна только сценариям на той же временной диаграмме. Так как все переменные Flash определяются на временных диаграммах, ни для одной переменной нельзя гарантировать, что она будет доступна во *всех* сценариях фильма. Поэтому ни одну переменную нельзя на законных основаниях назвать *глобальной*.

Чтобы избежать путаницы, будем называть переменные, прикрепленные к временным диаграммам, *переменными временной диаграммы* или *переменными клипа*. Однако глобальные переменные можно моделировать с помощью класса *Object*. Для создания переменной, доступной на всех временных диаграммах, используйте следующее предложение:

```
Object.prototype.myGlobalVariable = myValue;
```

Например:

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

Этот прием (и причины, по которым он работает) обсуждаются в разделе «Окончание цепочки наследования» главы 12 «Объекты и классы».

## Доступ к переменным других временных диаграмм

Несмотря на то что переменные на одной временной диаграмме не доступны непосредственно сценариям других временных диаграмм, доступ к ним может быть организован косвенным образом. Для созда-

ния, извлечения и присвоения значений переменным другой временной диаграммы можно использовать *синтаксис с точкой* – стандартное понятие, часто использующееся в объектно-ориентированных языках программирования, 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 мы обращаемся к переменной на другой временной диаграмме, используя имя 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` следующим образом:

```
square.x
```

А обратиться из главной временной диаграммы к переменной `y` в клипе `circle` можно так:

```
circle.y
```

Можно использовать эти ссылки из главной временной диаграммы фильма для присваивания и извлечения значений переменных в `square`:

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

Однако одного синтаксиса *clip.variable* не достаточно для ссылки на переменные в `square` из клипа `circle`. Если бы мы поместили ссылку `square.x` в кадр клипа `circle`, то интерпретатор стал бы искать клип с именем `square` ***внутри*** `circle`, в то время как `square` находится на главной временной диаграмме. Поэтому необходим механизм, который позволяет 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 всегда обратиться 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, поскольку она начинается на главной временной диаграмме фильма, _root. Вот еще один пример вложенности, показывающий обращение к переменной area экземпляра triangle, находящегося на временной диаграмме экземпляра shapes:

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

Любая ссылка на переменную, начинающаяся с ключевого слова _root, называется *абсолютной ссылкой*, потому что она описывает положение переменной относительно фиксированной, неизменной точки фильма – главной временной диаграммы.

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

```
_parent.myVariable
```

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

Возвращаясь к более раннему примеру, 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.

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);
```

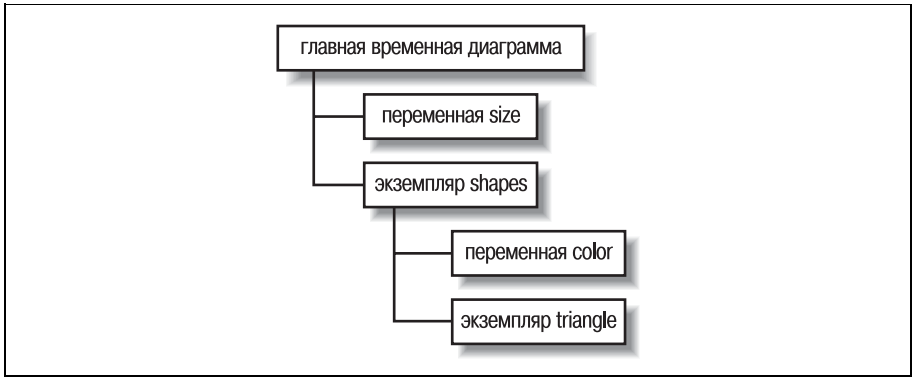


***Рис. 2.1.** Пример иерархии клипа*

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

Свойство `_parent` можно использовать два раза подряд, чтобы подняться по иерархии клипов и обратиться к переменной `size` на главной временной диаграмме. Сейчас мы прикрепим к клипу `triangle` некоторый код, который ссылается на переменную `size`, находящуюся на главной временной диаграмме:

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

Используя свойство `_parent` дважды, мы поднимаемся вверх на два уровня, что в данном контексте доставляет нас на главную временную диаграмму фильма.

Выбор подхода к адресации переменных зависит от того, что мы хотим достичь при размещении экземпляров символа клипа на разных временных диаграммах. Если в нашем примере с `triangle` требуется, чтобы ссылка на переменную `color` всегда указывала на ту переменную `color`, которая определена в клипе `shapes`, то следует использовать синтаксис с `_root`, дающий фиксированную ссылку на `color` в `shapes`. Но если нужно, чтобы ссылка на `color` указывала на разные переменные `color` в зависимости от того, на какой временной диаграмме лежит данный экземпляр `triangle`, то следует использовать синтаксис `_parent`.

## Доступ к переменным на различных уровнях документа

Свойство `_root` ссылается на главную временную диаграмму фильма на текущем уровне (т. е. в текущем документе), но проигрыватель Flash

может разместить в своем *стеке документов* несколько документов. На главную временную диаграмму любого фильма, загруженного в стек документов проигрывателя, можно ссылаться с помощью _level*n*, где *n* является номером уровня, на котором расположен фильм. Номера уровней начинаются с 0, например _level0, level1, _level2, _level3 и т. д. Информацию о загрузке нескольких фильмов можно найти в главе 13 «Клипы». 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 экземпляров клипов с использованием синтаксиса с точкой следите за тем, что вы даете имена экземплярам клипов на рабочем столе и правильно вводите их имена при ссылке в коде. Если экземпляры безымянные, ваш код не будет работать даже при синтаксической корректности в остальных отношениях. Неименованные экземпляры и ошибки при вводе имени очень часто являются источником проблем.

## Сравнение синтаксиса обращения к переменным во Flash 4 и Flash 5

Конструкции в стиле Flash 4, использующие косую черту и двоеточие, например /square:area, заменены во Flash 5 синтаксисом с точкой – значительно более удобным способом ссылки на переменные и временные диаграммы. Прежний синтаксис устарел и больше не рекомендуется к использованию. В табл. 2.1 показаны эквивалентные формы обращения к переменным во Flash 4 и Flash 5. О других синтаксических различиях читайте в приложении C «Обратная совместимость».

*Таблица 2.1. Синтаксис обращения к переменным во Flash 4 и Flash 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 |

*Таблица 2.1 (продолжение)*

| Синтаксис Flash 4 | Синтаксис Flash 5 | Ссылается на ... |
|---|---|---|
| `../` | `_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, которая содержит клип, содержащий текущий клип (на два уровня выше временной диаграммы текущего клипа) |
| `clip1:x` | `clip1.x` | Переменную `x` в экземпляре `clip1`, где `clip1` расположен на текущей временной диаграмме |
| `clip1/clip2:x` | `clip1.clip2.x` | Переменную `x` в экземпляре `clip2`, где `clip2` расположен внутри `clip1`, который, в свою очередь, находится на 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 фильма, загруженного в уровень 2 |

* «Временная диаграмма текущего клипа» – та временная диаграмма, которая содержит ссылку на переменную.

## Время жизни переменной клипа

Ранее мы говорили, что область видимости переменной отвечает на два вопроса: a) в течение какого времени существует переменная? и b) в каких частях нашего кода значения переменной могут извлекаться и присваиваться? Относительно переменных клипов нам теперь известны факторы, определяющие ответ на второй вопрос. Но мы еще не ответили на первый вопрос. Давайте вернемся к нему сейчас, рассмотрев еще один сценарий использования переменных.

### Сценарий 5

Допустим, мы создаем новый фильм с двумя ключевыми кадрами. В кадр 1 мы поместим экземпляр клипа `ball`. На временной диаграмме

ball создадим переменную radius. Кадр 2 нашей главной временной диаграммы пуст (в нем нет экземпляра ball).

Из кадра 1 главной временной диаграммы фильма мы можем получить значение radius с помощью следующего кода:

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

Теперь вопрос звучит так: Если переместить эту строку кода из кадра 1 в кадр 2 главной временной диаграммы, что появится в окне Output при воспроизведении фильма?

Ответ: Ничего не появится. Когда клип ball перемещается из главной временной диаграммы в кадр 2, все его переменные уничтожаются.

Переменные клипа существуют, пока клип, в котором они находятся, присутствует на рабочем столе. Переменные, определенные на главной временной диаграмме документа Flash, сохраняются в каждом документе, но теряются, как только документ выгружается из проигрывателя (с помощью функции *unloadMovie( )* или при загрузке нового фильма на уровень этого фильма).

Срок жизни переменных важен при создании сценариев фильмов, содержащих клипы, распределенные по нескольким кадрам на различных временных диаграммах. Всегда, прежде чем использовать переменные клипа, проверяйте наличие этого клипа на временной диаграмме.

## Локальные переменные

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

Локальные переменные используются в функциях и старых подпрограммах в стиле Flash 4. Если вы не работали ранее с функциями или подпрограммами, можете пропустить этот раздел и вернуться к нему после чтения главы 9 «Функции».

Функции часто используют переменные, которые не нужны вне их. Допустим, например, что у нас есть функция, отображающая все элементы указанного массива:

```
function displayElements(theArray) {
   var counter = 0;
   while(counter < theArray.length) {
```

```
        trace("Element " + counter + ": " + theArray[counter]);
        counter++;
    }
}
```

Переменная `counter` нужна для вывода массива, но после этого не используется. Можно было бы оставить ее определение на временной диаграмме, но это нехорошо по двум причинам: a) если `counter` продолжает существовать, она занимает память все оставшееся время нашего фильма; b) если `counter` доступна вне нашей функции, она может вступить в конфликт с другими переменными с именем `counter`. Поэтому было бы 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 удаляются 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: будучи помещено *вне* функции, предложение *var* создает обычную переменную временной диаграммы, а не локальную переменную. Как показано в примере 2.4, расположение предложения *var* оказывает решающее влияние.

Переменные внутри функции не обязательно должны быть локальными. Можно создавать или изменять переменные клипа внутри функции, опуская ключевое слово *var*. Если мы не используем ключевое слово *var*, а вместо этого просто присваиваем значение переменной, Flash в некоторых случаях рассматривает эту переменную как нелокальную. Рассмотрим такое присваивание переменной внутри функции:

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

Результат выполнения предложения `height = 10;` зависит от того, является ли `height` локальной переменной или переменной клипа. Если `height` была ранее объявлена как локальная переменная (что не соответствует нашему случаю), то предложение `height = 10;` просто модифицирует значение локальной переменной. Если локальной переменной с именем `height` нет, как в данном случае, то интерпретатор создает переменную клипа (нелокальную переменную) с именем `height` и присваивает ей значение 10. Не будучи локальной переменной, `height` продолжает существовать и после завершения работы функции.

Пример 2.4 демонстрирует использование 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
trace(x);          // Выводит: 10 (присваивание в нашей функции изменило
                   // первоначальное значение переменной)
trace(y);          // Выводит: 20 (нелокальная переменная y
                   // продолжает существовать)
trace(z);          // Ничего не выводит (локальная переменная z
                   // перестала существовать)
```

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 переменная с одинаковыми именами*

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

## Локальные переменные в подпрограммах

Хотя функции служат предпочтительным механизмом создания переносимых модулей кода, Flash 5 продолжает поддержку *подпрограмм* в стиле Flash 4. Во Flash 4 подпрограмму можно было создать, прикрепив блок кода к кадру с меткой. Впоследствии подпрограмма могла быть выполнена удаленно посредством действия *Call*. Но во Flash 4 любая переменная, объявленная в подпрограмме, не была локальной и и сохранялась, пока существовала временная диаграмма, в которой она была определена. Во Flash 5 вы можете создавать локальные переменные в подпрограммах тем же способом, каким мы создавали их в функциях, т. е. используя предложение *var*. Однако переменные, определенные в подпрограмме через *var*, создаются как локальные переменные только тогда, когда подпрограмма выполняется посредством

функции *Call*. Если сценарий в кадре подпрограммы выполняется в результате перемещения в кадр воспроизводящей головки, предложение *var* объявляет обычную нелокальную переменную временной диаграммы. Тем не менее, вместо подпрограмм следует использовать более современные функции и локальные переменные функций.

# Некоторые практические примеры

Мы очень много времени уделили теории переменных. Как насчет того, чтобы посмотреть некоторые из этих понятий в действии? В следующих примерах приводятся три варианта использования переменных. Объяснение кода см. в комментариях.

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

*Пример 2.6. Переместить воспроизводящую головку в случайный кадр текущей временной диаграммы*

```
var randomFrame;            // Хранит случайно выбранный номер кадра
var numFrames;              // Хранит общее количество кадров во временной
                            // диаграмме
numFrames = _totalframes;   // Присвоить свойство _totalframes
                            // переменной numFrames

// Выбрать случайный кадр
randomFrame = Math.floor(Math.random() * numFrames + 1);

gotoAndStop(randomFrame);   // Переместить воспроизводящую головку
                            // в случайно выбранный кадр
```

В примере 2.7 определяется расстояние между двумя клипами. Рабочий вариант этого примера можно получить из 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 квадратным корнем из выражения:
// (deltaX в квадрате плюс deltaY в квадрате).
dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));
```

```
// Аккуратные ссылки значительно легче читать, чем те,
// которые таковыми не являются:
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 переводятся в значения по шкале Цельсия и наоборот. Рабочий вариант этого примера можно получить из сетевого Хранилища кода.

*Пример 2.8. Преобразователь температуры Фаренгейт/Цельсий*

```
var fahrenheit;           // Температура по Фаренгейту
var celsius;              // Температура по Цельсию
var convertDirection;     // Система, в которую осуществляется преобразование.
                          // Допустимые значения – "fahrenheit" и "celsius"
fahrenheit = 451;                 // Задать температуру по Фаренгейту
celsius = 20;                     // Задать температуру по Цельсию
convertDirection = "celsius";     // В данном случае преобразовать в значение
                                  // по шкале Цельсия

if (convertDirection == "fahrenheit") {
   result = (celsius * 1.8) + 32;   // Вычислить значение по Цельсию.
   // Вывести результат
   trace (celsius + " degrees Celsius is " + result + " degrees Fahrenheit.");
} else if (convertDirection == "celsius") {
   result = (fahrenheit - 32) / 1.8;   // Вычислить значение по Фаренгейту.
   // Вывести результат
   trace (fahrenheit + " degrees Fahrenheit is " + result + " degrees Celsius.");
} else {
   trace ("Неверное направление преобразования.");
}
```

# Вперед!

Теперь нам все известно о том, как хранится информация в переменных, и настало время подробнее изучить их содержимое, а именно: *данные*. В следующих трех главах мы узнаем, что такое данные, как их можно 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, т. е. с информацией, которая обрабатывается в сценариях. В данной главе мы более глубоко исследуем данные и узнаем, как ActionScript определяет данные, распределяет их по категориям и хранит. Мы увидим также, как создавать и классифицировать данные.

## Данные и информация

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

```
0101010101010101101010110110101010101010100000101010101010110101010
1010101010101010111010101010101010101010101010101111101010101010101
0101010101010101010101010111010101010101010101010101010101010101010
```

Данные (data) – это информация в своем необработанном состоянии, т. е. сырая и бессмысленная. Напротив, информация имеет смысл. Рассмотрим, например число 8008898969. В качестве исходных данных оно не слишком осмысленно, но если классифицировать его семантически как телефонный номер (800) 889-8969, данные становятся полезной информацией.

В данной главе мы узнаем, как придать смысл необработанным компьютерным данным, чтобы они стали информацией, понимаемой человеком.

# Сохранение смысла данных с помощью типов данных

Как хранить информацию в компьютере в виде необработанных данных, без потери 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 регистрационных номеров почтовых посылок, – мы сохраняем контекст, а следовательно, и смысл наших данных. Когда установлена категория, каждое из неопределенных до того семизначных чисел обретает смысл.

Языки программирования используют *типы данных*, чтобы создать элементарные категории для наших 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 ниже. 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 число звезд, которое нужно поместить на ночное небо. Для хранения данных мы используем в ActionScript следующие типы данных:

- Для текстовых последовательностей, таких как «`hi there`», в ActionScript есть тип данных *string* (строка). *Строка* является последовательностью символов (букв, цифр и знаков пунктуации).
- Для чисел, например 351 или 7.5, в ActionScript есть тип данных *number* (числовой). Числа используются для счета и математических уравнений.
- Для принятия логических решений ActionScript предоставляет тип *Boolean* (булев). 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).
- 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*.

- Для списков *отдельных* элементов данных в ActionScript есть тип данных *array* (массив).
- Для обработки экземпляров клипов в ActionScript есть тип данных *movieclip* (клип фильма).
- И наконец, для произвольных встроенных и определенных пользователем классов данных в ActionScript есть очень мощный тип данных *object* (объект).

Любой элемент данных, которые мы храним в ActionScript, попадает в одну из этих категорий. Перед тем как изучить все типы данных в главе 4 «Элементарные типы данных», 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 наших сценариях.

*Символьное выражение* (или *литерал*) является последовательностью букв, цифр и знаков пунктуации, которая и *есть* элемент данных. Литерал данных – это буквальное описание данных в исходном коде программы. Он является противоположностью переменной, являющейся контейнером, просто содержащим данные. Для каждого типа данных есть собственные правила создания литералов. Вот несколько примеров литералов:

```
"loading...please wait"  // Строковый литерал
1.51                     // Числовой литерал
["jane", "jonathan"]     // Литерал массива
```

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

Мы можем также программируемым образом создавать данные с помощью *сложных выражений*. Сложные выражения представляют данные как фразы кода, значения которых должны вычисляться, а не восприниматься буквально. Вычисленное значение относится к представляемым данным. Например, каждое из следующих сложных выражений имеет результатом один элемент данных:

```
1999 + 1        // Получаем данное 2000
"hi " + "ma!"   // Получаем элемент данных "hi ma!"
firstName       // Дает значение переменной firstName
_currentframe   // Дает номер кадра текущей позиции воспроизводящей головки
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*.

При использовании как литералов, так и сложных выражений для создания данных 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";
```

Как отнести данные к надлежащему типу? То есть как указать, что данное является числом, строкой, массивом или чем-нибудь еще? В большинстве случаев мы не определяем сами категорию данных: интерпретатор ActionScript автоматически присваивает тип каждому данному или определяет его, основываясь на наборе внутренних правил.

## Автоматическая типизация литералов

Интерпретатор определяет тип литерального данного, 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 на то, что это массив

{x: 234, y: 456}     // Фигурные скобки и пары имя/значение свойства,
                     // разделенные запятыми, указывают на то, что это объект
```

Как видите, использование правильного синтаксиса с литералами данных 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 тому, который нам нужен. Однако в некоторых двусмысленных случаях требуется понимать правила, которые использует интерпретатор для определения типа данных выражения (см. примеры 2.2 и 2.3). Рассмотрим следующее выражение:

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

Операнд слева от + является строкой («1»), а операнд справа является числом (2). Оператор + может использоваться как с числами (сложение), так и со строками (конкатенация). Что будет значением выражения *"1" + 2* – число 3 или строка «12»? Чтобы избавиться от неоднозначности, интерпретатор использует 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 является произвольным, но оно позволяет единообразно интерпретировать код. Правило было выбрано с учетом обычного использования оператора плюс: если один из операндов является строкой, 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 предполагаемому типу данных, вызывает неоднозначность. Это вынуждает интерпретатор производить автоматическое *преобразование* типа данных в соответствии с произвольными, но предсказуемыми правилами. Давайте рассмотрим случаи, в которых происходит автоматическое преобразование типов данных, и выясним, каковы будут 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: первый является строкой, а второй – числом. Мы видели, что интерпретатор объединил два значения в строку «12». Обрати-

те 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 типов данных являются автоматическими: можно изменять тип данных явным образом, чтобы запретить назначение их типа, которое ActionScript осуществил бы по умолчанию.

## 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 он будет пытаться преобразовать данные типа B в данные типа A. Например, в приведенном ниже 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` (т. е. Not-a-Number – «не число»). `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 внутреннего 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.

Выражение, приводящие к числовому значению `NaN`, не очень употребительно: большинство преобразований имеет более конструктивные результаты. Например, если в строке содержатся только числовые символы, ее можно преобразовать в пригодное число. Выражение

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

внутренне преобразуется в следующее:

```
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 предложениями или предложениями цикла. Чтобы быть уверенным в результате выражения, в котором происходит автоматическое преобразование, нужно ответить на три вопроса: (a) какой тип данных предполагается в данном контексте; (b) что произойдет, если в этом контексте появится неожиданный тип данных; и, (c) если произойдет преобразование, каким будет результирующее значение?

Для ответа на первый и второй вопросы необходимо обратиться к соответствующим темам в данной книге (например, чтобы определить тип данных, ожидаемый в условном предложении операторе, см. главу 7 «Предложения»).

В следующих трех таблицах приводятся правила автоматического преобразования и 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` |
| Булевы | `1`, если исходное значение `true`; `0`, если исходное значение `false` |
| Числовая строка | Эквивалентное числовое значение, если строка состоит только из чисел десятичной системы 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` |
| `"Infinity"` | `Infinity` (бесконечность) |
| `"-Infinity"` | `-Infinity` (отрицательная бесконечность) |
| `"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" |
| Булевы | "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 при вызове метода *toString( )* объекта. По умолчанию, метод объекта *toString( )* возвращает "[object Object]". Метод *toString( )* может быть переделан так, чтобы возвращать более полезный результат (например, *toString( )* объекта *Date* возвращает: "Sun May 14 11:38:10 EDT 2000") |
| Клип | Путь к экземпляру клипа, заданный в абслютном виде, начиная с уровня документа в проигрывателе, например, "_level0.ball" |

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

*Таблица 3.3. Преобразование в булев тип*

| Исходные данные | Результат после преобразования |
|---|---|
| undefined | false |
| null | false |
| NaN | false |
| 0 | false |
| Infinity | true |
| -Infinity | true |
| Прочие числовые значения | true |

*Таблица 3.3 (продолжение)*

| Исходные данные | Результат после преобразования |
|---|---|
| Непустая строка | `true`, если строка может быть преобразована в допустимое число, отличное от нуля, и `false`, если нет; в ECMA-262 непустая строка всегда преобразовывается в `true` (Flash 5 нарушает эти правила 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 какой-либо элемент данных в строку, можно вызвать метод *toString( )*, например:

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

При вызове метода *toString( )* на числе 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. И функция и класс описаны в части III «Справочник по языку».

## Преобразование в строку путем конкатенации с пустой строкой

Поскольку оператор «плюс» (+) в правилах автоматического преобразования отдает предпочтение строкам, прибавление "" к любому элементу данных преобразует его в строку.

```
x + "";       // Преобразует x в строку
523 + "";     // Преобразует 523 в строку "523"
```

## Преобразование в число с помощью функции Number()

Так же, как функция *String( )* преобразует данные в строковый тип, функция *Number( )* преобразует свой аргумент в тип *number (числовой)*. Если преобразование в 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. И функция, и конструктор описаны в части III.

Поскольку введенные пользователем данные в экранных текстовых полях всегда принадлежат текстовому типу, при выполнении математических операций необходимо преобразовывать текстовые поля в числа. Например, для нахождения суммы текстовых полей `price1` и `price2` используем код:

```
totalCost = Number(price1) + Number(price2);
```

В противном случае `price1` и `price2` будут конкатенированы как строки, а не сложены как числа. Дополнительные сведения о текстовых полях см. в главе 18 «Экранные текстовые поля».

## Преобразование в число путем вычитания нуля

Чтобы заставить интерпретатор преобразовать элемент данных в число, можно вычесть ноль 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 использования *parseInt( )*:

```
parseInt("1a")                // Извлекает 1
parseInt("1.3a"               // Извлекает 1
parseInt("     1a")           // Извлекает 1
parseInt("I am 14 years old") // Возвращает NaN (первый отличный от
                              // пробела символ числом не является)
parseInt("14 years old")      // Извлекает 14
```

Функция *parseFloat( )* извлекает 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, имеющее 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( )*, является последовательностью символов, начинающейся с первого отличного от пробела символа и заканчивающейся символом перед первым нечисловым символом (любым символом, отличным от +, –, 0–9 или десятичной точки).

Вот некоторые примеры использования *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( )* см. в части III.

## Преобразование в булев тип

Когда требуется преобразовать элемент данных в булев тип, можно применять глобальную функцию *Boolean( )*, использующую синтаксис, аналогичный синтаксису функций *String( )* и *Number( )*. Например:

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

Не путайте глобальную функцию *Boolean( )* с одноименным конструктором встроенного класса. И функция и класс описаны в части III.

## Срок действия преобразования

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

```
var x = "10";       // x является строкой
y = x - 5;          // y теперь 5; значение x было преобразовано в число
trace(typeof x);    // Выводит: "string"; преобразование было временным
```

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, так как произошло при присваивании.
```

## Преобразование типов данных Flash 4 в типы данных Flash 5

Во Flash 4 строковые и числовые операторы были совершенно различными: одна группа операторов работала только с числами, а другая работала только со строками. Например, оператором конкатенации строк во Flash 4 был `&`, а оператором математического сложения был `+`. Аналогично, сравнение строк производилось с помощью операторов `eq` и `ne`, а сравнение чисел – с помощью `=` и `<>`. В табл. 3.4 приводятся синтаксические конструкции Flash 5 для аналогичных операторов Flash 4.

*Таблица 3.4. 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 | `>=`, `>`, `<=`, `<` | `>=`, `>`, `<=`, `<` |

Некоторые операторы Flash 5 могут действовать как над строками, так и над числами. Например, при действии над строками оператор + конкатенирует свои операнды, образуя новую строку. Но при использовании с числами оператор + складывает два свои операнда математически. Аналогично, оператор равенства (==) и оператор неравенства (!=) используются 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 автоматически вставляется функция *Number( )*, аргументом которой являются любые числовые данные, используемые в качестве операндов следующих потенциально неоднозначных операторов (если только операнд не является числовым операндом):

```
+, ==, !=, <>, <, >, >=, <=
```

В файлах Flash 4, конвертированных во Flash 5, будет также произведена замена оператора конкатенации строк (&) на новый оператор *add*. В табл. 3.5 представлены примеры перевода операторов Flash 4 в операторы Flash 5.

*Таблица 3.5. Примеры трансляции операторов Flash 4 в операторы Flash 5*

| Синтаксис Flash 4 | Синтаксис Flash 5 |
|---|---|
| `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` |

## Определение типа существующего элемента данных

Чтобы определить тип данных, содержащихся в заданном выражении, прежде чем продолжить выполнение участка кода, используется оператор *typeof*, как показано ниже:

```
typeof expression;
```

Оператор *typeof* возвращает строку, указывающую тип данных *expression*, согласно табл. 3.6.

*Таблица 3.6. 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 предложении *for-in* функция *typeof* дает удобный способ нахождения всех экземпляров клипа на временной диаграмме. Когда клипы определены, их можно присвоить массиву для проведения программной обработки. (Если пример 3.1 не полностью вам понятен, вернитесь к нему, закончив изучение части I «Основы ActionScript».)

*Пример 3.1. Заполнение массива динамически идентифицируемыми клипами*

```
var childClip = new Array();
var childClipCount = 0;

for (i in _root) {
   thisItem = _root[i];
   if (typeof thisItem == "movieclip") {
     // Обратите внимание на постфиксный оператор инкрементирования
     childClip[childClipCount++] = thisItem;
  }
}

// После заполнения массива можно использовать его для
// обработки клипов, которые в нем находятся
```

```
childClip[0]._x = 0;  // Поместить первый клип у левого края
                      // рабочего стола
childClip[1]._y = 0;  // Поместить второй клип у верхнего края
                      // рабочего стола
```

# 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 содержит один элемент данных (в противоположность массиву, содержащему несколько элементов) и описывает этот элемент данных буквально. Такой тип данных очень прост.

ActionScript поддерживает следующие элементарные типы данных: *number*, *string*, *boolean*, *undefined* и *null*. В ActionScript нет отдельного типа данных для одиночных символов, типа *char* в C/C++.

Элементарные типы данных могут содержать текстовые сообщения, номера кадров, размеры клипов и так далее, но их не так легко приспособить для более сложных данных. Для более искусной обработки данных – например, при моделировании движения десятка прыгающих мячиков или организации вопросника с 500 вопросов и ответов – мы обращаемся к использованию *сложных* (*composite*) типов данных. При использовании сложных типов данных можно 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 типом объекта и потому рассматриваются как сложные данные, но в таком качестве они используются редко. Подробнее о функциях как типе данных см. в главе 9 «Функции».

В то время как отдельное число является простым элементом данных, список (т. е. *массив*) нескольких чисел является *сложным* данным. Вот практический пример, показывающий пользу сложных типов данных. Допустим, мы хотим сохранить конфигурационный файл клиента с именем Derek. Можно создать ряд переменных, хранящих атрибуты клиента Derek как простые значения:

```
var custName = "Derek";
var custTitle = "Coding Genius";
var custAge = 30;
var custPhone = "416-222-3333";
```

Однако такой формат становится довольно утомительным, если добавятся еще хотя бы несколько пользователей. Для хранения всех атрибутов потребуется использовать последовательно именуемые перемен-

ные – cust1Name, cust2Name, cust1Title, cust2Title и т. д. Но с помощью массива можно хранить данные более эффективно:

```
cust1 = ["Derek", "Coding Genius", 30, "416-222-3333"];
```

Если нужно добавить других клиентов, мы просто создаем новые массивы:

```
cust2 = ["Komlos", "Comic Artist", 28, "515-515-3333"];
cust3 = ["Porter", "Chef", 51, "515-999-3333"];
```

Мило и аккуратно. В следующих главах мы узнаем значительно больше о сложных типах.

# Вперед!

Мы познакомились с представлением данных в ActionScript и готовы к дальнейшему изучению. В главе 4 мы изучим типы данных *number*, *string*, *boolean*, *undefined* и *null*, а в главе 5 «Операторы» научимся обрабатывать данные. В последующих главах 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, в ActionScript поддерживаются простые типы данных *number*, *string*, *boolean*, *undefined* и *null*. В этой главе мы научимся определять, просматривать и изменять данные каждого из этих типов.

## Тип Number

Числа используются для счета, математических действий и хранения числовых свойств в фильмах (таких как номер текущего кадра или его местоположение на рабочем столе). 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. Поэтому 1, 34523, –3, 0 и –9999999 являются целыми, а 223.45, –0.56 и 1/4 являются числами с плавающей точкой.

# Числовые литералы

Ранее мы узнали, что *литерал* является непосредственным представлением отдельного фиксированного значения. Тип данных *number* 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 такие числовые понятия, как «бесконечность».

## Целочисленные литералы

Целочисленные литералы, например 1, 2, 3, 99 и –200, должны следовать 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. Описание допустимых значений смотрите в свойствах `MIN_VALUE` и `MAX_VALUE` объекта *Number* в части III «Справочник по языку».
- Целые числа десятичной системы счисления не должны начинаться с нуля (например, 002, 000023 или 05).

Целые числа не всегда записаны в десятичной системе счисления. ActionScript поддерживает также восьмеричные и шестнадцатеричные числовые литералы. Примеры чисел перечисленных систем счисления можно найти по адресу:

*http://www.moock.org/asdg/technotes*

Ведущий ноль мы используем для обозначения восьмеричного числа. Например, для представления в ActionScript восьмеричного числа 723 мы напишем:

```
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 десятичной системы счисления. 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 число десятичной системы счисления
десятичная точка (.)
дробная часть (представленная как десятичное число)
порядок

Первые три составляющие довольно просты: в числе 3.14 «3» является десятичным числом, «.» – десятичная точка, а «14» – дробная часть. Однако четвертую составляющую (порядок) нужно рассмотреть более внимательно.

Чтобы представить очень большое положительное или отрицательное число как число с плавающей точкой, можно присоединить к числу порядок (десятичный логарифм) с помощью буквы E (или e). Чтобы определить величину числа с порядком, нужно умножить число на 10 в степени, заданной порядком. Например:

```
12e2   // 1200 (10 в квадрате равно 100, при умножении на 12 дает 1200)
143E-3 // 0.143 (10 в степени -3 дает .001, при умножении на 143 дает 0.143)
```

Возможно, вы узнали стандартную научную запись. Если вы не очень сильны в математике, вот простой совет по преобразованию: если порядок положительный, передвиньте десятичную точку вправо на равное ему число разрядов; если порядок отрицательный, передвиньте десятичную точку влево на равное ему число 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. Если нужно возвести число в произвольную степень, используется встроенная функция *Math.pow( )*, документированная в части III.

### 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.) Это значит, что 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 вычисления в ActionScript округляются нежелательным образом, например, вместо 0.143 получается 0.14300000000000001. Это происходит потому, что компьютеры преобразуют числа любой системы счисления во внутреннее двоичное представление, в результате чего могут образовываться бесконечные дроби в двоичной системе (типа десятичной дроби 0.3333333...). Однако точность компьютеров конечна, и они не могут точно представлять бесконечные дроби. Если разница может отрицательно повлиять на выполнение кода, следует вручную округлять числа. Например, округлим `myNumber` до трех десятичных знаков:

```
myNumber = Math.round(myNumber * 1000) / 1000;
```

А вот функция, которую можно использовать для округления произвольного числа до заданного числа десятичных знаков:

```
function trim(theNumber, decPlaces) {
  if (decPlaces >= 0) {
    var temp = Math.pow(10, decPlaces);
    return Math.round(theNumber * temp) / temp;
  }
}

// Округлить число до двух десятичных знаков
trace(trim(1.12645, 2));  // Выводит: 1.13
```

## Особые 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 такие понятия: «не число», «минимальное разрешенное значение», «максимальное разрешенное значение», «бесконечность», «отрицательная бесконечность».

Каждое из этих особых значений можно присваивать переменным и свойствам, либо использовать в символьных выражениях, как любой другой числовой литерал. Чаще, однако, специальные числовые значения возвращаются интерпретатором как результат вычисления каких-либо выражений.

## «Не число»: NaN

Иногда при математическом вычислении или попытке преобразования данных возвращается 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, которые имеют *числовой* тип, но не являются настоящим числом, ActionScript имеет ключевое слово `NaN`. Хотя `NaN` не представляет никакого числа, это допустимое значение *числового* типа, что демонстрируется в следующем коде:

```
x = 0/0;
trace(x);          // Выводит: NaN
trace(typeof x);   // Выводит: "number"
```

Так как `NaN` не является конечным числовым значением, при сравнении оно не равно самому себе. Если две переменные содержат значение `NaN`, они не считаются равными (хотя нам они кажутся равными). Чтобы решить эту проблему, используется встроенная функция *isNaN( )*, которая проверяет, не содержит ли переменная значение `NaN`:

```
x = 12 - "this doesn't make much sense";  // x теперь NaN
trace(isNaN(x));                          // Выводит: true
```

## Минимальное и максимальное допустимое значение: MIN_VALUE and MAX_VALUE

ActionScript предоставляет широкий, но не безграничный диапазон чисел. Число 1.7976931348623157e+308 является максимально допустимым, а минимально допустимым значением – 5e–324. Очевидно, использовать эти числа несколько неудобно, поэтому вместо них используются специальные значения – `Number.MAX_VALUE` и `Number.MIN_VALUE`.

`Number.MAX_VALUE` удобно использовать, если нужно 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` является самым маленьким допустимым *положительным* значением, а не самым большим по абсолютной величине *отрицательным* значением; самым большим по абсолютной величине отрицательным значением является `Number.MAX_VALUE`.

### Бесконечность и отрицательная бесконечность: Infinity и -Infinity

Если вычисление дает результат, который больше `Number.MAX_VALUE`, ActionScript использует для представления результата вычислений ключевое слово `Infinity`. Аналогично, если вычисление приводит к результату, который больше по абсолютной величине максимального разрешенного отрицательного числа, ActionScript называет результат ключевым словом `-Infinity`. `Infinity` и `-Infinity` можно использовать непосредственно как числовые литералы.

### Иррациональные числа

Кроме предоставления особых числовых значений `NaN`, `Infinity`, `-Infinity`, `Number.MAX_VALUE` и `Number.MIN_VALUE`, ActionScript обеспечивает удобный доступ к математическим константам через объект *Math*. Например:

```
Math.E        // Значение числа e, основания натуральных логарифмов
Math.LN10     // Натуральный логарифм 10
Math.LN2      // Натуральный логарифм 2
Math.LOG10E   // Логарифм e в десятичной системе счисления
Math.LOG2E    // Логарифм e в двоичной системе счисления
Math.PI       // Число Пи (т. е. 3.1415926...)
Math.SQRT1_2  // Квадратный корень из 1/2
Math.SQRT2    // Квадратный корень из 2 (т. е. 1.4142135...)
```

Константы представляют собой просто сокращенную запись приближенных значений часто встречающихся иррациональных чисел. Используются эти 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 части III.

## Действия с числами

Для работы с числами можно объединять их в математические выражения с помощью операторов, а также 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 выра-

жения вычисляются в порядке, определяемом *приоритетами*, приведенными в табл. 5.1. Например, умножение выполняется раньше, чем сложение. Вот примеры допустимого использования 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 используются встроенные функции объекта *Math*. Например:

```
Math.abs(x)       // Абсолютная величина x
Math.min(x, y)    // Меньшее из значений x и y
Math.pow(x, y)    // x в степени y
Math.round(x)     // x, округленная до ближайшего целого
```

Математические функции возвращают значения, которые можно использовать в выражениях как действительные числа. 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 1:

```
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
dieRoll = Math.floor(Math.random() * 6 + 1);
```

# Строковый тип данных

*Строка* – это тип данных, используемый для текстовых данных (букв, знаков пунктуации и других символов). Строковым литералом является любая комбинация символов, заключенная в кавычки:

```
"asdfksldfsdfeoif"  // Расстроенная строка
"greetings"         // Дружественная строка
"moock@moock.org"   // Строка саморекламы
"123"               // Похоже на число, но это строка
'singles'           // Одинарные кавычки тоже допустимы
```

Прежде чем рассматривать, как образовывать строковые литералы, исследуем, какие символы допускается использовать в строках.

## Кодировка символов

Как и все компьютерные данные, символы текста хранятся в памяти с использованием числового кода. Они *кодируются* для хранения и *декодируются* для отображения при помощи *набора символов* (*character set*), который строит *отображение* символов на их численные коды, т. е. связывает их между собой. Наборы символов различны для разных языков и алфавитов. Более старые западные приложения используют наборы, производные от ASCII – стандартного набора, содержащего лишь 128 символов – английский алфавит, цифры и основные символы пунктуации. Современные 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. ActionScript использует в качестве главного набора символов стандарт ISO-8859-1, известный также как *Latin 1*.

Набор символов Latin 1 охватывает большинство европейских языков, таких как французский, немецкий, итальянский, испанский, португальский и так далее, но не такие языки, как греческий, турецкий, славянские и русский. *Unicode*, предпочтительный международный стандарт кодировки символов, содержащий до миллиона символов, *не* поддерживается ActionScript (поддержка Unicode значительно увеличила бы размер Flash Player). Однако ActionScript поддерживает второй набор символов для японского языка, называемый *Shift-JIS*. При работе с текстом в ActionScript можно использовать любой символ из Latin 1 или Shift-JIS.

Несмотря на отсутствие поддержки собственно Unicode, можно использовать стандартные escape-последовательности Unicode для представления любых символов Latin 1 или Shift-JIS. Можно также обрабатывать строки символов с помощью функций в стиле Unicode. Теоретически поддержка Unicode может быть добавлена во Flash в будущем без нарушения работы старого кода.

В приложении B «Набор символов Latin 1 и коды клавиш» перечислены кодовые позиции (*code point*) всех символов Unicode, представляющие собой числовые позиции символов в наборе Unicode. Позднее мы увидим, как использовать эти кодовые позиции для работы с символами в сценариях.

# Строковые литералы

Чаще всего для образования строки используются одинарные или двойные кавычки, в которые заключаются группы символов из наборов Latin 1 или Shift-JIS:

```
"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, она ей должна и заканчиваться. Аналогично, если в начале строки используется одинарная кавычка, завершить строку тоже должна одинарная. 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 нужно определить, содержит ли переменная строковое значение, которое можно использовать:

```
if (firstName == "") {
  trace("Вы забыли ввести свое имя!");
}
```

Однако сравнение переменной с пустой строкой `""` может не всегда действовать желаемым образом. Помните, что `""` считается равной числу 0 и булеву значению `false` (см. табл. 3.1 и 3.3). Поэтому, чтобы быть уверенным в проверке пустой строки, сначала следует убедиться, что переменная принадлежит строковому типу, например:

```
if (typeof firstName == "string" && firstName == "") {
  trace("Вы забыли ввести свое имя!");
}
```

## Escape-последовательности

Ранее мы видели, что одинарные кавычки (`'`) можно использовать внутри литералов, заключенных в двойные кавычки, а двойные кавычки (`"`) можно использовать внутри литералов, заключенных в одинарные. Но что, если нужно использовать и те, и другие? Например:

```
'I remarked "Nice night, isn't it?"'
```

В таком виде эта строка кода вызовет ошибку, поскольку интерпретатор подумает, что строковый литерал заканчивается апострофом в слове «isn't». Интерпретатор прочтет это так:

```
'I remarked "Nice night, isn'  // Оставшаяся часть считается
                               // бессмысленным мусором
```

Чтобы использовать одинарную кавычку внутри строкового литерала, ограниченного одинарными кавычками, нужно использовать *escape-последовательность*.

Escape-последовательность представляет значение литеральной строки с помощью символа обратной косой черты (\), за которым следует код, соответствующий нужному символу, или сам символ. 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, представлены в табл. 4.1.

*Таблица 4.1. Escape-последовательности в ActionScript*

| Escape-последовательность | Значение |
|---|---|
| \b | Символ забоя (возврата на один символ с удалением) (ASCII 8) |
| \f | Символ перевода формата (ASCII 12) |
| \n | Символ новой строки; вызывает перенос строки (ASCII 10) |
| \r | Символ возврата каретки (CR); вызывает перенос 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 \ как начала escape-последовательности |

## Escape-последовательности в стиле Unicode

Не все символы Latin 1 и Shift-JIS есть на клавиатуре. Чтобы включить в строку отсутствующие на клавиатуре символы, используются escape-последовательности в стиле Unicode. Заметьте, что Flash на самом деле не поддерживает Unicode, он просто эмулирует соответствующий синтаксис.

Escape-последовательности в стиле Unicode начинаются с обратной косой черты и символа *'u' на нижнем регистре*, т. е. `\u`, а далее следует четырехзначное шестнадцатеричное число, соответствующее кодовой позиции символа в Unicode, например:

```
\u0040  // Знак @
\u00A9  // Символ copyright
\u0041  // Заглавная бкува "A"
```

*Кодовая позиция* является уникальным идентификационным номером, присваиваемым каждому символу в наборе символов Unicode. Смотрите список кодовых позиций Unicode для Latin 1 в приложении B. Кодовые позиции для Shift-JIS можно найти на сайте Консорциума Unicode:

*ftp://ftp.unicode.org/Public/MAPPINGS/EASTASIA/JIS/SHIFTJIS.TXT*

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
```

Помимо использования escape-последовательностей Unicode, любой символ можно вставить в строку с помощью более громоздкой встроенной функции *fromCharCode( )*, описываемой ниже в разделе «Функции кодов символов». Заметьте, что как с escape-последовательностями Unicode, так и с функцией *fromCharCode( )*, Flash 5 поддерживает только те кодовые позиции, которые отображаются на символы в наборах Latin 1 и Shift-JIS. Вставка других кодовых позиций не даст правильного символа Unicode, пока в будущих версиях Flash не появится поддержка дополнительных символов Unicode.

# 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 производить ловкие зрительные 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 кодовой позиции, изменять регистр символов строки и даже преобразовывать строку в переменную или имя 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 со значением «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 на то, что символ пробела не совпадает с пустой строкой, которую мы видели выше, потому что в пустой строке между кавычками не содержится никаких символов.

Можно также конкатенировать 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, одно (пробел) – в строковом литерале. Проблем нет. Так часто бывает.

Иногда нужно добавить символы к существующей строке. Например, можно изменить интонацию приветственного сообщения:

```
var greeting = "Hello";      // Наше приветственное сообщение
greeting = greeting + "?";   // Наше вопросительное приветственное
                             // сообщение: "Hello?"
```

Этот код решает свою задачу, но обратите внимание на дважды встречающееся в строке 2 слово `greeting`. Более эффективным является использование оператора +=, который добавляет строку, находящуюся справа, к строковой переменной слева:

```
var greeting = "Hello";     // Наше приветственное сообщение
greeting += "?";            // Наше вопросительное приветственное
                            // сообщение: "Hello?"
```

Оператор конкатенации строк Flash 4 (`&`) выполняет во Flash 5 *другую* операцию (поразрядное И). При экспорте файлов *.swf*, относящихся к Flash 4, нужно использовать для конкатенации строк оператор *add*. Отметим, что *add* поддерживается только для обратной совместимости: во Flash 5 предпочтительнее использовать оператор +.

## Функция concat()

Функция *concat( )* добавляет к строке символы, подобно +=. Так как *concat( )* является функцией, она использует 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 строки, к которой применяется, она лишь возвращает конкатенированное строковое значение. Чтобы использовать это значение, нужно присвоить его переменной или другому контейнеру данных. Чтобы понять разницу между += и *concat( )*, изучите внимательно следующий код:

```
var greeting = "Hello";
greeting.concat("?");
trace(greeting);  // Выводит "Hello"; переменная greeting
                  // не затронута функцией concat

finalGreeting = greeting.concat("?");
trace(finalGreeting);  // Выводит "Hello?"
```

Функция *concat( )* может принимать несколько аргументов, т. е. объединять несколько разделенных запятыми строк в одну строку:

```
firstName = "Karsten";

// Устанавливает finalGreeting в "Hello, Karsten?"
finalGreeting = greeting.concat(" ", firstName, "?");
```

что равносильно следующему:

```
finalGreeting = greeting;
finalGreeting += " " + firstName + "?";
```

# Сравнение строк

Чтобы проверить, одинаковы ли две строки, используются операторы равенства (==) и неравенства (!=). Строки часто сравнивают при выполнении кода в зависимости от условия. Например, если пользователь вводит пароль, нужно сравнить введенную им строку с фактическим паролем. Результат сравнения определяет поведение кода.

## 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
```

Если операнд справа содержит те же самые символы в том же самом порядке, что и операнд слева, две строки считаются равными, и результатом является булево значение `true`. Однако у букв верхнего и нижнего регистра разные кодовые позиции в наборе символов, поэтому они не считаются равными. Все следующие сравнения дают значение `false`:

```
"olive-orange" == "olive orange"  // Не равны
"nighttime" == "night time"       // Не равны
"Day 1" == "day 1"                // Не равны
```

Поскольку результатом сравнения являются булевы значения `true` или `false`, можно использовать их в качестве проверочных выражений в условных предложениях и циклах, например:

```
if (userGuess == password) {
  gotoAndStop("classifiedContent");
}
```

Если выражение *(userGuess == password)* имеет значение `true`, то выполняется предложение *gotoAndStop( «classifiedContent» )*. Если выражение имеет значение `false`, предложение *gotoAndStop( «classifiedContent» );* будет пропущено.

Далее в этой главе мы больше узнаем о булевых значениях. А условные предложения мы изучим в главе 7 «Условные предложения».

Чтобы проверить неравенство двух строк, используется 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 в том случае, если две строки не равны:

```
if (userGender != "boy") {
  // здесь идет код только для девочек...
}
```

При экспорте файлов *.swf*, относящихся к Flash 4, для проверки равенства и неравенства строк следует использовать более старые операторы `eq` и `ne`. Но `eq` и `ne` поддерживаются в целях обратной совместимости, а во Flash 5 предпочтительнее операторы `==` и `!=`.

## Сравнение по порядку символов и алфавиту

Две строки можно также сравнивать, основываясь на порядке символов. Ранее мы видели, что каждому символу присвоен числовой код, и что эти коды в наборе символов упорядочены. Можно проверить, 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"
"2" > "&"
"r" <= "R"
"$" >= "@"
```

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.

Поскольку символы ‘A’ … ‘Z’ и ‘a’ … ‘z’ сгруппированы в наборе символов Latin 1 в алфавитном порядке, то сравнение порядка символов часто используется для определения того, которая из двух букв рань-

ше следует в алфавите. Обратите, однако, внимание, что все заглавные буквы в наборе Latin 1 предшествуют всем строчным буквам. Если забыть об этом, можно получить неожиданные результаты:

```
"Z" < "a"       // Имеет значение true
"z" < "a"       // Имеет значение false
"Cow" < "bird"  // Имеет значение true
```

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 набора Latin 1 или Shift-JIS, чем сравниваемый символ правого операнда. Если оба операнда полностью совпадают, > возвращает `false`:

```
"b" > "a"      // true
"a" > "b"      // false
"ab" > "ac"    // false (сравниваемый символ – второй)
"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" занимают разные кодовые позиции)
```

*Меньше (<)*

Возвращает `true`, если сравниваемый символ левого операнда находится раньше в порядке символов набора Latin 1 или Shift-JIS, 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, обратитесь к приложению B.

В следующем примере определяется, является ли символ буквой латинского алфавита (а не цифрой, символом пунктуации или еще каким-либо знаком):

```
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. Мы изучим ИЛИ в главе 5 «Операторы».

При экспорте файлов *.swf, относящихся к* Flash 4, следует использовать более старые операторы сравнения строк `gt`, `ge`, `lt` и `le`. Хотя в целях обратной совместимости старые операторы и поддерживаются, во Flash 5 предпочтительнее эквивалентные операторы `>`, `>=`, `<`, и `<=`.

## Использование встроенных функций

За исключением функции *concat( )*, все использовавшиеся нами до сих пор 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, нужно выполнить *вызов функции*, который имеет вид:

```
string.functionName(arguments)
```

Вот пример применения функции *charAt( )* к переменной `myString`:

```
myString.charAt(2)
```

Строковые функции возвращают данные, которые некоторым образом связаны с исходной строкой. 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 выдать 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. 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, например, удлинить строку таким способом:

```
axiom.length = 100;  // Чудесно, но не работает
```

При экспорте файлов *.swf*, относящихся к Flash 4, следует использовать более старую функцию *length( )*, как показано ниже. Функция *length( )* поддерживается для обратной совместимости, но во Flash 5 предпочтительнее использовать свойство `length`.

Ниже мы используем функцию Flash 4 *length( )* для вывода числа символов в слове «obsolete»:

```
trace (length("obsolete"));  // Выводит: 8
```

## 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, а *index* является целым числом или выражением, дающим целое число и указывающим на позицию символа, который нужно извлечь. Значение *index* должно быть между 0 и *string.length - 1*. Если *index* находится вне этого диапазона, 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 есть "d"
lastLetter = country.charAt(country.length - 1); // lastLetter есть "a"
```

## Функция indexOf()

Функция *indexOf( )* используется для поиска символов в строке. Если строка, в которой производится поиск, содержит искомую последовательность, то *indexOf( )* возвращает индекс (т. е. позицию) ее первого вхождения в строку. В противном случае возвращается значение −1. Общий вид *indexOf( )* такой:

```
string.indexOf(character_sequence, start_index)
```

где *string* может являться любым литеральным строковым значением или идентификатором, который содержит строку; *character_sequence* является строкой, которая ищется, и может быть литеральным строковым значением или идентификатором, содержащим строку, а *start_index* является начальной позицией для поиска. Если *start_index* опущен, то поиск производится с начала строки *string*.

Давайте используем *indexOf( )*, чтобы проверить, содержит ли строка символ *W*:

```
"GWEN!".indexOf("W");  // Возвращает 1
```

Да, *W* – второй символ в строке «GWEN!», поэтому мы получаем 1, индекс символа *W*. Вспомните, что индексы символов начинаются с 0, поэтому второй символ имеет индекс 1.

Что произойдет, если мы будем искать символ 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
if (email.indexOf("@") == -1) {
  formStatus = "Адрес электронной почты неверен.";
}
```

Мы не ограничены поиском только одиночных символов. Можно искать в строке целую последовательность символов. Поищем в адресе фирмы слово «Canada»:

```
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 происхождения 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 специфического для Северной Америки формата почты:

```
if (isCanadian) {
  mailDesc = "Please enter your postal code.";
} else {
  mailDesc = "Please enter your zip code.";
}
```

Функция *indexOf( )* может также помочь определить, какую часть строки нужно извлечь. Мы увидим, как это действует, когда изучим функцию *substring( )*.

## Функция lastIndexOf()

Функция *indexOf( )* возвращает местоположение *первого* вхождения последовательности символов в строку. Функция же *lastIndexOf( )* возвращает местоположение *последнего* вхождения последовательнос-

ти символов в строку или −1, если последовательность не найдена. Общий вид функции *lastIndexOf( )* точно такой же, как у *indexOf( )*:

```
string.lastIndexOf(character_sequence, start_index)
```

Единственное различие в том, что *lastIndexOf( )* осуществляет поиск в обратном направлении, а *start_index* указывает на самый правый символ, который мы хотим включить в поиск, а не самый левый. Если *start_index* опущен, то 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) в ActionScript не поддерживаются.

## Извлечение частей строк

Иногда длинная строка содержит последовательность символов, к которой хотелось бы иметь более удобный доступ. 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( )*.

### Функция substring()

Функция *substring( )* используется для извлечения из строки последовательности символов, основываясь на индексах начального и конечного символов. Функция *substring( )* имеет следующий формат:

```
string.substring(start_index, end_index)
```

где *string* является любым строковым литералом или идентификатором, который содержит строку, *start_index* является индексом первого символа, включаемого в подстроку, а *end_index* является символом, следующим *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 кончаются имя, второе имя и фамилия, поэтому обычно ищется некоторый *разделитель*, например, пробел, чтобы помочь определить, где находятся границы между словами. 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. Хотя следующие два вызова функции дают одинаковый результат, не следует привыкать к использованию *substring( )* с переставленными индексами, 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, length)
```

где *string*, как обычно, является любым строковым литералом или идентификатором, содержащим строку; *start_ index* является первым символом, который нужно включить в строку; *length* указывает, сколько символов нужно включить в нашу строку, начиная со *start_index* и ведя отсчет вправо. Если *length* опущена, подстрока начинается со *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 *length* недопустимо.

Функция *substr( )* Flash 5 ближе всего к функции Flash 4 *substring( )*, которая тоже использует для извлечения подстроки начальный индекс и длину.

## 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* является любым строковым литералом или идентификатором, который содержит строку, а *start_index* является индексом первого символа, включаемого в подстроку. Если *start_index* является положительным целым числом, то это обычный индекс символа; если *start_index* является отрицательным целым числом, то соответствующий ему индекс символа определяется отсчетом в обратном направлении от конца строки (т. е. как *string*.length + *start_index*). Наконец, *end_index* является символом *за* последним символом, который нужно 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 (т. е. как *string*.length + *end_index*).

При использовании неотрицательных индексов *slice( )* действует точно так же, как *substring( )*. Пользуясь отрицательными индексами, помните, что таким образом вы не получите подстроку с символами, расположенными в обратном порядке, и не получите строку от *end_index* до *start_index* в указанном порядке. 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 подстроки с помощью отрицательного *end_index*. Внимательно посмотрите, как 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.
```

## Функция split()

Рассматривавшиеся нами до сих пор функции извлечения строк возвращали только одну последовательность символов за одно обращение. Если нужно одним махом извлечь сразу кучу подстрок, можно воспользоваться мощной функцией *split( )*. (Так как функция *split( )* использует массивы, вам, возможно, придется пропустить сейчас эту функцию и вернуться к ней позднее после чтения главы 11 «Массивы».)

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 строковым литералом или идентификатором, который содержит строку, а *delimiter* является одним или несколькими символами, обозначающими, где должна расщепляться строка *string*. Типичными разделителями служат запятые, пробелы и символы табуляции. Например, расщепить строку по каждой запятой можно так:

```
theString.split(",")
```

Одним из ловких приемов, которые можно выполнить с помощью *split( )*, является разбиение предложения на отдельные слова. В нашем описании функций *substring( )*, *substr( )* и *slice( )* приходилось вручную вытаскивать каждую часть имени из строки «Steven Sid Mumby». Посмотрите, насколько легче это делается при использовании 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];
```

### Вопросы эффективности извлечения строк

Функции Flash 5 *substr( )* и *slice( )* фактически реализованы путем наведения внешнего лоска на старую функцию Flash 4 *substring( )* и потому выполняются немного дольше. Разница в скорости составляет миллисекунды, но при интенсивной обработке строк может стать ощутимой. При выполнении многократно повторяемых операций для повышения производительности используйте функцию *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);
```

## Сочетание исследования строки с извлечением подстроки

Мы посмотрели, как искать в строке символы и как их оттуда извлекать. Две эти задачи можно эффективно объединить вместе.

В большинстве рассмотренных примеров в качестве аргументов использовались литералы, например:

```
var msg = "Welcome to my website!";
var firstWord = msg.substring(0, 7);  // 0 и 7 суть числовые литералы
```

Это хорошая демонстрация того, как работает функция *substring( )*, но на практике *она* обычно используется иначе. Чаще нам не известно заранее содержимое строки, и аргументы должны генерироваться динамически. Например, мы обычно произносим не статическую фразу типа «вернуть подстроку с индекса 0 по индекс 7», а нечто динамическое

типа «вернуть подстроку, начиная с первого символа, и до первого пробела». Такой более гибкий подход не требует предварительного знания содержимого строки. Извлечем первое слово из переменной `msg`, объединив функции *substring( )* и *indexOf( )*:

```
var firstWord = msg.substring(0, msg.indexOf(" "));
```

Выражение *msg.indexOf(" ")* вычисляет индекс первого пробела в `msg`. Наш прием будет работать независимо от того, где находится пробел. Это позволяет работать со строками, которые изменяются во время работы программы, и позволяет избежать подсчета символов, при котором легко допустить ошибку.

Сочетаний исследования строки с извлечением строки может быть огромное количество. В примере 4.2 извлекается второе слово переменной `msg` без жесткого кодирования индексов символов. На обычном языке нам нужно «извлечь из `msg` подстроку, начиная с символа после первого вхождения пробела и заканчивая символом перед вторым вхождением пробела». Мы сохраняем местонахождение первого и второго пробела в виде переменных, что делает более очевидным то, что происходит.

*Пример 4.2. 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

Строки можно переводить на верхний или нижний регистр с помощью встроенных функций *toUpperCase( )* и *toLowerCase( )*. Обычно они используются для форматирования строк при выводе и для сравнения строк, в которых используются разные регистры.

## Функция toUpperCase()

Функция *toUpperCase( )* преобразует все символы строки в верхний регистр (т. е. в заглавные 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* является любым строковым литералом или идентификатором, который содержит 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." msg не тронута 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
}
```

Пример 4.3 показывает, как с помощью функций строк и преобразования регистра можно развлечься и создать анимационный текст в текстовом поле. Чтобы увидеть это, нужно создать фильм из трех кадров с текстовым полем `msgOutput` на собственном слое и кодом примера 4.3 в слое *scripts*.

*Пример 4.3. Анимация с помощью регистра символов*

```
// КОД В КАДРЕ 1
var i = 0;
var msg = "my what fat cheeks you have";

function caseAni () {
  var part1 = msg.slice(0, i);
  var part2 = msg.charAt(i);
  var part2 = part2.toUpperCase();
  var part3 = msg.slice(i+1, msg.length);
  msg = part1 + part2 + part3;
  msgOutput = msg;
  msg = msg.toLowerCase();
  i++;

  if (i > (msg.length - 1)) {
    i=0;
  }
}
```

```
// КОД В КАДРЕ 2
caseAni();

// КОД В КАДРЕ 3
gotoAndPlay(2);
```

# Функции кода символов

Выше, 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*, например:

```
String.fromCharCode(code_point1, code_point2, ...)
```

Вызов *fromCharCode( )* всегда начинается с `String.fromCharCode`. Затем в качестве аргументов передаются одна или более кодовых позиций создаваемых символов. В отличие от escape-последовательностей в стиле Unicode, кодовые позиции в *fromCharCode( )* задаются как десятичные, а не шестнадцатеричные числа. Если вы не знакомы с шестнадцатеричными числами, то вам, может быть, удобнее пользоваться *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) + " 2001";
```

При экспорте файлов *.swf*, относящихся к Flash 4, нужно использовать более старые функции Flash 4 для создания символов: *chr( )* и *mbchr( )*. Хотя эти функции поддерживаются с целью обратной совместимости, во Flash 5 предпочтительнее использовать *fromCharCode( )*.

### Функция charCodeAt()

Чтобы определить кодовую позицию любого символа строки, используется функция *charCodeAt( )*, имеющая следующий формат:

```
string.charCodeAt(index)
```

где *string* является любым литеральным строковым значением или идентификатором, содержащим строку, а *index* является позицией исследуемого символа. Функция *charCodeAt( )* возвращает десятичное целое, соответствующее в Unicode кодовой позиции символа с указанным индексом. 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( )* используется для работы со строками, содержащими символы, которые нельзя непосредственно ввести с клавиатуры. Например, в следующем коде проверяется, является ли символ знаком copyright:

```
msg = String.fromCharCode(169) + " 2000";
if (msg.charCodeAt(0) == 169) {
  trace("The first character of msg is a copyright symbol.");
}
```

При экспорте файлов *.swf*, относящихся к Flash 4, нужно использовать более старые функции кодовой позиции Flash 4: *ord( )* и *mbord( )*. Хотя эти функции поддерживаются с целью обратной совместимости, во Flash 5 предпочтительнее использовать *charCodeAt( )*.

## Выполнение кода в строке с помощью eval

В ActionScript функция *eval( )* преобразует строку в идентификатор. Но чтобы до конца понять функцию ActionScript *eval( )*, нужно понять, как работает аналогичная функция *eval( )* JavaScript. В JavaScript *eval( )* является встроенной функцией 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. Допустим, например, что у вас есть десять функций, последовательно названных: *func1*, *func2*, *func3*, ..., *func10*. Можно выполнить эти функции с помощью 10 вызовов функций:

```
func1();
func2();
func3();
// и так далее...
```

Но можно выполнить их более удобным образом с помощью функции *eval( )* в цикле, например:

```
for (i = 1; i <= 10; i++){
  eval("func" + i + "()");
}
```

Функция *eval( )* ActionScript поддерживает малую часть возможностей родственной ей функции в JavaScript: она работает только тогда, когда ее аргумент является идентификатором. Поэтому функция *eval( )* ActionScript может извлекать только данные, связанные с указанным идентификатором, например:

```
var num = 1;
var person = "Eugene";
trace (eval("person" + num));  // Выводит: "Eugene"
```

Даже в этом урезанном виде *eval( )* весьма полезна. Вот как мы динамически создаем в цикле клипы. Мы помещаем клипы в массив, используя *eval( )* для обращения к ним:

```
for (var i = 0; i < 10; i++) {
  duplicateMovieClip("ballParent", "ball" + i, i);
  balls[i] = eval("ball" + i);
}
```

Учтите, однако, что *eval( )* может потребовать большой вычислительной мощности. В более требовательных сценариях лучше использовать оператор доступа к массиву для генерации динамических ссылок на клипы. Например:

```
duplicateMovieClip("ballParent", "ball" + i , i);
balls[ballCount] = _root ["ball" + i];
```

Во Flash 4 функция *eval( )* щедро использовалась для моделирования массивов путем динамической генерации имен переменных и ссылок. Этот прием не является рекомендованным или обязательным во Flash 5

благодаря встроенной поддержке Flash 5 массивов. Дополнительные подробности см. в разделе «Создание переменных с динамическими именами» в главе 2 «Переменные».

## Сравнение строковых операторов и функций Flash 4 и Flash 5

При описании строковых операторов и функций мы рассматривали эквивалентные приемы для Flash 4. При использовании Flash 5 с целью создания фильмов для Flash 4 следует всюду использовать строковые операторы и функции Flash 4. Но если фильм создается для Flash 5, то следует использовать модные операторы Flash 5. Если вы привыкли к синтаксису Flash 4, его эквиваленты для Flash 5 можно найти в табл. 4.2.

*Таблица 4.2. Операторы и функции Flash 4 и их эквиваленты во Flash 5*

| Синтаксис Flash 4 | Синтаксис Flash 5 | Описание |
|---|---|---|
| `""` | `""` или `''` | Строковый литерал |
| `&` | `+` (или *add* для обратной совместимости) | Оператор конкатенации строк |
| `eq` | `==` | Оператор равенства |
| `ge` | `>=` | Сравнение «больше или равно» |
| `gt` | `>` | Сравнение «больше» |
| `le` | `<=` | Сравнение «меньше или равно» |
| `lt` | `<` | Сравнение «меньше» |
| `ne` | `!=` | Оператор неравенства |
| *chr( )* или *mbchr( )* | *fromCharCode( )** | Создает символ из числового кода |
| *length( )* или *mblength( )* | `length`* | Функция во Flash 4, свойство во Flash 5; дает число символов в строке |
| *mbsubstring( )* | *substr( )* | Извлекает из строки последовательность символов |
| *ord( )* или *mbord( )* | *charCodeAt( )** | Возвращает кодовую позицию заданного символа |
| *substring( )* | *substr( )** | Извлекает из строки последовательность символов |

* Так как все строковые операции и функции Flash 5 работают с многобайтовыми символами, во Flash 5 нельзя принудительно использовать однобайтовые операции, как во Flash 4. Например, функция *fromCharCode( )* ближе всего к функции *chr( )* во Flash 5. То же относится к *mblength( )* и `length`, *mbsubstring( )* и *substr( )*, *mbord( )* и *charCodeAt( )*.

# 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 использоваться в качестве имен переменных или идентификаторов.

Булевы значения используются 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, какого объема повреждения должны быть осуществлены, когда выстрел достигает 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 присвоить переменной значение `false`:

```
shipHasDoubleShots = false;
```

В результате большее выражение *shipHasDoubleShots* == *true* получит значение `false`, и при поражении цели выстрелом будет выполняться сценарий выполнения разрушений с одинарной мощностью.

Все операторы сравнения выражают свои результаты в булевых величинах. Когда мы спрашиваем: «Совпадает ли то, что ввел пользователь, с истинным паролем?», ответ дается как булево значение:

```
// userGuess == password возвращает true или false
if (userGuess == password) {
  gotoAndStop("secretContent");
}
```

А если мы спрашиваем: «Повернут ли этот клип более чем на 90 градусов?», то ответ снова будет булев:

```
// myClip._rotation > 90 возвращает true или false
if (myClip._rotation > 90) {
//  Обесцветить клип, если он повернут более чем на 90 градусов
```

```
  myClip._alpha = 50;
}
```

Многие внутренние свойства и методы ActionScript описывают среду фильмов Flash в булевых терминах. Например, если мы спросим: «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 формулировать сложные логические выражения с помощью булевых операторов.

## Использование булевых выражений для создания предзагрузчика

Давайте рассмотрим прикладной пример с булевыми выражениями. Допустим, у нас есть документ с 500 кадрами и богатым содержанием. Начинает нашу открывающую последовательность кадр 20 с меткой `intro`. Мы поместим в кадр 2 главной временной диаграммы фильма следующий код:

```
if (_framesloaded >= _totalframes) {
  gotoAndPlay("intro");
} else {
  gotoAndPlay(1);
}
```

При воспроизведении фильма воспроизводящая головка входит в кадр 2. Интерпретатор ActionScript доходит до условного предложения и вычисляет булево выражение *_framesloaded >= _totalframes*. Пока фильм еще загружается, `_framesloaded` меньше количества кадров в фильме (`_totalframes`). Если `_framesloaded` меньше или равно `_totalframes`, то выражение *_framesloaded >= _totalframes* имеет значение `false`. Поэтому предложение *gotoAndPlay("intro")* пропускается и вместо него выполняется *gotoAndPlay(1)*. Предложение *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 *_framesloaded >= _totalframes* не станет `true` (т. е. пока не будут загружены все кадры). В этот момент будет выполнено предложение *gotoAndPlay("intro")*, которое отправляет воспроизводящую головку на метку `intro`. Оттуда можно спокойно начать наш фильм, потому что все кадры загружены.

Ура! Вы создали предварительный загрузчик на основе булева выражения. Значительно больше об условных предложениях и управлении фильмами с помощью булевых выражений мы узнаем в главе 7.

# Тип undefined

Большая часть изученных до сих пор типов данных использовалась для хранения и обработки информации. Тип данных *undefined* имеет более узкое применение: он используется, чтобы проверять, 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 предыдущее предложение означает:

```
var velocity = undefined;
```

Чтобы проверить, есть ли у переменной значение, можно сравнить переменную с `undefined`, например:

```
if (myVariable != undefined) {
  // myVariable имеет значение, поэтому продолжить выполнение...
}
```

Обратите внимание, что значение `undefined` при использовании в качестве строки преобразуется в пустую строку. Например, если `firstName` является `undefined`, то следующее предложение *trace( )* выведет `""` (пустую строку):

```
var firstName;
trace(firstName);  // Не выводит ничего (пустую строку)
```

Этот же код в JavaScript вывел бы вместо пустой строки строку «undefined». ActionScript преобразует `undefined` в `""` в целях обратной совместимости.

Так как типа *undefined* во Flash 4 ActionScript не было, многие программы Flash 4 использовали пустую строку для проверки того, содержит ли переменная используемое значение. Часто встречался такой код:

```
if (myVar eq "") {
  // Пока ничего не делать: myVar не определена
}
```

Если бы Flash 5 преобразовывал `undefined` в строковом контексте во что-либо отличное от `""`, то такой старый код нарушал бы работу проигрывателя Flash 5.

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* используется для представления отсутствия данных и имеет только одно допустимое значение – элементарное значение `null`. Это значение 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 допустимого числа, строки, булева значения, массива или объекта.

Обратите внимание, что при сравнении `null` может быть равным только самому себе и `undefined`:

```
null == undefined;   // true
null == null;        // true
```

# Вперед!

Мы уже изучили значительно больше, чем вам может казаться. Мы построили прочный фундамент, который позволит разобраться в более сложных темах, ожидающих нас в следующих главах. Если вы пока не овладели всеми деталями, не беспокойтесь. Когда это понадобится, вы вспомните то, чему учились. Сохраняйте свою 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*) является символом или ключевым словом, обрабатывающим, объединяющим или преобразующим данные. Если вы – начинающий программист, то заметите, что некоторые математические операторы, такие как + (сложение) и – (вычитание), вам хорошо знакомы. В других случаях вам понадобится изучить особый синтаксис программирования, даже если известны понятия. Например, чтобы перемножить два числа, ActionScript использует символ `*` (оператор умножения) вместо `X`, который обычно используется в средней школе. Например, вот 5, умноженное на 6:

```
5 * 6;
```

## Общие характеристики операторов

Хотя у каждого оператора своя специальная задача, есть ряд характеристик, общих для всех операторов. Прежде чем рассматривать операторы отдельно, посмотрим, как они ведут себя в целом.

### Операторы и выражения

Операторы выполняют некоторое действие, используя передаваемые им значения данных (*операнды*). Например, в операции `5 * 6` числа 5 и 6 являются *операндами* оператора умножения (`*`). Операнды могут быть любыми выражениями, например:

```
player1score + bonusScore;              // Операнды являются переменными
(x + y) - (Math.PI * radius * radius); // Операнды являются сложными
                                        // выражениями
```

Обратите внимание, что во втором примере левый и правый операнды оператора «–» являются выражениями, которые сами содержат другие операции. Можно использовать сложные выражения для создания еще больших выражений, например:

```
((x + y) - (Math.PI * radius * radius)) / 2  // Разделить все на 2
```

Если выражения становятся очень большими, можно использовать переменные для хранения промежуточных результатов, что и удобнее, и понятнее. 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*, с двумя – *бинарными*, а операторы с тремя – *тернарными*. Для наших задач мы будем группировать операторы не по количеству принимаемых ими операндов, а по производимым действиям.

## Приоритеты операторов

*Приоритеты* операторов определяют, какая операция в выражении с несколькими операторами выполняется раньше других. 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 `+`. Если возникают сомнения, или при 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!
```

Табл. 5.1 показывает приоритеты всех операторов. Первыми выполняются операторы с наивысшим приоритетом (вверху таблицы). Операторы с одинаковым приоритетом выполняются слева направо.

*Таблица 5.1. Ассоциативность и приоритет операторов ActionScript*

| Оператор | Приоритет | Ассоциативность | Описание |
|---|---|---|---|
| x++ | 16 | слева направо | постфиксное инкрементирование |
| x-- | 16 | слева направо | постфиксное декрементирование |
| . | 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 | 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 (с остатком) |
| + | 12 | слева направо | сложение или конкатенация строк |
| – | 12 | слева направо | вычитание |
| << | 11 | слева направо | поразрядный сдвиг влево |
| >> | 11 | слева направо | поразрядный сдвиг вправо со знаком |

*Таблица 5.1 (продолжение)*

| Оператор | Приоритет | Ассоциативность | Описание |
|---|---|---|---|
| >>> | 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 |
| == | 9 | слева направо | равенство |
| != | 9 | слева направо | не равно |
| & | 8 | слева направо | поразрядное И |
| ^ | 7 | слева направо | поразрядное исключающее ИЛИ |
| \| | 6 | слева направо | поразрядное ИЛИ |
| && | 5 | слева направо | логическое И |
| \|\| | 4 | слева направо | логическое 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 & с присваиванием |
| ^= | 2 | справа налево | поразрядное XOR с присваиванием |
| \|= | 2 | справа налево | поразрядное ИЛИ с присваиванием |
| , | 1 | слева направо | запятая |

## Ассоциативность операторов

Как мы только что узнали, приоритет операторов указывает на порядок их действия: те операторы, которые имеют более высокий приоритет, выполняются раньше имеющих более низкий приоритет. Но что происходит, когда есть несколько операторов с одинаковым приоритетом? В этом случае применяются правила *ассоциативности операторов*, указывающие на направление, в котором 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 / являются ассоциативными слева, поэтому первой выполняется стоящая слева операция * над значениями b и c. Этот пример эквивалентен следующему:

```
a = (b * c) / d
```

Напротив, оператор = (присваивания) является ассоциативным 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 выражению:

```
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. На протяжении этой главы мы будем отмечать случаи, в которых ассоциативность является частым 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 используется с числовыми операндами, и конкатенацию, если используется со строковыми операндами. Если операнды имеют разные или неверные типы, ActionScript осуществляет преобразование типов согласно правилам, описанным в главе 3 «Данные и типы данных», что может оказать серьезное влияние на ваш код.

# 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 нужно поместить значение. *expression* представляет собой значение (т. е. данные), которое нужно сохранить. Например:

```
x = 4;                  // Присваивает 4 переменной x
x = y;                  // Присваивает значение y переменной  x
name = "dj duong";      // Присваивает строку переменной name
products[3] = "Flash"   // Присваивает строку 4-му элементу products

// Присваивает число свойству area объекта square
square.area = square.width * 2;
```

Можно одновременно осуществить несколько присваиваний, например:

```
x = y = 4;  // Set both x and y to 4
```

Помните, что операции присваивания ассоциативны справа налево, поэтому сначала 4 присваивается `y`, затем значение `y` (которое теперь равно 4) присваивается `x`.

## Сочетание операций с присваиванием

Операции присваивания часто используются для задания нового значения переменной, 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
```

ActionScript поддерживает сокращенную запись присваивания, называемую *составным присваиванием* (*compound assignment*), которая объединяет такие операторы, как +, – , /, с оператором присваивания, образуя единую операцию «вычислить во время присваивания». Чтобы объединить сложение с присваиванием, используется +=. Деление с присваиванием объединяется 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 используется хотя бы один строковый операнд, он осуществляет конкатенацию строк, а не математическое сложение.

Если попытка преобразовать нечисловой операнд в число оказывается неудачной, операнду присваивается особое числовое значение `NaN`. Это приводит к тому, что операция в целом дает значение `NaN`. Подробности преобразования в число можно увидеть в табл. 3.1.

## Сложение

Оператор *сложения* возвращает сумму двух своих операндов:

```
operand1 + operand2
```

Чтобы возвращаемый математический результат имел смысл, операндами + должны быть выражения, возвращающие численные значения, например:

```
234 + 5          // Returns 239
(2 * 3 * 4) + 5  // Returns 29
```

Оператор сложения уникален среди арифметических операторов тем, что если один или оба его операнда являются строками, он выполняет конкатенацию строк. Смотрите раздел «Соединение строк» в главе 4 «Элементарные типы данных».

## Инкрементирование

Удобной разновидностью сложения является оператор *инкрементирования*, который принимает один операнд и просто 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 (пространный синтаксис)
x++;        // Прибавить 1: x теперь 3
++x;        // Прибавить 1: x теперь 4
```

При самостоятельном использовании разницы между постфиксным и префиксным инкрементированием нет, хотя обычно чаще используется постфиксное инкрементирование.

Однако при использовании в больших выражениях постфиксное и префиксное инкрементирование ведут себя по-разному: префиксное инкрементирование добавляет 1 к операнду *operand* и возвращает *operand + 1*, а постфиксное инкрементирование добавляет 1 к *operand*, но возвращает значение самого *operand*, а не *operand + 1*:

```
var x = 1;
// Постфиксное инкрементирование: переменной y присваивается 1, затем x
// увеличивается до 2
var y = x++;

var x = 1;
// Постфиксное инкрементирование: сначала увеличивается x, поэтому
// переменной y присваивается 2
var y = ++x;
```

Мы снова обратимся к операторам инкрементирования в главе 8 «Предложения цикла».

## Вычитание

Оператор *вычитания* вычитает второй операнд из первого. Его общий вид такой:

```
operand1 - operand2
```

Операнды могут быть любыми допустимыми выражениями. Если какой-либо 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 (т. е. положительной) разности между двумя числами осуществляется с помощью метода *Math. abs( )* (см. часть III «Справочник по языку»).

## Декрементирование

Оператор *декрементирования* аналогичен оператору инкрементирования, но не прибавляет единицу к текущему значению операнда, а вычитает из него единицу. Подобно инкрементированию, оператор де-

крементирования имеет два общих формата, называемые *префиксное декрементирование* и *постфиксное декрементирование*:

```
--operand  // префиксное декрементирование
operand--  // постфиксное декрементирование
```

В обоих форматах он используется 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 используемого декрементирования имеет значение только в составе более крупного выражения:

```
var x = 10;
var y;
x = x - 1;     // x теперь 9
x--;           // x теперь 8
--x;           // x теперь 7
y = --x;       // y теперь 6, x теперь 6
y = x--;       // y по-прежнему 6, x теперь 5
```

## Умножение

Оператор *умножения* перемножает два числовых операнда и возвращает результат (т. е. *произведение*). Общий вид умножения такой:

```
operand1 * operand2
```

Операнды могут быть любыми допустимыми выражениями. Символ * используется вместо символа X («умножить») из обычной математики. Если какой-либо из операндов имеет нечисловой тип и преобразование в действительное число оказывается невозможным, операция дает в результате NaN:

```
6 * 5  // Возвращает 30
```

## Деление

Оператор *деления* делит первый операнд (*делимое*) на второй операнд (*делитель*) и возвращает результат (*частное*). Общий вид деления следующий:

```
operand1 / operand2
```

Операнды должны быть допустимыми числовыми выражениями. Символ / используется вместо символа ÷ из обычной математики. Если какой-либо из операндов имеет нечисловой тип и преобразование в действительное число оказывается невозможным, операция дает в результате NaN. Если получается дробный результат, то частное является 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 Director's Lingo, при делении возвращается целое число, если хотя бы один из операндов не является числом с плавающей точкой.

Если делитель равен нулю, то результат деления равен:

- `NaN`, если делимое равно нулю или не является числом
- `Infinity`, если делимое является положительным числом
- `-Infinity`, если делимое является отрицательным числом

Несколько примеров:

```
trace (0/0);          // NaN
trace ("a"/0);        // NaN
trace (1/0);          // Infinity
trace (-1/0);         // -Infinity
```

Заметьте, что если делимое – `Infinity`, то результат равен:

- `NaN`, если делитель: `Infinity`, `-Infinity` или не является числом
- `Infinity`, если делитель: 0 или конечное положительное число
- `-Infinity`, если делитель: отрицательное конечное число

Несколько примеров:

```
trace (Infinity/Infinity);          // NaN
trace (Infinity/-Infinity);         // NaN
trace (Infinity/"a");               // NaN
trace (Infinity/0);                 // Infinity
trace (Infinity/1);                 // Infinity
trace (Infinity/1000);              // Infinity
trace (Infinity/-1);                // -Infinity
trace (-Infinity/2);                // -Infinity
trace (-Infinity/-2);               // Infinity
```

Наконец, заметим, что любое конечное число при делении на `Infinity` дает 0.

```
trace (0/Infinity);               // 0
trace (1/Infinity);               // 0
trace (-10000/Infinity);          // 0
```

При малейшей возможности для делителя равняться нулю следует перед выполнением деления проверить его значение:

```
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 деления по модулю следующий:

```
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 действительное число оказывается невозможным, операция дает результат `NaN`.

Если число четное, то его деление по модулю 2 дает ноль. Чтобы проверить четность числа, можно использовать прием, показанный в примере 5.1.

*Пример 5.1. Использование деления по модулю для проверки четности чисел*

```
var x = 3;
if (x%2 == 0) {
  trace("x is even");
} else {
  trace("x is odd");
}
```

## Унарное отрицание

Оператор *унарного отрицания* принимает только один операнд. Он меняет знак операнда на противоположный (т. е. положительное число становится 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
}
```

# Операторы равенства и неравенства

Оператор *равенства* (==) используется для проверки, не одинаковые ли значения имеют два данных 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 значение `false`. Например:

```
var x = 2;
x == 1     // false
x == 2     // true
```

Оператор равенства записывается с помощью двух знаков «равно» (==) подряд. Он определяет, равны ли два выражения. Не следует путать его с оператором присваивания (=), который используется для присваивания переменной нового значения.

Рассмотрим такой пример:

```
if (x = 5) {
  trace ("x равен 5")
}
```

В этом примере *не* проверяется, равен ли `x` пяти. Вместо этого переменной `x` *присваивается* значение 5. Правильное выражение должно быть таким:

```
// 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) |
|---|---|
| Number | Если *operand1* – такое же число, как *operand2*, результат `true`. Если оба операнда `+Infinity` или оба `-Infinity`, результат `true`. Если оба операнда `-0` или `+0`, результат `true`. Для всех остальных комбинаций, в том числе если *один* или *оба* операнда `NaN`, результат `false`:<br>`1 == 4 // false`<br>`4 == 4 // true`<br>`NaN == NaN // false`<br>`+Infinity = -Infinity // false` |
| String* | Выполняет сравнение строк с учетом регистра. Если *operand1* и *operand2* суть строки одинаковой длины, содержащие одинаковую последовательность символов, результат `true`; в противном случае, результат `false`:<br>`"Flash" == "Flash" // true`<br>`"O'Reilly" == "O Reilly" // false`<br>`"Moock" == "moock" // false ("m" и "M" – разные символы)` |
| Булев | Если оба операнда `true` или оба операнда `false`, результат `true`; в противном случае, результат `false`:<br>`true == true // true`<br>`false == false // true`<br>`true == false // false` |
| `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» |

* В Flash 4 оператором равенства строк был `eq`. Хотя `eq` поддерживается во Flash 5 для обратной совместимости, использовать его не рекомендуется, если только не производится экспорт в формат *.swf* Flash 4.

## Равенство сложных типов данных

Так как переменные, содержащие составные данные (объекты, массивы, функции или клипы), хранят *ссылки* на данные, а не сами данные, то 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.

В последующих примерах иллюстрируется, 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 являются двумя разными массивами. Поэтому ссылки являются разными, и сравнение дает результат false:

```
nameList1 = ["Linkovich", "Harris", "Sadler"];
nameList2 = ["Linkovich", "Harris", "Sadler"];
nameList1 == nameList2  // false
```

В этом примере cities и canadianCities ссылаются на один и тот же массив:

```
canadianCities = ["Toronto","Montreal","Vancouver"];
cities = canadianCities;
cities == canadianCities  // true
```

В очередном примере myFirstBall и mySecondBall имеют один и тот же конструктор (т. е. оба объекта порождены одним и тем же классом), но они существуют как отдельные (не равные) экземпляры:

```
myFirstBall = new Ball();
mySecondBall = new Ball();
myFirstBall == mySecondBall  // false
```

Поэтому говорят, что проверка равенства для сложных типов производится по ссылке, а не по значению. Дополнительно о разнице см. в разделе «Копирование, сравнение и передача данных» в главе 15 «Более сложные темы».

Чтобы сделать 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 рассмотрим, что происходит, когда сравниваются операнды разных типов, например, строка и число:

```
"asdf" == 13;
```

Если типы операндов разные, интерпретатор перед сравнением осуществляет 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 – число, а другой – строка, 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 является объектом, 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 получено допустимого результата.

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, используется 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 лучше, чем «если `x` равен `y`, не делать ничего, в противном случае сделать это», как показано далее в примере 5.2. Оператор 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 типов, что и оператор равенства, и всегда возвращает противоположный результат, в том числе при использовании в качестве операнда `NaN`:

```
NaN != 7    // true
NaN != NaN  // true!
```

В некоторых языках, в том числе Flash4 ActionScript, в качестве оператора неравенства используется оператор `<>`. Смотрите также оператор отрицания (`!`), обсуждаемый ниже.

## Частые случаи применения операций равенства

Мы будем часто использовать операции равенства для создания булевых выражений в условных предложениях или для присваивания булева значения переменной. В примере 5.2 показаны обе ситуации и продемонстрированы операторы `!=` и `==` в действии.

*Пример 5.2. Использование 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("Используйте IE4 или более поздний.");
}
```

Опытные программисты быстро укажут, что можно сократить *if(isWin==true)* до *if(isWin)*, потому что `isWin` содержит булево значение, которое и нужно предложению *if*. Это совершенно верно, но мы лишимся хорошего примера использования оператора равенства. Кроме того, для новичков в программировании более пространная форма обычно понятнее. Оба подхода совершенно приемлемы. Более подробно на этой теме мы задержимся в главе 7 «Условные предложения».

# Операторы сравнения

Операторы *сравнения* (называемые также операторами *отношения*) используются для определения того, которое из двух значений стоит первым при заданном порядке. 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 работают только со строками и числами. Если два операнда сравнения являются числами, то сравнение производится математически: *5 < 10* дает `true`, *-3 < -6* дает `false` и т. д. Когда два операнда оператора сравнения являются строками, сравнение производится в соответствии с кодовыми позициями символов, как показано в приложении B «Набор символов Latin 1 и коды клавиш». Подробности сравнения строк см. в разделе «Сравнение строк» главы 4.

Данные в операции сравнения, не являющиеся строкой или числом, интерпретатор пытается преобразовать в данные типа *string* или *number*. Действие преобразования типа в операциях сравнения мы рассмотрим после обсуждения самих операторов сравнения.

## Оператор «меньше»

Оператор *меньше* имеет общий вид:

```
operand1 < operand2
```

Когда операнды являются числами, 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 являются строками, оператор «меньше» возвращает булево значение `true`, если *operand1* предшествует *operand2* «по алфавиту» (см. приложение B); 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" меньше "i"
```

## Оператор «больше»

Оператор *больше* имеет общий вид:

```
operand1 > operand2
```

Когда операнды являются числами, 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* «по алфавиту» (см. приложение В); 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 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 различие с 5 < 5
-3 <= -6     // false
-6 <= -3     // true
```

Когда операнды являются строками, оператор «меньше» возвращает булево значение `true`, если *operand1* предшествует *operand2* «по алфавиту» или операнды идентичны согласно правилам, описанным в разделе «Сравнение строк» главы 4; в остальных случаях 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" меньше "i"
```

Обратите внимание, что оператор <= пишется со знаком равенства после знака «меньше». Оператор =< не верен.

## Оператор «больше или равно»

Оператор *больше или равно* имеет 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* «по алфавиту» или операнды идентичны согласно правилам, описанным в разделе «Сравнение строк» главы 4; в остальных случаях 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" меньше "i"
```

Обратите внимание, что оператор >= пишется со знаком равенства после знака «больше». Оператор => не верен.

## Операции сравнения и преобразование типов данных

Чаще всего операторы сравнения используются для сравнения чисел. Поэтому преобразования типов, вызываемые операторами сравнения, отдают предпочтение числам. Если два операнда в каком-либо операторе сравнения принадлежат разным типам или ни один из операндов не является строкой или числом, предпринимается попытка преобразования типа согласно следующей процедуре:

1. Если оба операнда – числа, сравнить операнды математически и возвратить результат. Если одно или оба числа являются `NaN`, то результат сравнения – `false`, кроме случая оператора `!=`.
2. Если оба операнда – строки, сравнить операнды в лексикографическом порядке, используя кодовые позиции символов, показанные в приложении B, и возвратить результат.

3. Если один операнд является числом, а другой – строкой, то преобразовать строку в число и вернуться на шаг 1.
4. Если один из операндов является булевым значением, `null` или `undefined`, преобразовать операнд в число и вернуться на шаг 1.
5. Если один из операндов является объектом, вызвать его метод *valueOf( )* для преобразования объекта в значение элементарного типа и вернуться на шаг 1. Если вызов метода *valueOf( )* оказывается неудачным или не возвращает значения элементарного типа, возвратить `false`.
6. Возвратить `false`.

Обратите внимание, что преобразования типа, выполняемые во время сравнения, не изменяют значение или тип данных исходного объекта. Результаты временного преобразования отбрасываются, как только завершается 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 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!
```

В приведенном ниже примере, несмотря на то что кодовой позицией буквы «A» является 65, преобразование «A» в число дает `NaN`, а это значит, что выражение в целом имеет значение `false`. Для получения кодовой позиции строки используйте функцию *charCodeAt( )*:

```
"A" <= 9999                // false
"A".charCodeAt(0) < 9999   // true
```

# Строковые операторы

Во Flash 4 при использовании строковых данных требовались особые строковые операторы – конкатенации (`&`), равенства (`eq`), неравенства (`ne`) и сравнения (`lt`, `gt`, `le`, `ge`). Во Flash 5 и выше специальные строковые операторы являются устаревшими (но поддерживаются). Для сравнения строк используйте операторы `==`, `!=`, `<`, `>`, `<=` и `>=`, если только вы не осуществляете экспорт в формат Flash 4 *.swf*; в этом случае необходимо использовать старые операторы (`eq`, `ne`, `lt`, `gt`, `le`, `ge`). Используйте `+` для конкатенации строк, если только не осуществляете экс-

порт в формат Flash 4; в этом случае необходимо использовать `add` вместо `&` (т. к. во Flash 5 и выше `&` является оператором поразрядного И).

Дополнительные сведения и список эквивалентных строковых операций Flash 4 и Flash 5 можно найти в разделе «Работа со строками» и табл. 4.2 главы 4.

# Логические операторы

В главе 4 было показано, как принимать логические решения с использованием булевых значений. Рассматривавшиеся там решения основывались на одиночных факторах: если фильм не загружен, не начинать его воспроизведение; если у космического корабля оружие удвоенной мощности, увеличить разрушения, наносимые его выстрелами, и так далее. Не всегда логика программ так проста. Часто требуется рассматривать множественные факторы в логике *ветвления* (т. е. принятия решений).

Предположим, например, что мы создаем персональный Flash-сайт, который требует регистрации пользователей. Если пользователь регистрируется как гость, предлагаемое ему содержание может быть ограничено. При авторизации на сайте зарегистрированного пользователя устанавливается переменная, которая на протяжении всего фильма указывает, что для данного пользователя доступ к просмотру содержимого ограничен:

```
var userStatus = "registered";
```

Чтобы определить, разрешен ли доступ в привилегированную область, используем такую булеву проверку:

```
if (userStatus == "registered") {
  // okay, впустить...
}
```

Допустим, что нужно 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 гости, 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 допущенных пользователей? Это можно сделать грубым способом, дублируя участки нашего кода:

```
if (userStatus == "registered") {
  // Выполнить код для легальных пользователей...
}
```

```
if (userStatus == "specialGuest") {
  // Выполнить то же самый код для легальных пользователей...
}
```

Очевидно, сопровождение такого кода будет связано с сильной головной болью. Любые сложности, вносимые в программу, будут удваиваться, и мы столкнемся с серьезными проблемами контроля версий.

Лучше было бы произвести одну булеву проверку, которая сказала бы: «Если пользователь зарегистрирован *или* пользователь является особым гостем, то предоставить доступ.» Образовывать такие составные логические конструкции можно с помощью булевых *логических операторов*. Вот как логический оператор ИЛИ (||) позволяет в одном выражении проверить состояние нескольких булевых операций:

```
if (userStatus == "registered" || userStatus == "specialGuest") {
  // Выполнить код для легальных пользователей...
}
```

Красиво? Иногда программирование очень элегантно. Такие отчетливые вертикальные линии ... такие гибкие скобки ...

## Логическое ИЛИ

Оператор *логическое ИЛИ* чаще всего используется для инициирования 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 клавиши Backslash (\) с изображением прерывистой вертикальной черты, которая на большинстве клавиатур находится в правой верхней области. Логическое ИЛИ имеет общий вид:

```
operand1 || operand2
```

Когда оба операнда являются булевыми выражениями, логическое ИЛИ возвращает `true`, если один из операндов `true`, и возвращает `false`, только если *оба* операнда `false`. Суммарно:

```
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 (именно, 10)
```

На практике небулевы значения, возвращаемые выражением с логическим ИЛИ, используются редко. Вместо этого результат обычно используется в условном предложении, где его применяют для принятия булева решения. Рассмотрим пример 5.3:

*Пример 5.3. Операция логического ИЛИ как проверочное выражение в условном предложении*

```
var x = 10;
var y = 15;
if (x || y) {
  trace("Один из x и y отличен от нуля");
}
```

В строке 3 мы видим логическую операцию ИЛИ $(x \| y)$ там, где ожидается булева величина в качестве проверочного выражения предложения *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 наше предложение *if* выглядит так:

```
if (10) {
  trace("Один из x и y отличен от нуля");
}
```

Но 10 является числом, а не булевой величиной. Что же происходит? Предложение *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 предложение *trace( )* в фигурных скобках.

Обратите внимание, что если первый операнд в операции логического ИЛИ разрешается в `true`, нет необходимости и потому неэффективно вычислять второй операнд. Поэтому ActionScript вычисляет второй операнд, только если первый операнд разрешается в `false`. Это обстоятельство полезно в тех случаях, когда вам нужно, чтобы второй операнд вычислялся только тогда, когда первый операнд разрешается в `false`. В следующем примере мы проверяем, не вышло ли число за допустимые границы. Если число слишком маленькое, не нужно во второй проверке выяснять, не является ли оно слишком большим:

```
if (xPos < 0 || xPos > 100) {
  trace ("xPos не попадает в диапазон от 0 до 100 включительно.");
}
```

## Логическое И

Как и оператор логического ИЛИ, оператор *логического И* в основном используется для условного выполнения блока кода, но только если удовлетворены оба условия. Оператор логического И имеет 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`. Суммарно:

```
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 использование оператора логического И на двух реальных примерах. В примере 5.4 мы выполняем предложение *trace( )* только тогда, когда обе переменные больше 50.

*Пример 5.4. Операция логического И как проверочное выражение в условном предложении*

```
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 предложение *trace( )*.

В примере 5.5 мы возвращаемся к нашему примеру сайта, в котором доступ предоставлялся зарегистрированным пользователям и особым гостям. Но на этот раз мы впустим пользователей, только если дата отлична от 1 января. Обратите внимание на использование объекта *Date* для определения текущей даты. Заметьте, что *getMonth( )* для января возвращает 0.

*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...предложения if могут для
// улучшения читаемости разбиваться на несколько строк
if ( ((userStatus == "registered") || (userStatus == "specialGuest"))
    && (month + day) > 1) {
  // Впустить пользователя...
}
```

Формальное поведение оператора И вполне аналогично действию оператора ИЛИ. Сначала *operand1* преобразуется в булево значение. Если результатом преобразования является `false`, возвращается значение *operand1*. Если результат этого преобразования `true`, то возвращается значение *operand2*.

Если первый операнд в операции логического ИЛИ разрешается в `false`, нет необходимости и потому неэффективно вычислять второй операнд. Поэтому 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 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 противоположное ему.

Как и оператор «не равно» (`!=`), оператор логического отрицания удобно использовать для проверки именно того, что операнд *не является* чем-то. Вспомните пример 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 сегодняшний день *является* первым 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 $month+day > 1$, использовавшейся в примере 5.5.

Другое типичное использование оператора логического отрицания состоит в *переключении* переменной с `true` на `false` и наоборот. Допустим, например, что имеется одна кнопка для включения и выключения звука. Можно использовать такой код:

```
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 `!` используется также в операторе (`!=`). В качестве программной *лексемы* (т. е. знака) символ `!` обычно означает *не*, или *противоположный*. Он не имеет отношения к символу «!», используемому для обозначения факториала в обычных математических обозначениях.

## Логические выражения и значения данных

Использование логических операторов для вычисления булева результата аналогично использованию арифметических операторов для получения числового результата. Например, следующие четыре операн-

да 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 условного предложения).

# Оператор группирования

Помимо использования в вызовах функций, круглые скобки `()` могут использоваться для группирования высказываний в коде, чтобы преодолеть приоритеты различных операторов. Круглые скобки требуются также в некоторых предложениях, в частности, в предложении *if*. Круглые скобки имеют общий вид:

```
(expression)
```

Заключенное в скобки выражение *expression* вычисляется, и возвращается его результат. Вот несколько примеров:

```
if (x == y) {                  // Необходимый синтаксис в предложении 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 используемый оператор *запятая* (`,`) позволяет вычислять два выражения там, где ожидается одно выражение. Он имеет общий вид:

```
operand1, operand2
```

Оба операнда должны быть допустимыми выражениями. При выполнении оператора запятой вычисляется *operand1*, затем вычисляется *operand2* и возвращается полученное значение *operand2*. На практике значение, возвращаемое оператором запятая, обычно игнорируется.

Оператор запятой используется, 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`, являются операндами оператора запятой. (Эти выражения осуществляют инициализацию `i` и `j` значениями 0 и 10.) Выражения `i++` и `j--` являются операндами второго оператора запятой. (Эти выражения вызывают инкрементирование `i` и декрементирование `j` при каждом выполнении цикла.) Оператор запятой используется для того, чтобы втиснуть несколько выражений в рамки синтаксиса цикла *for*.

*Пример 5.6. Использования оператора запятой в цикле for*

```
for (var i=0, j=10; i!=j; i++, j--) {
  trace ("i: " + i + " j: " + j);
}

// что производит такую выдачу...
 i: 0 j: 10
 i: 1 j: 9
 i: 2 j: 8
 i: 3 j: 7
 i: 4 j: 6
```

# Оператор void

Оператор *void* используется для вычисления выражения без возврата результата. Его синтаксис следующий:

```
void expression
```

где *expression* является выражением, которое нужно вычислить. В JavaScript *void* используется для вычисления выражения JavaScript в гипертекстовой ссылке так, чтобы результат не появлялся в адресной панели броузера. Такое применение *void* в ActionScript бывает редко, если вообще встречается – этот оператор включен в состав языка для совместимости с ECMA-262.

# Прочие операторы

Остальные операторы относятся к темам, рассматриваемым в других главах. Мы включим их сюда только для краткой справки, а подробнее их применение опишем в других главах.

## Поразрядные операторы

Если вы собираетесь разрабатывать большие системы, в которых любая оптимизация использования памяти, скорости расчетов и скорости передачи данных оказывает важное влияние на производительность, почитайте об операторах поразрядного действия в главе 15. В противном случае используйте булевы логические операторы, которые решают те же задачи, что и поразрядные операторы, хотя и менее оптимальным образом.

## Оператор typeof

Оператор *typeof* используется для определения типа данных выражения. Он принимает один операнд:

```
typeof operand;
```

где *operand* может быть любым допустимым выражением. Операция *typeof* возвращает строку, обозначающую тип данных вычисляемого *operand*. Дополнительные сведения см. 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* следующий:

```
new constructor
```

где *constructor* должен быть функцией, определяющей свойства вновь создаваемого объекта. См. главу 11 «Массивы» и главу 12 «Объекты и классы».

## Оператор delete

Оператор *delete* используется для удаления из сценария объекта, свойства объекта, элемента массива или переменных. Синтаксис *delete* следующий:

```
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. См. главу 11.

## 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 вид:

```
array[element]
```

где *array* – имя массива или литерал массива, а *element* – выражение, значение которого является неотрицательным целым числом, представляющим собой индекс элемента массива, к которому происходит обращение (индексация начинается с нуля).

При обращении к объекту оператор имеет вид:

```
object[property]
```

где *object* – имя объекта или литерал объекта, а *property* – любое выражение, значение которого является строкой, представляющей имя свойства объекта, к которому происходит обращение.

При использовании в левой части оператора присваивания (=) элементу или свойству присваивается новое значение, представленное в правой части присваивания:

```
var colors = new Array();     // Создать новый массив
colors[0] = "orange";         // Задать его первый элемент
colors[1] = "green";          // Задать его второй элемент

var ball = new Object();      // Создать новый объект
var myProp = "xVelocity";     // Сохранить строку в переменной
ball["radius"] = 150;         // Задать свойство radius
ball[myProp] = 10;            // Задать свойство xVelocity
                              // через myProp
```

При использовании в других местах выражение возвращает значение указанного элемента или свойства:

```
diameter = ball["radius"] * 2;  // Присваивает diameter значение 300
trace(colors[0]);               // Выводит "orange"
```

## Оператор «точка»

Оператор *точка* является основным средством 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» такой:

```
object.property
```

где *object* должен быть именем объекта или литералом объекта, а *property* является идентификатором, представляющим свойство *object*. Обратите внимание на то, что *property* ***не*** может быть произвольным выражением или строковым литералом – это должно быть имя свойства. Поскольку элементы массива нумеруются, а не именуются, оператор «точка» нельзя использовать для доступа к элементам массива.

При использовании в качестве левого операнда оператора присваивания оператор «точка» участвует в присвоении свойству нового значения:

```
var ball = new Object();
ball.radius = 150;
ball.xVelocity = 10;
```

При использовании в любом другом месте оператор «точка» возвращает значение указанного свойства объекта:

```
diameter = ball.radius;
newX = ball.xPosition + ball.xVelocity;
```

См. главу 12 «Объекты и классы» и главу 13 «Клипы».

## Условный оператор

*Условный* оператор является синтаксической конструкцией, позволяющей в сжатом виде представить простое условное предложение. Этот оператор принимает три операнда и имеет формат:

```
condition ? result_if_true : result_if_false;
```

Когда выполняется условная операция, 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*). См. главу 7.

## Оператор вызова функции

Как мы видели по функции *trace( )* и функциям обработки строк, для вызова функции используется оператор вызова функции `()`. Он имеет следующий общий вид:

```
function_name(argument_list)
```

Первый операнд, *function_name*, является именем некоторой функции и *обязан* быть идентификатором, а не выражением. Функция должна существовать, иначе интерпретатор выдаст ошибку. *argument_list* является последовательностью из нуля или более аргументов, передаве-

мых 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
myClip.play();                  // Метод клипа, без аргументов
myRectangle.area(6, 9);         // Определенный пользователем метод,
                                // два аргумента
init();                         // Определенная пользователем функция,
                                // без аргументов
```

# Вперед!

Как и во всех языках, в ActionScript есть грамматика и разные части речи. На протяжении нескольких последних глав вы познакомились с эквивалентами структуры предложения, существительных, прилагательных и союзов. Теперь перейдем к предложениям, которые, подобно глаголам, могут выполнять действия. Предложения дополняют наши возможности создавать инструкции, похожие на предложения языка.

# 6

# Предложения

В предыдущих главах мы увидели, как ActionScript хранит и обрабатывает данные. В этой главе мы узнаем, как с этими данными можно что-то делать. Мы будем давать Flash инструкции в виде *предложений* (*statements*), или фраз кода, дающих интерпретатору команду на выполнение некоторой задачи.

Для того чтобы вызвать во Flash некоторое событие – прекратить звуковоспроизведение, воспроизвести фильм, выполнить в цикле некоторый код – используются предложения. Фактически, программу ActionScript можно определить просто как перечень предложений, которые сообщают интерпретатору, выполнения каких действий мы хотим от Flash. Вот, например, полная программа ActionScript, которая состоит из четырех предложений и предписывает интерпретатору загрузить в броузер страницу:

```
var protocol = "http";                                  // Предложение 1
var domain = "www.moock.org";                           // Предложение 2
var path = "webdesign/flash/";                          // Предложение 3
getURL(protocol + "://" + domain + "/" + path);         // Предложение 4
```

Создание сценария фильма с помощью ActionScript состоит просто в прикреплении предложений к кадрам, клипам и кнопкам. Эта глава исследует синтаксическую структуру предложений и перечисляет общие категории предложений. В ней мы коснемся всех предложений ActionScript, но некоторые наиболее важные из них более подробно будут исследованы в следующих главах.

# Типы предложений

Концептуально предложения можно разбить на пять типов:

*Предложения, управляющие логикой выполнения программы*

циклы
условные предложения
ifFrameLoaded

*Предложения, объявляющие переменные*

var
set

*Предложения, объявляющие функции, а также вызывающие их и возвращающие значения*

function
function call
call
return

*Предложения для работы с объектами*

with
for … in

*Предложения, представляющие значения данных*

любое выражение (особенно выражение с побочным эффектом)

Эти неформальные категории помогут понять, какие действия можно требовать от интерпретатора с помощью предложений. На первый взгляд, список может показаться довольно ограниченным. Однако мы увидим, что существует много разновидностей условных предложений и циклов, а также тысячи вещей, которые можно делать с помощью вызовов функций. Во-первых, рассмотрим формат предложений.

# Синтаксис предложений

Предложения обычно состоят из *ключевых слов* и *выражений*. Мы уже видели предложение *var*, которое объявляет и может дополнительно инициализировать переменную. Вот пример общего синтаксиса предложения *var*:

```
var numFrames;
```

*Ключевое слово* (в данном случае *var*) идентифицирует начало предложения. Затем следует синтаксис, требуемый предложением, который в нашем примере состоит просто из имени переменной, `numFrames`. Наконец, точка с запятой отмечает конец предложения.

В ActionScript каждое предложение должно заканчиваться точкой с запятой. (Использование точки с запятой обеспечивает удобный формат, но официально не является обязательным. В главе 14 «Лексическая структура» еще будет сказано об употреблении точки с запятой.)

Некоторые предложения могут иметь несколько форматов. Например, можно использовать *var* с начальным присваиванием 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 присваиванием
```

На протяжении этой главы мы изучим специфический синтаксис каждого предложения.

## Блоки предложений

Некоторые предложения фактически содержат другие предложения, или *подпредложения* (*substatements*), в составе своего синтаксиса. Например, предложение *if* имеет такой синтаксис:

```
if (expression) substatement;
```

*substatement*, которое выполняется только если *expression* дает истинное значение true, может быть одиночным предложением, например, объявлением переменной:

```
if (x == 5) var numFrames = 2;
```

либо это может быть *ряд* предложений, объединенных в *блок предложений*:

```
if (x == 5) {
  var numFrames;
  numFrames = 10;
  play();
}
```

Как можно видеть, блок предложений представляет собой любое число предложений, расположенных на одной или более строках и заключенных в фигурные скобки:

```
{ statement1; statement2; statement3... }
```

Используя блок предложений в качестве подпредложения нашего предложения *if*, мы смогли задать несколько предложений там, где обычно ожидается только одно предложение. Очень удобно.

Всюду, где ActionScript ожидает одно предложение, можно использовать блок предложений. На самом деле, использование блока предложений иногда является даже обязательным. Например, предложение *function* всегда должно содержать блок предложений, даже если тело объявляемой функции содержит всего одно предложение:

```
function aheadTwoFrames() {
  gotoAndStop(_currentframe + 2);
}
```

Чтобы улучшить читаемость, блоки предложений обычно записываются в таком формате:

```
parent_syntax {
  substatement1;
  substatement2;
  substatement3;
}
```

где *parent_syntax* обозначает предложение, для которого мы определяем блок предложений.

Обратите внимание на то, что открывающая фигурная скобка находится в конце первой строки после *parent_syntax*. Подпредложения блока предложений помещаются каждое на своей строке с отступом два пробела. Закрывающая фигурная скобка, также на отдельной строке, помещается на одном уровне с главным исходным предложением. Подпредложения в блоках предложений должны заканчиваться точкой с запятой, но точек с запятой нет в строках с фигурными скобками.

Использование отступов является просто соглашением, а не требованием синтаксиса. Эта книга придерживается стиля, используемого редактором Flash ActionScript.

# Предложения ActionScript

Теперь, когда вы знаете, какой формат имеет типичное предложение, бегло ознакомьтесь с тем, что могут делать предложения ActionScript.

*Таблица 6.1. Предложения ActionScript*

| Предложение | Синтаксис | Использование |
|---|---|---|
| break | `break;` | Прекращает выполнение цикла |
| call | `call (`*frame*`);` | Выполняет сценарий удаленного кадра |
| continue | `continue;` | Заново начинает текущий цикл |

*Таблица 6.1 (продолжение)*

| Предложение | Синтаксис | Использование |
|---|---|---|
| do-while | `do {`<br>`  statements`<br>`} while (expression);` | Разновидность цикла *while* |
| пустое предложение | `;` | Занимает место там, где ожидается предложение, и используется с *evaluate* в режиме новичка |
| for | `for (init; test; increment) {`<br>`  statements`<br>`}` | Многократно выполняет некоторый код (цикл *for*) |
| for-in | `for (property in object) {`<br>`  statements`<br>`}` | Перечисляет свойства объекта |
| функция | `function name(parameters) {`<br>`  statements`<br>`}` | Объявляет функцию |
| if-else if-else | `if (expression) {`<br>`  statements`<br>`} else if (expression) {`<br>`  statements`<br>`} else {`<br>`  statements`<br>`}` | Выполняет некоторый код, исходя из условия или ряда условий |
| ifFrame-Loaded | `ifFrameLoaded (frame) {`<br>`  statements`<br>`}` | Выполняет некоторый код, если загружен определенный кадр; во Flash 5 устарело |
| return | `return;`<br>`return expression;` | Выходит из функции или возвращает ее значение |
| set | `set (variable, value);` | Присваивает значение динамически именуемой переменной |
| var | `var variableName;`<br>`var variableName = expression;` | Объявляет переменную и дополнительно может ее инициализировать |
| while | `while (expression) {`<br>`  statements`<br>`}` | Многократно выполняет некоторый код (цикл *while*) |
| with | `with (objectName) {`<br>`  statements`<br>`}` | Выполняет некоторый код в контексте указанного объекта |

## Циклы и условные предложения

Мы уже неофициально сталкивались несколько раз с циклами и условными предложениями. Оба эти типа предложений в совокупности обеспечивают основную долю сложной логики в программах. Циклы позволяют многократно выполнять предложения, а условные предложения позволяют выполнять предложения только при определенных обстоятельствах. Полное описание см. в главе 7 «Условные предложения» и главе 8 «Предложения цикла».

## Предложения выражений

Любое выражение может быть независимо использовано в качестве допустимого предложения, но если выражение не осуществляет никаких действий и с его результатом ничего не производится, использование его довольно бессмысленно:

```
"hi there";  // Это предложение ничего не производит
345 + 5;     // И это тоже
```

Однако некоторые выражения имеют *побочные эффекты*, которые изменяют состояние системы путем присваиваний переменным или свойствам или физического изменения окружения Flash. В следующем примере выражение изменяет значение `x`:

```
x = 345 + 5;  // Гораздо больше пользы
```

Вызов функции является наиболее мощным видом предложения в выражении. 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 на другой кадр:

```
gotoAndStop(5);  // Значение _currentframe изменилось на 5.
```

Подробнее о вызовах функций мы узнаем в разделе «Выполнение функций» главы 9 «Функции».

## Предложение var

Предложение *var* объявляет новую переменную:

```
var x;
```

В предложении *var* также можно задать начальное значение новой переменной:

```
var x = 10;
```

При использовании вне функции предложение *var* создает переменную, областью видимости которой является временная диаграмма, содержащая предложение. При использовании внутри функции предло-

жение *var* создает локальную для этой функции переменную (т. е. такую, которая уничтожается по завершении выполнения функции). Подробнее см. в главе 2 «Переменные».

## Предложение set (Set Variable)

Большинство присваиваний переменным осуществляется выражениями присваивания в виде предложения, например:

```
x = 30;
```

Однако для такого вида выражений требуется заранее знать имя переменной. Если нужно генерировать имя переменной динамически во время присваивания, можно использовать предложение *set*, имеющее следующий синтаксис:

```
set (variable, expression);
```

где *variable* является строковым выражением, дающим имя переменной, участвующей в присваивании, а *expression* – 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 предложении *set* ничего не присваивается. Вместо этого извлекается значение `y` (`"x"`), и это значение используется в качестве имени переменной, которой, собственно, 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 предложение *set* называлось *Set Variable*. В той версии ActionScript оно часто использовалось для динамического присваивания переменным, имена которых последовательно генерировались программным образом. Это позволяло программисту моделировать массивы, встроенной поддержки которых во Flash 4 ActionScript не было. Так как во Flash 5 массивы включены, *set* используется редко. Дополнительные сведения о моделировании массивов с помощью *set* можно найти в главе 2.

## Предложение function

Подобно тому как предложение *var* используется для *объявления* (т. е. создания) переменных, предложение *function* используется для

объявления функций. Предложение *function* имеет следующий синтаксис:

```
function funcName (param1, param2, param3,...paramn) {
  statements
}
```

Ключевое слово `function` начинает предложение; *funcName* является именем объявляемой функции; *param1* ... *paramn* определяют параметры, необходимые для выполнения функции; *statements* – список из одного или более предложений, которые будут выполнены при вызове функции.

Предложение *function* создает функцию, которая может использоваться в дальнейшем, но не выполняет ее. Для выполнения функции ее имя указывается в предложении *вызова функции*.

## Предложения вызова функции

Предложение вызова функции выполняет встроенную или определенную пользователем функцию, используя имя этой функции и входные данные, необходимые для работы функции. Термины *вызвать* (*call, invoke*) и *выполнить* (*run*) часто используются взаимозаменяемо для обозначения использования имени функции для ее выполнения. Предложение вызова функции имеет следующий общий синтаксис:

```
funcName (arg1, arg2, ... argn);
```

где *funcName* является именем функции, которую нужно выполнить, а *arg1* ... *argn* является списком *аргументов* (т. е. входных значений), которые функция предполагает получить при своем выполнении.

Предложение вызова функции является исключительно мощным и фундаментальным инструментом: это основной способ управления фильмами Flash. Когда нам нужно 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 запуска их методов:

```
circle.area();
today.getDate();
```

Поскольку экземпляры клипов фильмов являются объектами, в сценариях часто используются вызовы методов:

```
ball.play()
intro.gotoAndStop("end");
```

Подробнее о вызовах функций мы узнаем в главе 9, а в главе 12 «Объекты и классы» мы увидим, как функции могут стать методами объекта.

## Предложение call()

В простейшем виде функция является последовательностью многократно используемых предложений, которые могут быть выполнены в любой момент выполнения программы. Хотя Flash 4 не поддерживал настоящих функций, в нем делалась попытка обеспечить некоторую возможность их использования путем активации удаленных сценариев. Во Flash 4 можно было создать нечто похожее на функцию путем прикрепления списка предложений к кадру с меткой. Создаваемую при этом псевдофункцию можно было затем выполнить с помощью предложения *call( )*:

```
call (frame);
```

Предложение *call( )* выполняет сценарий в кадре, задаваемом параметром `frame`, который может являться любой меткой или номером кадра. Если указанный кадр не загружен, предложение *call( )* просто не выполняется.

Очевидно, что псевдофункции Flash 4 уступают настоящим функциям, поэтому обычно нет причин использовать предложение *call( )* при создании документов Flash 5 или более поздних версий. Однако при создании фильмов для Flash 4 необходимо использовать подпрограммы в старом стиле и предложение *call( )* Flash 4.

## Предложение return

При вызове функции можно передавать ей одно или более значений (*параметров* или *аргументов*), участвующих в выполнении. Аналогично, функция может передавать обратно *возвращаемое значение* (значение, получаемое в результате выполнения функции и пересылаемое обратно вызвавшему). Для завершения выполнения функции и, если нужно, возврата значения, в ее теле используется предложение *return*. Предложение *return* имеет один из следующих форматов:

```
return;
return expression;
```

Необязательное `expression`, если оно есть, становится значением, возвращаемым функцией. Предложение *return* без возвращаемого значения завершает выполнение функции и возвращает значение `undefined`. Все предложения *return* осуществляют выход из текущей функции. Заметьте, что предложение *return* не обязательно должно присутствовать в функции: функция без *return* просто заканчивает работу после выполнения последнего предложения в теле функции и возвращает значение `undefined`. Подробнее о создании, вызове и завершении работы функций см. в главе 9.

## Предложение with

Предложение *with* предоставляет сокращенный способ обращения к свойствам объекта без необходимости каждый раз указывать имя объекта. Общий вид предложения *with*:

```
with (object) {
  statements
}
```

Если обращение к свойству происходит в блоке предложения *with*, проверяется наличие этого свойства у объекта *object*. Если у *object* существует указанное свойство, то оно и используется для разрешения ссылки на свойство. Если у *object* нет такого свойства, оно ищется в текущей временной диаграмме или функции.

Следующий пример демонстрирует разницу между выполнением предложения внутри *with* и вне *with*:

```
PI = 10;               // Задать 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)
```

Помимо предоставления удобного доступа к свойствам объекта, *with* может использоваться 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 предложения *with*. Обратите внимание в предыдущем примере на то, что переменная `larger` *не* определена в объекте *Math*, поэтому ссылка на свойство воздействует на временную диаграмму или функцию, содержащую предложение *with*. Следующий код показывает неверную попытку присвоить переменной значение `myClip`:

```
with (myClip) {
  var x = 10;  // значение x задается в текущей временной диаграмме,
               // а не в myClip
}
```

Можно, однако, с полным правом использовать *with* для воздействия на экземпляры клипов другими способами. Это предложение предоставляет удобный способ работы с глубоко вложенными структурами экземпляров. Например, следующий код:

```
  _root.form.userProfile.userID = "U346BX";
```

```
_root.form.userProfile.gotoAndPlay("questionnaire");
```

можно заменить таким:

```
with (_root.form.userProfile) {
  userID = "U346BX";              // Выполняет присваивание существующей
                                  // переменной экземпляра userProfile
  gotoAndPlay("questionnaire");   // Функция применяется
                                  // к экземпляру userProfile
}
```

Но *with* не является единственным способом обеспечения такого удобства. Можно просто присвоить наш экземпляр переменной и использовать эту переменную в ссылках:

```
var userForm = _root.form.userProfile;
userForm.useriD = "U346BX";
userForm.gotoAndPlay("questionnaire");
```

Многим разработчикам использование переменных кажется проще для чтения и работы, но допустимыми являются оба подхода. Подробнее о работе с клипами как с объектами мы узнаем в главе 13 «Клипы».

## Предложение ifFrameLoaded

Предложение Flash 5 *ifFrameLoaded* заменяет действие *If Frame Is Loaded*, использовавшееся в предыдущих версиях. Подобно предложению *if*, предложение *ifFrameLoaded* имеет подпредложение, которое выполняется только при определенных обстоятельствах. Вот его синтаксис:

```
ifFrameLoaded (expression) {
  statements
}
```

где *expression* должно быть номером кадра или строкой, указывающей метку кадра. Если кадр, заданный *expression*, загружен в Player, выполняются предложения *statements*. Если нет, блок предложений пропускается.

Предложение *ifFrameLoaded* неудобно использовать для предварительной загрузки сценариев: в нем нет предложения *else*. Поэтому оно считается устаревшим и должно использоваться только при создании фильмов для Flash 3 и более старых версий. Начиная с Flash 4 следует использовать свойства `_totalframes` и `_framesloaded` с предложениями *if-else* для создания более гибкого предварительного загрузчика. Например:

```
if (_totalframes > 0 && _framesloaded == _totalframes) {
  gotoAndPlay("beginMovie");
} else {
  gotoAndPlay(_currentframe - 1);
}
```

## Пустое предложение

Для полноты изложения материала следует отметить, что допускается использование предложения, не имеющего содержания и состоящего только из точки с запятой:

```
;
```

*Пустое предложение* почти не имеет практической ценности и может использоваться как заместитель там, где обычно ожидается предложение. Оно не обязательно, если вам просто нужно добавить в код пустые строки. ActionScript игнорирует пустые строки, которые можно использовать для удобства чтения кода.

При создании документа в режиме для новичков одиночная точка с запятой появляется, когда в блок кода добавляется действие *evaluate*. При этом можно добавить произвольное предложение путем ввода его в поле Expression панели Parameters.

# Сравнение предложений с действиями

В среде редактирования Flash ActionScript вы не найдете слова «statement» (предложение). Даже в руководстве Macromedia «Flash 5 ActionScript Reference Guide» термины «Action» и «statement» взаимно заменяются.

Использование слова «Action» (действие, процедура) как синонима для слова «statement» скрадывает различие между несколькими разными инструментами ActionScript. Чтобы увидеть это, откройте среду разработки ActionScript и посмотрите папку Actions, как показано на рис. 1.2. В списке действий этой папки вы найдете предложения, которые мы видели ранее в табл. 6.1. Вперемежку с предложениями там находится много функций – *gotoAndPlay( )*, *getURL( )*, *startDrag( )* и т. д. Хотя перечисленные в Actions функции можно использовать в предложениях, формально они не являются уникальными типами предложений – они просто встроенные функции. Предложения, некоторые встроенные функции и обработчики событий названы Macromedia действиями (Actions). В данной книге мы не используем этот общий термин. Каждое действие (Action) мы описываем термином, соответствующим его формальной роли в языке: *предложение*, *функция* или *обработчик событий*.

# Вперед!

Хотя мы много узнали об предложениях, два очень важных типа предложений – условные предложения и циклы – еще ждут своего описания. Мы разберемся с этими двумя типами в двух последующих главах.

# 7

# Условные предложения

Большинство примеров кода, которые мы до сих пор видели, были *линейными*: все предложения выполнялись поочередно, начиная с первого и заканчивая последним. Линейный код всегда делает одно и то же. Напротив, *условное предложение* выполняет действие только в том случае, когда удовлетворено указанное условие. В линейном коде интерпретатор может выполнить предложение A, затем предложение B и, наконец, предложение C. Пользуясь условными предложениями, можно потребовать от интерпретатора выполнить предложение A, а затем выполнить *либо* предложение B, *либо* предложение C, в зависимости от условия X.

Условные предложения можно использовать для создания и контроля ситуаций, имеющих 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 во время регистрации пользователя должен быть выполнен только один из этих блоков. Условное предложение позволяет выполнить нужный блок и пропустить неподходящий.

Как интерпретатор узнает, какой блок кода должен быть выполнен? Когда мы определяем условное предложение, то указываем условие, которое должно быть удовлетворено, чтобы выполнялся *первый* блок кода. Если условие не удовлетворяется, может быть выполнен альтернативный блок (который, в свою очередь, может тоже содержать усло-

вие). В сущности, мы устанавливаем в программе логику блок-схемы, которую можно записать на псевдокоде следующим образом:

```
if (первое условие выполнено) {
  // Выполнить этот код
} else if (второе условие выполнено) {
  // Выполнить этот код
} ...otherwise {
  // Выполнить этот код
}
```

Конечно, каждое условие нужно описать на языке, понятном интерпретатору. Это не проблема – все дело в синтаксисе, который мы сейчас изучим. Смысл в том, что все условные предложения разрешают или запрещают выполнение блока кода, исходя из заданного условия. Теперь посмотрим на практике, как работают условные предложения.

# Предложение if

Предложение *if* постоянно используется в качестве условного предложения общего назначения. С помощью *if* в коде создается двухлучевое разветвление, как развилка на дороге. Предложение *if* содержит одно или более подпредложений, которые выполняются, только если удовлетворено заданное условие. Синтаксис предложения *if* следующий:

```
if (condition) {
  substatements
}
```

Предложение *if* начинается, как и можно ожидать, с ключевого слова `if`. Условие *condition*, которое должно быть удовлетворено, чтобы выполнились предложения *substatements*, заключается в круглые скобки. *substatements* – одно или более предложений ActionScript. Каждое подпредложение должно занимать отдельную строку и оканчиваться точкой с запятой. Предложение *if* в целом завершается закрывающей фигурной скобкой (}) без замыкающей точки с запятой.

Условие *condition* предложения *if* может быть любым допустимым выражением. При выполнении предложения *if*, интерпретатор проверяет значение этого выражение (называемого *проверочным выражением*). Если оно равно `true`, то выполняются *substatements*. В противном случае *substatements* не выполняются. Ниже в качестве проверочных выражений используются простые булевы выражения:

```
if (true) {
  trace("The condition was met!");  // Это предложение будет выполнено
}
if (false) {
  trace("The condition was met!");  // Это предложение не выполняется
}
```

Конечно, нет практического смысла использовать булевы литералы в качестве проверочных выражений, поскольку их значения никогда не меняются. Вместо них мы будем пользоваться сложными выражениями, *возвращающими* булевы значения. Например, выражения, в которые входит операция сравнения, возвращают булево значение и отлично подходят в качестве проверочных выражений:

```
var pointerX = _xmouse;  // Положение мыши по горизонтали

// Если pointerX > 300 дает true...
if (pointerX > 300) {
  // ...выполняется это предложение
  trace("Мышь находится за отметкой 300 пикселов ");
}
```

Теперь о самом главном: проверочное выражение условного предложения не обязательно должно давать в результате булево значение – годится любое выражение. В качестве проверочного выражения условного предложения можно использовать число или строку:

```
if ("hi") {
  trace("Условие выполнено!");
}
if (4) {
  trace("Условие выполнено!");
}
```

Почему это действует, когда выражения «hi» и 4 не являются булевыми? Ответ заключается в чудесных преобразованиях типов, показанных в табл. 3.3. Когда проверочное выражение условного предложения не является булевым, интерпретатор преобразует его в булево. Например, интерпретатор преобразует «hi» в `false`, потому что все нечисловые строки при использовании в булевом контексте принимают значение `false`. Поэтому условие не выполнено, и первое предложение *trace ( )* тоже не выполняется. Аналогично, интерпретатор преобразует число 4 в `true` (любое ненулевое число преобразуется в `true`), поэтому второе предложение *trace ( )* выполняется.

Все наши прежние усилия по изучению преобразования типов данных оправдались! Вот несколько основных практических примеров. Попытайтесь определить для каждого предложения, будет ли оно выполнено:

```
x = 3;
if (x) {
  trace("x не равен нулю");
}
```

В следующем примере используется оператор ИЛИ, описанный в главе 5 «Операторы»:

```
lastName = "";
firstName = "";
if (firstName != "" || lastName != "") {
  trace("Welcome " + firstName + " " + lastName);
}
```

Наконец, проверим, существует ли объект клипа:

```
if (myClip) {
  myClip._x = 0;  // Если myClip существует, поместить его
                  // на левый край рабочего стола
}
```

# Предложение else

С помощью одиночного предложения *if* можно спровоцировать выполнение одиночного блока кода. Добавив предложение *else*, можно выбрать, который из двух блоков кода будет исполнен. Синтаксически предложение *else* является расширением предложения *if*:

```
if (condition) {
  substatements1
} else {
  substatements2
}
```

где *condition* может быть любым допустимым выражением. *substatements1* будут выполнены, если *condition* имеет значение `true`; *substatements2* будут выполнены, если *condition* имеет значение `false`. Иными словами, предложение *else* полностью подходит для отражения взаимоисключающего решения: один блок кода будет выполнен, а другой – нет.

Вот некоторый код для иллюстрации:

```
var lastName = "Grossman";
var gender = "male";
if (gender == "male") {
  trace("Good morning, Mr. " + lastName + ".");
} else {
  trace("Good morning, Ms. " + lastName + ".");
}
```

Предложение *else* часто действует как резервный план для предложения *if*. Вспомните наш пример сайта, защищенного паролем. Если пароль верный, мы позволяем пользователю зайти на сайт, в противном случае выводим сообщение об ошибке. Вот код, который можно использовать для выполнения проверки пароля (пусть переменные `userName` и `password` содержат введенные пользователем данные, а `validUser`

и correctPassword являются правильными значениями этих переменных для регистрации):

```
if (userName == validUser && password == correctPassword) {
  gotoAndPlay("intro");
} else {
  gotoAndStop("loginError");
}
```

## Условный оператор

Простые условные предложения из двух частей удобно выражать с помощью условного оператора (?:). Условный оператор состоит из трех частей, в которых можно узнать аналоги компонентов предложений *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 просто более короткий способ записать следующее условное предложение:

```
if (condition) {
  expression1
} else {
  expression2
}
```

Как и условное предложение, условный оператор можно использовать для управления логикой выполнения программы:

```
// Если команда "go", воспроизвести фильм, иначе остановиться
command == "go" ? play() : stop();
```

Условный оператор предоставляет также краткий способ присвоения переменным значений:

```
var guess = "c";
var answer = "d";
var response = (guess == answer) ? "Right" : "Wrong";
```

Это эквивалентно следующему:

```
var guess = "c";
var answer = "d";
if (guess == answer) {
  response = "Right";
} else {
  response = "Wrong";
}
```

# Предложение 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 ни одного) из *неограниченного* числа блоков кода. Как и *else*, *else if* является синтаксическим расширением предложения *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. *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 так далее для всех имеющихся предложений *else if*. Если ни одно из проверочных предложений не имеет значения `true`, будут выполнены предложения завершающего предложения *else*. Например, можно написать программу проверки регистрации с умными сообщениями об ошибках:

```
if (userName != validUser) {
  message = "Пользователь не найден. Попробуйте снова.";
  gotoAndStop("loginError");
} else if (password != correctPassword) {
  message = "Пароль неверен. Попробуйте снова.";
  gotoAndStop("loginError");
} else {
  gotoAndPlay("intro");
}
```

Обратите внимание, что предложение *else if* является просто объединением *else* с вложенным предложением *if*. Хотя следующие два участка кода эквивалентны, первый из них значительно легче читать:

```
// Обычный синтаксис "else if"
if (x > y) {
  trace("x is larger than y");
} else if (x < y) {
  trace("x is smaller than y");
} else {
  trace("x and y are equal");
}
```

```
// Расширенная цепочка if/else
if (x > y) {
  trace("x is larger than y");
} else {
  if (x < y) {
    trace("x is smaller than y");
  } else {
    trace("x and y are equal");
  }
}
```

# Эмуляция предложения switch

Хотя предложения переключателя *switch* (иногда называемые предложениями *case*) не поддерживаются в ActionScript, этот распространенный вид сложного условного предложения можно эмулировать. Предложение *switch* позволяет выполнить только один из ряда возможных блоков кода, исходя из значения одного проверочного выражения. Например, в следующем предложении *switch* JavaScript мы приветствуем пользователя сообщением, зависящим от значения проверочного выражения `gender`:

```
var surname = "Porter";
var gender = "male";

switch (gender) {
  case "femaleMarried" :
    alert("Hello Mrs. " + surname);
    break;
  case "femaleGeneric" :
    alert("Hello Ms. " + surname);
    break;
  case "male" :
    alert("Hello Mr. " + surname);
    break;
  default :
    alert("Hello " + surname);
}
```

В примере JavaScript *switch* пытается найти соответствие значения `gender` и одного из выражений *case*: «femaleMarried», «femaleGeneric» или «male». Поскольку `gender` соответствует выражению «male», выполняется подпредложение *alert("Hello Mr. " + surname);*. Если проверочное выражение не соответствует ни одному случаю, то выполняется предложение по умолчанию *alert("Hello " + surname);*.

В ActionScript можно эмулировать предложение *switch* с помощью последовательности предложений *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);
}
```

При более развитом подходе можно моделировать *switch* в виде ряда функций, хранимых в свойствах общего объекта. Такой подход продемонстрирован в примере 7.1. Чтобы понять, как он работает, следует внимательно изучить комментарии. Обратите также внимание на условный оператор, с которым мы встречались выше.

*Пример 7.1. Эмуляция предложения switch*

```
var surname = "Porter"; // Имя нашего пользователя
var gender = "male";    // Пол нашего пользователя (проверочное выражение)

// Создать объект, который будет эмулировать предложение switch
var mySwitch = new Object();

// Присвоить объекту mySwitch свойства "выражения case".
// Каждое свойство "выражения case" содержит функцию.
mySwitch.femaleMarried = function() {
  trace("Hello Mrs. " + surname);
};
mySwitch.femaleGeneric = function() {
  trace("Hello Ms. " + surname);
};
mySwitch.male = function() {
  trace("Hello Mr. " + surname);
};
mySwitch.default = function() {
  trace("Hello " + surname);
};

// Теперь выполним надлежащую функцию в зависимости от значения
// gender (в нашем случае, "male"). Если заданное свойство
// не существует, выполнить функцию по умолчанию.
mySwitch[gender] ? mySwitch[gender]() : mySwitch["default"]();
```

# Компактный синтаксис условных предложений

На протяжении этой книги мы используем во всех условных предложениях блоки предложений (предложения, заключенные в фигурные скобки), даже если в этих блоках всего одно предложение:

```
if (x == y) {
  trace("x and y are equal");
}
```

Для краткости ActionScript разрешает не использовать фигурные скобки, когда в условном предложении лишь одно подпредложение. Единственное подпредложение можно вполне законно поместить прямо после предложения *if* или *else if* без фигурных скобок, например:

```
if (x == y) trace ("x and y are equal");
```

Или так:

```
if (x == y)
  trace ("x and y are equal");
```

Некоторые программисты считают такой стиль неудобным для чтения и более подверженным ошибкам, но, несомненно, при этом экономится место в исходном коде.

# Вперед!

Условные выражения составляют крайне важную часть ActionScript. Они дают точный контроль над выполнением кода. В следующей главе мы рассмотрим другой важный тип предложений, управляющих выполнением – *циклы*.

# 8

# Предложения цикла

В предыдущей главе мы узнали, что *условное предложение* вызывает однократное выполнение блока предложений, если его проверочное выражение имеет значение `true`. *Цикл* же производит многократное выполнение блока предложений, пока его проверочное выражение сохраняет значение `true`.

Циклы бывают различных типов: *while*, *do-while*, *for* и *for-in*. Первые три типа обладают сходным действием, различаясь синтаксисом. Последний тип цикла, *for-in*, является особым видом цикла, используемым с объектами. Мы начнем исследование циклов с предложения *while*, являющегося простейшим для понимания видом цикла.

## Цикл while

По структуре предложение *while* во многом похоже на предложение *if*: главное предложение содержит блок подпредложений, выполняемых только тогда, когда заданное условие имеет значение `true`:

```
while (condition) {
  substatements
}
```

Если условие имеет значение `true`, выполняются предложения *substatements*. Но, в отличие от *if*, по завершении последнего из подпредложений выполнение повторяется снова с начала предложения *while* (т. е. интерпретатор «зацикливается» на начале предложения *while*). Второй проход через предложение *while* действует точно так же, как первый: вычисляется условие, и если оно по-прежнему оказывается

true, снова выполняются *substatements*. Этот процесс продолжается, пока *condition* не станет false, и тогда выполнение продолжится с тех предложений, которые следуют 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, скорее всего, вызовет ошибку. Чтобы узнать причину, проследим за тем, как интерпретатор выполняет этот пример.

Мы начнем с предложения, предшествующего *while*, а именно, *var i = 3*, которое присваивает переменной i значение 3. Поскольку переменная i используется в проверочном выражении цикла, этот шаг часто называется *инициализацией* цикла. Затем мы начинаем выполнять предложение *while*, вычисляя проверочное выражение: *i < 5*. Поскольку i равно 3, а 3 меньше 5, значением проверочного выражения является true, и мы выполняем предложение *trace( )* в цикле.

Когда это сделано, наступает момент перезапуска цикла. Мы снова проверяем значение проверочного выражения. Значение переменной i не изменилось, поэтому проверочное выражение по-прежнему true, и мы снова выполняем предложение *trace( )*. Закончив выполнение тела цикла, нужно перезапустить цикл. Что дальше? Переменная i все еще не изменилась, поэтому проверочное выражение по-прежнему true, и мы должны выполнять предложение *trace( )* снова, и снова, и снова – до бесконечности. Поскольку проверочное выражение всегда возвращает true, невозможно выйти из цикла – мы навсегда попали в *бесконечный цикл* и оказались не в состоянии выполнить предложения, которые могут находиться за циклом *while*. Как мы увидим позже, в ActionScript бесконечный цикл вызывает ошибку.

Наш цикл бесконечен, поскольку в нем отсутствует *обновляющее предложение*, которое изменяет значение переменной, используемой в проверочном выражении. Обновляющее предложение обычно приводит к тому, что проверочное выражение в конечном итоге становится равным false, благодаря чему цикл завершается. Подправим наш бесконечный цикл, добавив к нему обновляющее предложение:

```
var i = 3;
while (i < 5) {
  trace("x is less than 5");
  i++;
}
```

Предложение *i++* находится в конце тела цикла. Когда интерпретатор проходит через цикл, он, как и прежде, выполняет предложение *trace( )*,

но выполняет также предложение *i*++, которое прибавляет единицу к переменной i. При каждом проходе цикла значение i возрастает. После второго прохода значение i равно 5, поэтому проверочное предложение *i* < *5* получает значение false. Благодаря этому цикл благополучно завершается.

Наше обновляющее предложение выполняет в цикле базовую функцию – ведет счетчик. Переменная i (называемая *счетчиком*) пробегает предсказуемую последовательность чисел, что в высшей степени 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 это делается в цикле:

```
var i = 1;
while (i <= 5) {
  duplicateMovieClip("square", "square" + i, i);
  i++;
}
```

Представьте себе разницу, если мы делаем 100 копий square!

Циклы чрезвычайно полезны для работы с данными, особенно теми, что 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"];  // Create an array
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 на использование переменной i в проверочном выражении и одновременно в качестве индекса массива, что является типичным. Ниже мы снова используем i как аргумент функции *charAt( )*:

```
var city = "Toronto";
trace("Буквами переменной 'city' являются:");
var i = 0;
while (i < city.length) {
  trace(city.charAt(i));
  i++;
}
```

Окно Output показывает следующее:

```
Буквами переменной 'city' являются:
T
o
r
o
n
t
o
```

Наконец, используем 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 проходить через данные. Это особенно удобно, когда максимальное значение счетчика определено 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 false, цикл будет за-

вершен. Вот, например, как просмотреть стек уровней 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
loadMovie("myMovie.swf", i);
```

# Терминология циклов

В предыдущем разделе мы столкнулись с несколькими новыми терминами. Рассмотрим их более формально, чтобы лучше понимать при работе с циклами:

*Инициализация*

Предложение или выражение, определяющие одну или более переменных, используемых в проверочном выражении цикла.

*Проверочное выражение*

Условие, при удовлетворении которого выполняются подпредложения в теле цикла. Часто называется *условием* (*condition*), или *проверкой* (*test*), или, иногда, *управлением* (*control*).

*Модификация (update)*

Предложения, модифицирующие переменные, используемые в проверочном выражении перед следующей проверкой. Типичное модифицирующее предложение инкрементирует или декрементирует счетчик цикла.

*Итерация*

Одно полное выполнение проверочного выражения и предложений в теле цикла. Иногда ее также называют одним *циклом* (*loop*) или одним *проходом* (*pass*).

*Вложенный цикл (nested loop)*

Цикл, содержаший в себе другой цикл, благодаря которому можно совершать обход неких двумерных данных. Например, можно циклически пройти по каждой строке для каждого столбца таблицы. Внешний цикл, или цикл верхнего уровня, будет перемещаться по столбцам, а внутренний цикл – по строкам каждого столбца.

*Итератор или индексная переменная*

Переменная, значение которой уменьшается или увеличивается при каждом проходе цикла, обычно используемая для подсчета или последовательного перемещения по данным. Итераторы циклов

часто называют *счетчиками*. Обычно итераторы именуют буквами `i`, `j` и `k` или `x`, `y` и `z`. В последовательности вложенных циклов `i` обычно является итератором верхнего уровня, `j` – первого вложенного цикла, `k` – второго вложенного цикла, и так далее. Для ясности можно использовать любое понравившееся имя переменной. Например, можно использовать в качестве имени переменной `charNum`, чтобы помнить, что это текущий символ строки.

*Тело цикла*

Блок предложений, выполняемых, когда удовлетворено условие цикла. Тело может не выполняться вообще или выполняться тысячи раз.

*Заголовок цикла (loop header) или управление цикла (loop control)*

Часть цикла, содержащая ключевое слово предложения цикла (*while*, *for*, *do-while* или *for-in*) и условие цикла. Заголовок цикла зависит от типа цикла. В цикле *for* он включает в себя инициализацию, проверку и модификацию; в цикле *while* заголовком является только проверочное выражение.

*Бесконечный цикл*

Цикл, который выполняется бесконечно, т. к. проверочное выражение никогда не дает значение `false`. Бесконечные циклы вызывают в ActionScript ошибку, о чем рассказывается ниже в разделе «Максимальное число итераций».

# Цикл do-while

Как мы уже видели, предложение *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 предложения *do-while* в некотором роде обратен синтаксису предложения *while*:

```
do {
  substatements
} while (condition);
```

Начинается цикл с ключевого слова *do*, затем следуют находящиеся в теле подпредложения *substatements*. При первом проходе интерпретатором предложения *do-while* эти *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 `false`, на чем выполнение предложения *do-while* за-

канчивается. Обратите внимание на необходимость точки с запятой после скобок, содержащих условие *condition*.

Очевидно, *do-while* удобно использовать, когда нужно 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. Использование цикла do-while*

```
var numStars = 5;
var i = 1;
do {
  // Duplicate the starParent clip
  duplicateMovieClip(starParent, "star" + i, i);

  // Place the duplicated clip randomly on Stage
  _root["star" + i]._x = Math.floor(Math.random() * 551);
  _root["star" + i]._y = Math.floor(Math.random() * 401);
} while (i++ < numStars);
```

Заметили ли вы, что мы тайком изменили переменную `i` в проверочном выражении? Вспомните из главы 5 «Операторы», что постфиксный оператор инкрементирования возвращает значение своего операнда, а также прибавляет к этому операнду единицу. Оператор инкрементирования очень удобен (и часто используется) при работе с циклами.

# Цикл for

Цикл *for* по сути синонимичен циклу *while*, но записывается в более компактном синтаксисе. Самое заметное то, что заголовок цикла может, помимо проверочного выражения, содержать предложения 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 предложение *initialization* (один и только один раз). Обычно оно используется для задания начального значения переменной итератора. Как и в других циклах, если *condition* имеет значение `true`, выполняются *substatements*. В противном случае, цикл завершается. В *конце* каждой итерации выполняется предложение *update*, перед очередной проверкой *condition*, позволяющей узнать, дол-

жен 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("Now serving number " + i);
  i++;
}
```

Привыкнув к синтаксису цикла *for*, вы обнаружите, что он экономит место и позволяет легко просматривать тело и заголовок цикла.

## Множественные итераторы в циклах for

Если нужно управлять в цикле более чем одним фактором, можно использовать 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 предложении *for* с помощью оператора запятая:

```
for (var i = 1, j = 10; i <= 10; i++, j--) {
  trace("Going up " + i);
  trace("Going down " + j);
}
```

# Цикл for–in

Предложение *for-in* является специальным циклом, используемым для перечисления свойств объекта. Новички в программировании могут пока пропустить этот раздел и вернуться к нему после чтения главы 12 «Объекты и классы».

Вместо повторения серии предложений, пока заданное проверочное выражение не даст значения `false`, цикл *for-in* осуществляет по одной

итерации для каждого свойства заданного объекта. Предложения *for-in* не требуют явного обновляющего предложения, поскольку число итераций цикла определяется числом свойств рассматриваемого объекта. Синтаксис цикла *for-in* выглядит так:

```
for (var thisProp in object) {
  substatements;  // Предложения обычно как-то используют thisProp
}
```

Предложения *substatements* выполняются по одному разу для каждого свойства *object*; *object* является именем произвольного действительного объекта; *thisProp* является произвольным именем переменной или идентификатора. На каждой итерации цикла переменная *thisProp* временно хранит 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. Простейшим примером цикла *for-in* является сценарий, который перечисляет свойства объекта. Ниже мы создаем объект, а затем перечисляем по пунктам 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` как строки, для извлечения этих свойств можно воспользоваться `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. Демонстрацию цикла *for-in*, используемого как детектор клипов, можно увидеть в примере 3.1.

Обратите внимание, что свойства объекта, просматриваемые в цикле *for-in*, нумеруются в непредсказуемом порядке. Кроме того, предложения *for-in* не всегда перечисляют все свойства объекта. Если объект определен пользователем, перечисляются все свойства, включая унаследованные. Но некоторые свойства встроенных объектов предложением *for-in* пропускаются. Например, этим циклом не перечисляются методы встроенных объектов. Если вы хотите использовать предложение *for-in* для работы со свойствами встроенного объекта, сначала создайте проверочный цикл для определения доступных свойств объекта.

Текстовые поля ввода, не имеющие значений по умолчанию, не перечисляются в цикле *for-in*. Поэтому код для проверки формы, определяющий пустые текстовые поля, будет работать неправильно, если их не объявить явно как обычные переменные на временной диаграмме, где они находятся. См. раздел «Пустые текстовые поля и цикл for-in» главы 18.

Предложение *for-in* можно также использовать для извлечения элементов массива, и тогда оно принимает вид:

```
for (var thisElem in array) {
  substatements; // Обычно предложения 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 предложение простого цикла дает значение `false`, цикл завершается. Однако если цикл становится более сложным, может потребоваться остановка его выполнения независимо от значения проверочного предложения. Для этого используются предложения *break* и *continue*.

## Предложение break

Предложение *break* прекращает выполнение текущего цикла. Его синтаксис скромен:

```
break
```

Единственным требованием является нахождение *break* в теле цикла.

Предложение *break* предоставляет способ остановить процесс, продолжать который больше нет необходимости. Например, можно использовать цикл *for-in* в программе проверки формы, которая обходит переменные текстовых полей ввода на временной диаграмме. При обнаружении пустого поля ввода мы предупреждаем пользователя о том, что он неправильно заполнил форму. Прекратить процесс можно, выполнив предложение *break*. Соответствующий код показан в примере 8.3.

Обратите внимание, что в примере предполагается существование клипа с именем `form`, который содержит ряд объявленных входных переменных с именами `input01`, `input02` и т. д.

*Пример 8.3. Простая программа проверки полей формы*

```
for (var prop in form) {
  // Если это свойство служит одним из наших "входных" текстовых полей
  if (prop.indexOf("input") != -1) {
    // Если в поле формы пусто, прервать операцию
    if (form[prop] == "") {
      displayMessage = "Пожалуйста, заполните форму полностью.";
      break;
    }
    // Все подпредложения после команды break пропускаются,
    // когда выполнена break
  }
}
// Отсюда возобновляется выполнение после окончания цикла, вызванного командой
break или присваиванием проверочному выражению значения false
```

Предложение *break* можно использовать для прерывания цикла, который иначе был бы бесконечным. Это позволяет, скажем, выполнить предложения первой части блока и не выполнять предложения, следующие за *if (условие) break;*. Общий подход показан в примере 8.4.

*Пример 8.4. Прерывание бесконечного цикла*

```
while (true) {
  // Здесь идут начальные предложения
  if (condition) break;
  // Здесь идут последующие предложения
}
```

## Предложение continue

Предложение *continue* аналогично предложению *break*, поскольку прерывает текущую итерацию цикла, но в отличие от *break* возобновляет выполнение цикла с очередной естественной итерации. Синтаксис предложения *continue* простой:

```
continue
```

Во всех типах циклов предложение *continue* прерывает текущую итерацию тела цикла, но при возобновлении цикла могут быть некоторые различия в зависимости от типа предложения цикла. В циклах *while* и *do-while* проверочное выражение проверяется перед возобновлением цикла. В цикле *for* перед проверкой условия выполняется модификация цикла. А в цикле *for-in* новая итерация начинается с очередного свойства рассматриваемого объекта (если таковое существует).

С помощью предложения *continue* можно сделать выполнение тела цикла необязательным в зависимости от обстоятельств. Например, в следующем коде мы перемещаем все экземпляры клипов, не являющиеся прозрачными, 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

Как отмечалось раньше, в ActionScript не разрешаются бесконечные циклы. В проигрывателе Flash 5 их выполнение ограничено 15 секундами. Количество итераций, которые могут быть выполнены за это время, зависит от того, что находится внутри цикла, и от скорости компьютера. В целях безопасности не следует создавать циклы, выполнение которых требует более нескольких секунд (для процессора это составляет целую эру!). 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), стоит переписать код с использованием цикла временной диаграммы. Циклы временной диаграммы позволяют изменять ход выполнения сценария на экране и избегать возможного вывода сообщения об ошибке, показанного на рис. 8.1.

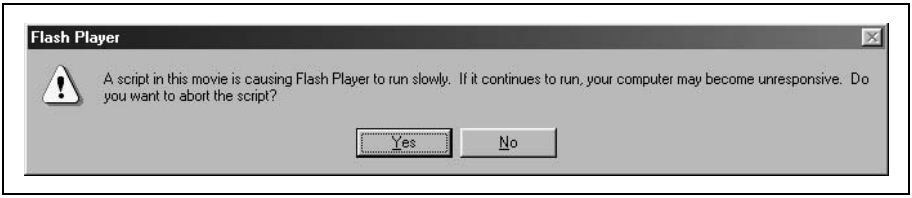


***Рис. 8.1.*** *Плохой цикл! Закрывай, парень!*

Если цикл выполнялся в проигрывателе Flash 5 больше 15 секунд, выводится окно, предупреждающее пользователя, что сценарий в фильме задерживает воспроизведение фильма. Пользователю предлагается выбор: ждать завершения сценария или прекратить его выполнение.

Проигрыватель Flash 4 еще более строг: он разрешает только 200000 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 выключаются.

# Циклы временной диаграммы и событий клипа

Все циклы, которые мы до сих пор видели, заставляют интерпретатор повторно выполнять блоки кода. Большинство ваших циклов будет иметь этот тип «предложения ActionScript». Но иногда желательно также создать *цикл временной диаграммы* (*timeline loop*), зациклив воспроизводящую головку Flash на временной диаграмме. Для этого прикрепите ряд предложений 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. Начните новый фильм Flash.
2. К кадру 1 прикрепите следующее предложение:

```
trace("Привет! Добро пожаловать в кадр 1");
```

3. К кадру 2 прикрепите следующие предложения:

```
trace("Это кадр 2");
gotoAndPlay(1);
```

4. Выберите меню 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 этим ошибки.
- Выполнять блок кода, требующий обновления рабочего стола между итерациями.

Эта вторая особенность циклов временной диаграммы требует некоторых разъяснений. При выполнении сценария любого кадра рабочий стол фильма зрительно не обновляется до окончания сценария. Это означает, что обычные предложения цикла не могут использоваться для выполнения повторяющихся зрительных и звуковых заданий, потому что результаты заданий между итерациями цикла не выводятся. Например, перемещение клипа требует обновления рабочего стола, поэтому с помощью обычного цикла нельзя осуществить программную анимацию клипа.

Вы можете предположить, что следующий код зрительно перемещает клип `ball` в горизонтальном направлении по рабочему столу:

```
for (var i = 0; i < 50; i++) {
  ball._x += 10;
}
```

Теоретически у этого цикла правильный подход: он многократно изменяет положение шарика `ball` малыми порциями, что должно создавать впечатление движения. Однако на практике шарик не сдвигается при каждом изменении позиции `_x`, потому что рабочий стол 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.

Чтобы после каждого выполнения предложения `ball._x += 10;` совершать обновление рабочего стола, можно использовать такой цикл временной диаграммы:

```
// КОД В КАДРЕ 1
ball._x += 10;

// КОД В КАДРЕ 2
gotoAndPlay(1);
```

Поскольку Flash обновляет рабочий стол между каждыми двумя кадрами, шарик будет двигаться. Но цикл временной диаграммы полностью монополизирует временную диаграмму, на которой находится. Во время его выполнения на этой временной диаграмме нельзя воспроизводить обычное содержимое. Более правильный подход состоит в помещении нашего цикла временной диаграммы в пустой клип из двух кадров. Мы сможем использовать обновление рабочего стола между итерациями цикла, не замораживая временной диаграммы, которая может потребоваться для другой анимации.

## Создание цикла временной диаграммы пустого клипа

Следующие шаги показывают, как создать цикл временной диаграммы пустого клипа:

1. Начать новый фильм Flash.
2. Создать символ клипа с именем *ball*, который содержит круг.
3. На главном рабочем столе переименовать слой *Layer 1* в *ball*.
4. В слое *ball* разместить экземпляр символа *ball*.
5. Дать экземпляру клипа *ball* имя `ball`.
6. Выбрать Insert → New Symbol для создания символа пустого клипа.
7. Дать символу клипа имя `process`.
8. К кадру 1 клипа `process` прикрепить следующий код:

```
_root.ball._x += 10;
```

9. К кадру 2 клипа `process` добавить код:

```
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`.
13. Выбрать Control → Test Movie.

Теперь экземпляр `processMoveBall` будет перемещать `ball`, не мешая воспроизведению главной временной диаграммы, на которой располагается `ball`.

Обратите внимание, что шаг 12 не является обязательным, но он дает нам больший контроль над циклом. Дав имя нашему экземпляру цикла временной диаграммы, мы можем останавливать и запускать наш цикл, запуская и останавливая воспроизведение экземпляра:

```
processMoveBall.play();
processMoveBall.stop();
```

Заметьте, что `processMoveBall` и `ball` должны существовать на главной временной диаграмме в течение предполагаемого времени выполнения цикла. Для большей переносимости можно было создать в `process` относительную ссылку на клип `ball`:

```
_parent.ball._x += 10;
```

А если требуется управлять шариком из любой временной диаграммы, можно использовать абсолютную ссылку на `ball`:

```
_root.ball._x += 10;
```

Циклы временной диаграммы нельзя зацикливать на одном кадре. Это значит, что если поместить функцию `gotoAndPlay(5)` в кадре 5 фильма, она будет проигнорирована. Проигрыватель понимает, что воспроизводящая головка уже находится в кадре 5, и просто ничего не совершает.

Вы можете найти файлы *.fla* временной диаграммы и цикла пустого клипа в сетевом Хранилище кода.

## Циклы событий клипа во Flash 5

Циклы временной диаграммы эффективны, но не всегда элегантны. Во Flash 5 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 посмотрите подробноее об обработчиках событий клипов в главе 10 «События и их обработчики»).

Будучи помещенным в клип, обработчик события *enterFrame* влечет выполнение блока кода при каждом прохождении кадра в фильме. Можно использовать обработчик события *enterFrame* в пустом клипе с одним кадром для многократного выполнения блока кода, позволяя рабочему столу обновляться между каждыми двумя повторениями (так, как это делает цикл временной диаграммы). Чтобы испытать это, выполните следующие шаги:

1. Выполните шаги с 1 по 7 из предыдущего раздела.
2. На главном рабочем столе создайте слой с именем *scripts*.
3. Поместите экземпляр клипа `process` в слой *scripts*.
4. Выделите экземпляр `process` и прикрепите к нему код:

```
onClipEvent(enterFrame) {
_root.ball._x += 10;
}
```

5. Выберите Control → Test Movie.

Экземпляр `ball` должен двигаться по рабочему столу.

Циклы событий клипа освобождают от необходимости помещать код внутри клипа и не требуют, как циклы временных диаграмм, цикла из двух кадров. Все действие цикла события клипа происходит в одном обработчике события. Однако только что приведенный пример события клипа обладает потенциальным недостатком: нет способа программным путем запустить или остановить цикл после его начала. Единственным способом прекращения цикла является физическое удаление экземпляра `process` из временной диаграммы с помощью пустого ключевого кадра.

Чтобы создать цикл события, который можно произвольно запускать и останавливать, нужно создать пустой клип, содержащий *другой* пустой клип, в котором находится цикл события. После этого, когда нам нужно запустить или остановить цикл, можно динамически прикреплять и удалять весь пакет. Несколько запутанно, но в результате достигается значительная гибкость. Чтобы испытать это, выполните следующие шаги:

1. Выполните шаги с 1 по 5 из раздела «Создание цикла временной диаграммы пустого клипа».
2. Дважды выберите пункт меню Insert → New Symbol, чтобы создать два символа пустых клипов.
3. Одному символу дайте имя `process`, а другому – `eventLoop`.
4. Выберите в библиотеке клип `process`, затем выберите Options → Linkage. Появится окно диалога Symbol Linkage Properties.
5. Выполните Export This Symbol.
6. В поле Identifier введите `processMoveBall` и щелкните OK.
7. Из кадра 1 клипа `process` перетащите экземпляр `eventLoop` на рабочий стол.
8. Выделите экземпляр `eventLoop` и прикрепите к нему следующий код:

```
onClipEvent(enterFrame) {
_parent._parent.ball._x += 10;
}
```

9. Вернитесь в главную временную диаграмму фильма и прикрепите к кадру 1 следующий код:

```
attachMovie("processMoveBall", "processMoveBall", 5000);
```

10. Если нужно остановить цикл события, выполните предложение:

```
_root.processMoveBall.removeMovieClip();
```

11. Выполните Control → Test Movie.

Экземпляр `ball` снова должен перемещаться по рабочему столу, но на этот раз можно в любой момент запускать и останавливать его с помощью функций *attachMovie( )* и *removeMovieClip( )*, показанных в шагах 9 и 10.

Примеры обычных и управляемых циклов событий клипа есть в электронном Хранилище кода.

## Обеспечение переносимости циклов событий

Оба цикла событий клипа, которые мы только что видели, содержали строку кода, изменяющую положение экземпляра `ball` на рабочем столе. Например:

```
onClipEvent(enterFrame) {
  _parent._parent.ball._x += 10;  // Изменить положение ball
}
```

Такой способ действует, но он неряшлив. Прикрепив содержательный код к событию клипа, мы децентрализовали наш код, разбросав логику и поведение по фильму. Чтобы сохранять доступность кода во время создания документа и лучше структурировать его для повторного использования, из циклов событий следует только вызывать функции. Поэтому вместо фактического перемещения клипа `ball` следует вызывать функцию, которая перемещает клип `ball`, например:

```
onClipEvent(enterFrame) {
  _parent._parent.moveBall();
}
```

Определенная пользователем функция *moveBall( )* должна быть определена в той же временной диаграмме, к которой прикрепляется клип `processMoveBall`, например:

```
function moveBall() {
  ball._x += 10;
}
```

Подробнее о функциях и переносимости кода мы поговорим в главе 9 «Функции».

Если приложение простое, можно вообще обойтись без пустого клипа с обработчиком события. В некоторых случаях вполне допустимо прикрепить цикл события непосредственно к управляемому в настоящий момент клипу. В нашем примере с `ball` можно было избежать необходимости в пустых клипах, прикрепив непосредственно к экземпляру `ball` такой код:

```
onClipEvent(enterFrame) {
  _x += 10;
}
```

Такой подход крайне удобен, но его нелегко масштабировать и, так же как первый пример, он страдает невозможностью запускать и останавливать цикл.

## Воздействие частоты кадров на циклы временной диаграммы и обработчики событий клипов

Поскольку циклы временной диаграммы и событий клипов каждый кадр проходят ровно один раз, частота их выполнения связана со скоростью воспроизведения 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` в наших первых примерах, то неявно задавали скорость шарика относительно частоты кадров. Наш код говорит, что в каждом новом кадре нужно переместить шарик на десять пикселов вправо:

```
_ball += 10;
```

Поэтому скорость шарика зависит от частоты кадров. Если фильм воспроизводится со скоростью 12 кадров в секунду, то клип `ball` перемещается со скоростью 120 пикселов в секунду. Если фильм воспроизводится со скоростью 30 кадров в секунду, наш клип `ball` перемещается со скоростью *300* пикселов в секунду!

При определении временных параметров анимаций, управляемых сценарием, возникает мысль рассчитывать расстояние, на которое перемещается объект, в зависимости от частоты 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 часто отображает их медленнее. Замедление может быть различным в зависимости от загрузки системы: если выполняются другие программы или Flash выполняет задачу, требующую интенсивного использования процессора, частота кадров может на некоторое время упасть, а затем восстановиться до нормальной величины.

Вы можете сами убедиться в этом с помощью инструмента временного слежения, имеющегося по адресу:

*http://www.moock.org/webdesign/flash/actionscript/fps-speedometer*

В некоторых случаях воспроизведение анимации на слегка отличающихся скоростях может допускаться. 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 8.5 демонстрирует грубый образец анимации, основанной на времени (т. е. скорость `ball` не зависит от частоты кадров). Новый фильм должен иметь три кадра и два слоя: один слой с экземпляром `ball`, а другой с нашими сценариями.

*Пример 8.5. Расчет величины перемещения в зависимости от времени, а не 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 (пройденное) в секундах
var moveAmount;              // Величина перемещения

// КОД В КАДРЕ 2
then = now;
now = getTimer();
elapsed = now - then;
numSeconds = elapsed / 1000;
moveAmount = distancePerSecond * numSeconds;
ball._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.

# Вперед!

Мы забрались довольно далеко. Конец этой главы знаменует определенную веху: изучение предложений ActionScript закончено. Это означает, что в нашем распоряжении есть переменные, данные, типы данных, выражения, операторы и предложения. Эти составляющие 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 «Основы ActionScript» 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, ведь я собираюсь рассказать вам о функциях, а они являются очень мощной частью ActionScript. Функция – это просто фрагмент кода, который можно многократно использовать в программе. Функции не только придают сценариям чрезвычайную гибкость и удобство использования, но также предоставляют возможность управлять элементами фильма Flash. Трудно представить себе программирование без функций: они облегчают все – от сортировки слов до вычисления расстояния между двумя клипами. В этой главе мы познакомимся с функциями, а в главе 12 «Объекты и классы» узнаем, как создавать сложные мощные программы с помощью функций.

Сначала сосредоточимся на *программных функциях* – функциях, которые мы сами создаем в своих сценариях. Узнав, как создавать собственные функции, мы познакомимся с основами:

*Объявление функции*

Создание функций, используемых в сценариях.

*Вызов функции*

Действие, запускающее выполнение функции. Иными словами, выполнение кода функции.

*Аргументы и параметры функций*

Предоставление вызываемым функциям данных для обработки.

*Завершение функций*

Окончание выполнения функции с возможным возвратом результата.

*Область видимости функции*

Доступность и срок жизни функции, а также достижимость переменных, содержащихся в теле функции.

Изучив эти аспекты, мы рассмотрим, как они применяются к *внутренним функциям*, т. е. функциям, встроенным в ActionScript. Вперед!

# Создание функций

Чтобы создать простую *функцию*, нужны только имя функции и блок предложений, подлежащих выполнению, например:

```
function funcName () {
  statements
}
```

Ключевое слово `function` начинает *объявление* новой функции. Затем следует имя функции, *funcName*, которое позднее будет использоваться для *вызова* нашей функции. *funcName* должно быть допустимым идентификатором.[1] Затем ставится пара круглых скобок `()`, в которые заключаются возможные параметры (их мы обсудим позже). Если у функции нет параметров, оставим скобки пустыми. Наконец, далее следует *тело функции* (т. е. блок предложений), содержащее код, выполняемый при вызове функции.

Напишем простую (очень простую) функцию:

1. Начнем новый фильм Flash.
2. К кадру 1 главной временной диаграммы прикрепим следующий код:

```
function sayHi () {
  trace("Hi there!");
}
```

Это просто: мы всего лишь создали функцию с именем *sayHi()*. При ее запуске будет выполнено предложение *trace()*, находящееся в теле функции. Не закрывайте созданный вами фильм: далее 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 «Операторы»:

```
funcName()
```

[1] См. в главе 14 «Лексическая структура» правила составления допустимых идентификаторов.

(Можно почти различить слабый запах тлеющего в воздухе корня мандрагоры.)

Скобки могут содержать любые параметры, определенные функцией. Если функция не определяет параметры, скобки остаются пустыми. Попробуем вызвать нашу функцию *sayHi( )*, не имеющую параметров, чтобы освоиться с основами вызова функций.

Еще раз напомним объявление нашей функции *sayHi( )*:

```
function sayHi () {
  trace("Hi there!");
}
```

А вот вызов нашей функции *sayHi( )*:

```
sayHi();
```

При выполнении этой строчки вызывается *sayHi( )*, поэтому выполняется тело этой функции, что влечет появление «Hi there!» в окне Output.

Как видите, набрать на клавиатуре *sayHi( );* более удобно, чем полностью предложение *trace( )*, а имя функции *sayHi* представляет собой более содержательное описание того, что делает наш код. Использование продуманных имен функций облегчает чтение кода, делая его похожим на естественный язык.

Прежде чем продолжить, обратим внимание на точку с запятой в конце вызова функции:

```
sayHi();
```

Точка с запятой добавляется, поскольку вызов функции составляет законченное предложение, а хороший формат требует, чтобы все предложения заканчивались точкой с запятой.

Можете не верить, но мы только что изучили основы создания и вызова функций. Не так плохо. Функции позволяют централизовать код и облегчить его сопровождение, особенно если одно и то же действие приходится многократно выполнять на протяжении программы. Функции становятся еще более мощными, если использовать их с параметрами, что мы и обсудим в следующем разделе.

# Передача функции информации

В предыдущем разделе мы создали функцию, выполнявшую простое предложение *trace( )* – это был не самый убедительный представитель рода функций. Вот более интересная функция, которая перемещает экземпляр клипа с именем `ball` на небольшое расстояние:

```
function moveBall () {
  ball._x += 10;
```

```
    ball._y += 10;
}
```

С помощью определенной нами функции *moveBall( )* можно в любой момент переместить `ball` по диагонали, вызвав функцию *moveBall( )*:

```
moveBall();
```

Шарик перемещается по диагонали вправо вниз. (Заметьте, что исходное положение (0, 0) находится в левом верхнем углу рабочего стола. Увеличение значения `_x` перемещает шарик вправо, но, в отличие от декартовых координат, увеличение значения `_y` перемещает шарик *вниз*, а не вверх.)

Наша функция *moveBall( )* удобна, но ей недостает гибкости. Она действует только с одним клипом шарика (`ball`), перемещает `ball` только в одном направлении и всегда на одно и то же расстояние.

Правильно спроектированная функция должна определять один сегмент кода, работающий в различных обстоятельствах. Можно обобщить функцию *moveBall( )* так, чтобы она перемещала *любой* клип на *любое* расстояние в *любом* направлении. Первый шаг в обобщении любой функции состоит в определении факторов, управляющих 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 желательно задавать при вызове функции.

## Создание функций с параметрами

Вспомните общий синтаксис объявления простой функции:

```
function funcName () {
  statements
}
```

Чтобы добавить *параметры*, являющиеся переменными, которые можно использовать в функции, мы передаем список допустимых идентификаторов в скобках объявления функции. Параметры разделяются запятыми, как показано ниже:

```
function funcName (param1, param2, param3,...paramn) {
  statements
}
```

Если определены параметры, можно обращаться к ним из тела функции, как к любым другим переменным, например:

```
function say(msg) {                // Определить параметр msg
  trace("The message is " + msg);  // Использовать msg в предложении trace()
}
```

В нашем определении функции указан параметр msg. Предложение *trace( )* использует этот параметр, как любую другую переменную, выводя ее значение в окно Output.

Зададим параметры функции *moveBall( )*, чтобы при каждом ее выполнении можно было устанавливать различные имя клипа, расстояние по горизонтали и расстояние по вертикали. Соответствующий 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( )*. Кроме того, мы определили три параметра: theClip (клип, который нужно переместить), xDist (величина перемещения по горизонтали) и yDist (величина перемещения по вертикали). Внутри тела функции мы используем эти параметры вместо жестко закодированных в исходном варианте значений, позволяя тем самым перемещать любой клип на любое расстояние по горизонтали и вертикали.

## Вызов функций с параметрами

Когда мы создаем функцию, то задаем имена параметров, которые, в сущности, являются заместителями значений: при вызове функции передаются значения, которые ставятся вместо каждого из параметров.

Слова *параметры* и *аргументы* используются в этой книге, как и в большей части документации, взаимозаменяемым образом. Формально *аргументы* являются значениями, используемыми при вызове функции, а *параметры* являются заместителями 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, используемые в качестве аргументов, могут быть любыми допустимыми выражениями ActionScript, в том числе составными выражениями. Например, ранее мы определили простую функцию *say( )*, которая требует единственного параметра msg:

```
function say(msg) {
  trace("The message is " + msg);
}
```

Чтобы вызвать *say( )*, мы используем такое предложение:

```
say("This is my first argument...how touching");
```

Или такое:

```
say(99);
```

Обратите внимание, что значения «This is my first argument ... how touching» и 99 принадлежат различным типам данных. ActionScript разрешает передавать функции данные любого типа, если только функция знает, что делать с переданным значением. (Языки типа C требуют предварительно объявить тип данных для каждого параметра и выводят ошибку при передаче данных неверного типа.)

Перед передачей любого аргумента его значение полностью вычисляется. Поэтому можно вызывать функции со сложными выражениями, например:

```
var name = "Gula";
say("Welcome to my web site " + name);
```

Поскольку выражение "*Welcome to my web site* " + *name* вычисляется до передачи его функции *say( )*, функция получает значение «Welcome to my web site Gula». Это означает, что можно генерировать аргументы функции программным образом. Мощная вещь.

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`. Любой клип можно переместить на любое расстояние путем вызова общей функции *moveClip( )* с различными аргументами. Переместим экземпляр клипа с именем `square` на 2 пиксела вправо и на 100 пикселов вниз:

```
moveClip(square, 2, 100);
```

# Выход из функций и возвращение значений

В отсутствие иных указаний функция завершает работу естественным путем, когда интерпретатор заканчивает выполнение последнего предложения в теле функции. Можно, однако, завершить функцию, не доходя до последнего предложения. Кроме того, функция может возвра-

щать результат (передавать вычисленное значение) в вызвавший ее код . Посмотрим, как это действует.

## Завершение выполнения функции

Предложение *return*, с которым мы познакомились в главе 6 «Предложения», можно использовать для завершения выполнения функции и возвращения результата. Когда интерпретатор во время выполнения функции сталкивается с предложением *return*, он пропускает все оставшиеся в функции предложения. Рассмотрим такой пример:

```
function say(msg) {
  return;
  trace(msg);          // Эта строка никогда не достигается
}
```

Этот пример нереалистичен, потому что предложение *return* всегда вызывает завершение функции до того, как будет достигнуто предложение *trace( )*. Поэтому предложение *return* обычно является последним предложением функции, если только оно не используется внутри условного предложения. В следующем примере *return* используется для выхода при неверном пароле:

```
var correctPass = "cactus";
function enterSite(pass) {
  if (pass != correctPass) {
    // Выйти, если пароль неверен
    return;
  }
  // Этот код достигается 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 возвратиться в место вызова функции. Если предложение *return* отсутствует, ActionScript действует так, как если бы последняя строка функции содержала предложение *return*:

```
function say(msg) {
  trace(msg);
  return;              // В таком контексте эта строка совершенно лишняя
}
```

Независимо от того, является предложение *return* подразумеваемым или явным, по окончании работы функции выполнение возобновляется со строчки кода, следующей за предложением вызова функции. Например:

```
say ("Something");  // Выполняется код функции say()
// По завершении функции say() выполнение возобновляется здесь
trace ("Something else");
```

## Возвращение значений функциями

Как мы видели, *return* всегда завершает функцию. Но его можно также использовать для возврата значения в сценарий, который вызвал функцию, с помощью следующего синтаксиса:

```
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);  // Sets total to 3
```

Выражение или результат, возвращаемое предложением *return*, называется *возвращаемым значением* функции.

Обратите внимание, что наша функция *combine( )* просто вычисляет и возвращает сумму двух чисел (либо конкатенирует две строки). Она не совершает явного действия, как функция *sayHi( )*, которая выводила сообщение, или *moveClip( )*, которая перемещала клип. Использовать возвращаемое функцией значение можно, присвоив его переменной:

```
var total = combine (5, 6);               // Устанавливает total в 11
var greet = combine ("Hello ", "Cheryl")  // greet будет "Hello Cheryl"
```

Результатом вызова функции является 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 в качестве аргументов 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 как

```
var phrase = combine (combine(5,6), " people were at the party");
```

Здесь 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 предложении *return* не содержится выражения, которое должно возвращаться, или предложение *return* опущено целиком, функция возвращает значение `undefined`. На практике это является частой причиной ошибок. Например, следующий код не делает ничего полезного, поскольку отсутствует предложение *return*:

```
function combine(a, b) {
  var result = a + b;  // Результат вычисляется, но не возвращается
}
```

Аналогично, не верен и такой код:

```
function combine(a, b) {
  var result = a + b;
  return;  // Вы забыли указать возвращаемое значение
}
```

При создании функции, которая должна возвращать результат вычислений, не забудьте поместить в нее предложение *return*, которое действительно возвращает 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*, которые удобно использовать, 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 имя функции опускается и после блока предложений ставится точка с запятой. Общий формат такой:

```
function (param1, param2, ... paramn) { statements };
```

где *param1,param2,...paramn* является необязательным списком параметров, а *statements* – одно или более предложений, образующих тело функции. Так как имени функции нет, литерал 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( )* языка JavaScript, динамические функции нельзя составлять на этапе выполнения, как это можно делать в JavaScript.

# Доступность и срок жизни функций

Как и переменные, функции живут не вечно, и к ним нельзя непосредственно обращаться из любого места фильма. Чтобы с уверенностью вызывать функции, нужно знать, как обращаться к ним из различных мест фильма, и перед вызовом убедиться в их существовании.

## Доступность функций

Программные функции (те функции, которые мы сами создаем в коде) *непосредственно* доступны для вызова только с помощью:

- Кода, прикрепленного к временной диаграмме клипа, в котором находится объявление функции;
- Кнопки на временной диаграмме клипа, в котором находится объявление функции.

Под «непосредственной доступностью» мы имеем в виду, что функцию можно вызвать просто по имени, без ссылки на клип или объект, например:

```
myFunction();
```

Функции доступны *косвенно* (т. е. удаленным образом) из любой точки фильма с использованием синтаксиса точки. Как и при удаленной ссылке на переменные, необходимо задать путь к клипу, который содержит 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, используя абсолютный путь к `rectangle`, например:

```
_root.rectangle.area();
```

Для обращения к функции из удаленной временной диаграммы клипа используются те же правила, что и для ссылки на удаленные переменные, описанные в главе 2 «Переменные». (Дополнительно о вызове удаленных функций мы расскажем в главе 13 «Клипы».)

Не все функции прикрепляются к временным диаграммам клипов. Некоторые программные функции могут прикрепляться к объектам, встроенным или определенным пользователем. Если функция прикреплена к объекту, она доступна только через своего хозяина-объекта. Подробнее о доступности и сроке жизни *методов* (т. е. функций, прикрепленных к объектам), мы узнаем в главе 12.

## Срок жизни функций

Функция, определенная на временной диаграмме клипа, теряется при удалении этого клипа с рабочего стола. Попытка вызвать функцию, которая определена в не существующем более клипе, окажется неудачной, и дополнительного уведомления об этом не последует. Лучшим способом обеспечить постоянное присутствие функции является ее определение на главной временной диаграмме фильма, потому что главная временная диаграмма сохраняется на протяжении всего фильма.

Обратите внимание, что функции в сценарии кадра инициализируются и становятся доступными, когда воспроизводящая головка впервые входит в кадр, содержащий сценарий. Поэтому можно использовать вызов функции до ее объявления, если и вызов и объявление находятся в одном сценарии. Например:

```
// Вызов функции tellTime() перед ее объявлением
tellTime();

// Объявление функции tellTime()
function tellTime() {
  var now = new Date();
  trace(now);
}
```

# Область видимости функции

Предложения ActionScript имеют определенную *область видимости* (*scope*), или область действия, в которой они допустимы. Когда предложение прикрепляется к клипу, область видимости этого предложения ограничивается клипом, который его содержит. Например, ниже мы ссылаемся на переменную `score` в предложении присваивания:

```
score = 10;
```

Если это предложение прикреплено к клипу clipA, то интерпретатор присвоит значение переменной score, находящейся в clipA, потому что область видимости этого предложения ограничена clipA. Если это предложение прикреплено к клипу clipB, то интерпретатор присвоит значение переменной score, находящейся в clipB, потому что область видимости этого предложения ограничена clipB. Местонахождение предложения определяет его область видимости, а следовательно, его действие.

Предложения в теле функции действуют в своей собственной отдельной области видимости, называемой *локальной областью видимости*. Локальная область видимости – это как личная телефонная кабинка для функции, отдельная от области видимости клипа или объекта, к которым она прикреплена. Локальная область видимости функции создается при ее вызове и уничтожается по завершении выполнения функции. При разрешении переменных, на которые ссылаются предложения тела функции, интерпретатор ищет их сначала в области видимости функции.

Например, параметры функции определяются в ее *локальной* области видимости, а не в области видимости временной диаграммы, содержащей функцию. Поэтому параметры доступны предложениям в теле функции, только когда функция выполняется. Предложения вне функции не имеют доступа к параметрам функции.

Локальная область видимости функции предоставляет пространство для временных переменных, используемых исключительно внутри функции. Этим предотвращаются возможные конфликты имен между переменными функции и переменными временной диаграммы, а также сокращается общий расход памяти.

## Цепочка областей видимости

Несмотря на то, что функции действуют в собственных локальных областях видимости, обычные переменные временной диаграммы сохраняют свою доступность для предложений тела 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 объект или клип, содержащие функцию.

Допустим, например, что мы определили переменную firstName на временной диаграмме клипа clipA. Определим также функцию *getName( )* в клипе clipA. Внутри функции *getName( )* мы ссылаемся на переменную firstName в предложении *trace( )*:

```
firstName = "Christine";

function getName () {
  trace(firstName);
```

```
}

getName();
```

Когда мы вызываем *getName( )*, интерпретатор должен найти значение `firstName`. Переменной с именем `firstName` в локальной области видимости *getName( )* нет, поэтому интерпретатор станет искать переменную `firstName` в `clipA`. Там он найдет `firstName` и выведет ее значение, «Christine».

Наше предложение *trace( )* может ссылаться на переменную временной диаграммы из тела функции *getName( )*, потому что в самой *getName( )* не определены параметр или 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 присваиваем значение «Kathy» параметру `firstName`. Когда интерпретатор выполняет предложение *trace( )*, он осуществляет поиск в локальной области видимости и находит там параметр `firstName` со значением «Kathy». Поэтому функция выводит на сей раз «Kathy», а не «Christine». Несмотря на то что существует переменная временной диаграммы `firstName`, предпочтение отдается локальной переменной с именем `firstName`.

Этот пример иллюстрирует действие *цепочки областей видимости* (*scope chain*) – иерархии объектов, используемых интерпретатором для разрешения ссылок на переменные и свойства объектов. Для функций, прикрепленных к временным диаграммам, цепочка областей видимости имеет только два уровня: локальная область видимости и область видимости клипа, содержащего функцию. Но когда мы прикрепляем функции к клиентским объектам и классам, цепочка областей видимости может иметь значительно больше уровней, как мы увидим в разделе «Наследование свойств объектов» в главе 12.

Если область видимости переменных и свойств, к которым нужно обращаться из тела функции, отличается от области видимости предложения, следует использовать синтаксис точки для создания явной ссылки. Например:

```
function dynamicGoto() {
  // Намеренный выход за пределы локальной области видимости
  _root.myClip.gotoAndPlay(_root.myClip.targetFrame);
}
```

Обратите внимание, что цепочка областей видимости функции определяется относительно предложения, содержащего *объявление* функции, а не предложения, содержащего ее *вызов*. То есть область видимости кода в теле функции – клип, в котором находится объявление функции, а не клип, содержащий предложение с вызовом функции.

Вот пример, показывающий ошибочное использование 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`, будет повернут весь главный фильм, а не только `clipA`:

```
// КОД ВО ВРЕМЕННОЙ ДИАГРАММЕ clipA
_root.rotate(30);  // Ой! Это поворачивает весь фильм
```

Эту ситуацию можно исправить, передав клип, который должен быть повернут, в качестве аргумента функции `rotate()`:

```
function rotate (theClip, degrees) {
  theClip._rotation += degrees;
}

// Вызвать rotate() со ссылкой на clipA
_root.rotate(clipA, 30);
```

## Локальные переменные

Переменные, определяемые в локальной области видимости функции, называются *локальными переменными*. Локальные переменные, в том числе параметры, доступны только предложениям в теле функции, в которой они определены, и существуют только во время выполнения этой функции. Чтобы создать локальную переменную (отличную от параметров, которые автоматически становятся локальными переменными), используется предложение *var* внутри любой функции, например:

```
function funcName() {
  var temp = "just testing!";  // Объявляет локальную переменную temp
}
```

Локальные переменные полезны для временного хранения информации. В следующем примере мы используем локальную переменную `lastSpacePlusOne` для хранения временного результата. Как и все локальные переменные, она умирает 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 освобождается. Хранение всех временных значений в локальных переменных позволяет сберечь память. Кроме того, при объявлении локальной переменной не нужно 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 изнутри функции можно читать переменные временной диаграммы. Но можно также создавать их и осуществлять в них запись. Предложения с присваиваниями переменным, которых нет в локальной области видимости, выполняются с переменными временной диаграммы. В следующем примере `x` является локальной переменной, но `y` и `z` являются переменными 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 (он уничтожился после завершения функции)
trace(y);         // y равен 13 (y существует после конца функции)
trace(z);         // z равен 2 (z был устойчиво изменен функцией)
```

Правила для создания переменных временной диаграммы применимы, даже если наша функция является методом объекта. Мы не станем рассказывать об объектах до главы 12, но для тех, кто знаком с объектно-ориентированным программированием, в примере 9.2 приведен код, доказывающий это предложение. Обратите внимание, что `x` определяется на временной диаграмме, потому что она не существует ни в локальной области видимости *newFunc( )*, ни как свойство `newObj`.

*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 неопределен
```

# Еще раз о параметрах функции

Теперь, когда мы освоились с функциями, вернемся к теме их параметров. Для данного обсуждения требуется некоторое знание объектов, поэтому новичкам в программировании сначала может потребоваться ознакомление с главой 12.

## Число параметров

Ранее мы узнали, что параметры функции объявляются при создании функции. Вспомним синтаксис:

```
function funcName (param1, param2, param3,...paramn) {
  statements
}
```

На первый взгляд удивительно, что число параметров, передаваемых функции, может отличаться от числа параметров, указанных в формальном объявлении функции. Функции могут принимать любое количество параметров – как больше, так и меньше «расчетного» числа. Если функция вызывается с меньшим количеством параметров, чем объявлено, 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);  // Выводит: "x is 10", "y is ", and "z is ", потому что
               // y и z не определены (и выводятся как пустое место)
```

Если функция вызывается с количеством параметров, *большим* объявленного, значения лишних параметров можно получить с помощью объекта *arguments*. (Очевидно, к лишним параметрам нельзя обратиться по имени, как к тем, которые были явно объявлены.)

## Объект arguments

Во время выполнения любой функции встроенный объект *arguments* предоставляет доступ к трем видам информации: a) количеству параметров, которые были переданы функции, b) массиву, содержащему значения всех этих параметров и c) имени выполняемой функции. Объект *arguments* является, в действительности, некоторым особым

гибридом между массивом и объектом, обладающим также некоторыми другими свойствами.

## Извлечение значений параметров из массива

Массив `arguments` позволяет получить значение любого параметра функции независимо от того, был ли он определен формально в объявлении функции. Для доступа к параметру рассматриваются индексы массива `arguments`:

```
arguments[n]
```

где *n* является индексом параметра, к которому мы обращаемся. Первый параметр (стоящий первым слева в выражении вызова функции) хранится под индексом 0, и обращение к нему выглядит так: `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 количество элементов в `arguments`:

```
arguments.length
```

Можно легко организовать цикл по всем параметрам, переданным функции, и вывести их в окно Output, как показано в примере 9.3.

*Пример 9.3. Вывод неизвестного числа параметров*

```
function showArgs() {
  for (var i = 0; i < arguments.length; i++) {
    trace("Parameter " + (i + 1) + " is " + arguments[i]);
  }
}

showArgs(123, 23, "skip intro");

// Выводит...
Parameter 1 is 123
Parameter 2 is 23
Parameter 3 is skip intro
```

Массив `arguments` позволяет создавать очень гибкие функции, которые принимают произвольное число параметров.

Вот общая функция, которая удаляет с рабочего стола имеющиеся на нем 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 приему произвольного числа параметров? Как насчет функции, возвращающей среднее значение произвольного количества чисел? (Подсказка: сложите все аргументы, а затем разделите на количество аргументов.)

### Свойство callee

Как мы видели, массив `arguments` позволяет извлекать параметры функции. У объекта *arguments* есть одно свойство, `callee`, которое возвращает ссылку на выполняемую функцию. Обычно нам известно имя вызываемой функции, но если выполняется анонимная функция, созданная с помощью литерала функции, свойство `callee` может оказаться полезным. В примере 9.4 показана функция, созданная с помощью литерала, которая выполняется рекурсивно, не зная собственного имени. Смотрите раздел «Рекурсивные функции», расположенный далее в этой главе.

*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 этой главе позже.

## Элементарные и составные значения параметров

Мы должны рассмотреть еще одну тонкость в использовании параметров: различие между передачей в функцию *элементарных* и *составных* данных.

Когда мы передаем в качестве аргументов функции элементарные данные, функция получает копию данных, а не их оригинал. Изменения, произведенные над 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 элементарные данные передаются, как говорят, *по значению*. Однако когда мы передаем в качестве аргумента функции *составные* данные, функция получает не дубликат данных, а *ссылку*, которая указывает на те же данные, что и исходный аргумент. Изменение данных через переменную параметра воздействует на исходные данные, а потому и на все переменные, которые указывают на те же данные, даже вне функции. Поэтому составные данные передаются, как говорят, *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() задает значение первого элемента массива
function setValue(myArray) {
  myArray[0] = "Sid";   // задать 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 хотя мы можем изменять значения отдельных элементов массива изнутри функции, присваивание нового значения параметру разорвет его связь с первоначальным аргументом. Последующие изменения переменной параметра не окажут влияния на исходный аргумент. В примере 9.7 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 15 «Более сложные темы».

# Рекурсивные функции

*Рекурсивной* называется функция, вызывающая сама 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 исчерпает всю память, вызвав сообщение об ошибке:

```
function trouble() {
  trouble();
}
```

Реальные рекурсивные функции вызывают себя только тогда, когда удовлетворяется заданное условие (предотвращая таким образом *бесконечную рекурсию*). В примере 9.4 рекурсия использовалась для обратного счета от заданного числа до 1, но очевидно, что можно осуществить это 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. Пример 9.8 показывает функцию факториала, которая использует рекурсию.

*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 обычно, есть несколько способов сделать одно и то же дело. 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;
    }
}
```

Примеры 9.8 и 9.9 представляют два способа решения одной и той же проблемы. Метод рекурсии говорит: «Факториал 6 – это 6, умноженное на факториал 5. Факториал 5 – это 5, умноженное на факториал 4 ...», и так далее. Нерекурсивный метод проходит в цикле числа от 1 до x и перемножает их все в одно большое число.

Какой подход лучше выбрать, зависит от задачи. Некоторые проблемы легче решаются с использованием рекурсии, но рекурсия может оказаться медленнее нерекурсивных решений. Лучше всего использовать рекурсию, когда вы не знаете, насколько глубоко вложенной может быть структура данных. Например, допустим, вы хотите перечислить все файлы во всех вложенных подкаталогах, до бесконечности. Нельзя 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);
}
```

Когда мы будем рассматривать объект *XML* в части III «Справочник по языку», то рекурсия будет использоваться для перечисления всех элементов документа.

# Внутренние функции

Хотя мы разобрались, как вызывать пользовательские функции, не будем забывать, что ActionScript поставляется вместе с набором встроенных функций (родственных глаголам в нашей языковой аналогии). Мы уже видели некоторые встроенные функции, которые позволяют 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. Надеемся, вы обратили внимание, что вызываете встроенные функции с помощью оператора вызова функции (круглые скобки) и списка параметров (в данном случае значение 5) в точности, как обычные функции, определенные пользователем! Встроенные функции, например *gotoAndPlay( )*, используются точно так же, как функции, которые мы строим сами. Естественно, встроенные функции делают иные вещи, нежели наши пользовательские функции, и нет смысла писать собственную функцию, выполняющую то же, что уже имеющаяся встроенная функция ActionScript. Но, как и у любой пользовательской функции, у каждой встроенной функции есть имя, необязательные параметры и возвращаемое значение (иногда являющееся `undefined`).

Хотя Flash давно называет *gotoAndPlay* «действием» («Action»), мы теперь видим его подлинный вид – это внутренняя функция. В главе 6 мы узнали, что некоторые действия Flash являются предложениями, а некоторые – функциями. Теперь, тщательно изучив те и другие, вы можете отличать одно от другого.

Рассматривая некоторые действия как *функции*, вам будет легче запомнить и использовать их синтаксис. Например, для загрузки фильма в проигрыватель Flash нужно знать о существовании функции *loadMovie( )* и ожидаемых ею параметрах. После краткого визита в часть III мы находим нужные данные и можем легко составить предложение вроде следующего:

```
loadMovie("myMovie.swf", 1);
```

Это легкое дело, если вы освоились с функциями.

Встроенные функции ActionScript многочисленны и разнообразны. Они позволяют управлять элементами фильма, наделяя нас возможностью исследовать и изучить все – от громкости звука до объема текста, выделенного в поле для редактирования. Исчерпывающий список

встроенных функций ActionScript можно найти в части III. Периодически просматривайте его, чтобы ближе познакомиться с типами имеющихся функций, хотя запоминать их специфический синтаксис нет необходимости.

## Доступность внутренних функций

Некоторые встроенные функции доступны всюду, в то время как другие могут вызываться только через объект. Если функция является *глобальной*, ее можно использовать всюду; однако если она служит *методом* объекта, то должна использоваться вместе с этим 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. Давайте посмотрим, что это значит и как влияет на то, что можно делать с помощью функций.

## Передача функций в функции

Это может показаться удивительным, но любую функцию можно использовать в качестве аргумента для другой функции, например, следующим образом:

```
function1(function2);
```

Обратите внимание, что если за *function2* нет круглых скобок, то интерпретатор не выполняет *function2( )*, а просто передает ее «ссылку на объект» в качестве аргумента *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)
```

Поскольку функции являются типом объекта, можно обращаться с ними, как с любыми другими данными. В следующем примере внут-

ренняя функция *gotoAndPlay* присваивается переменной gp, что дает более краткий способ ссылки на функцию:

```
gp = gotoAndPlay;  // Создает краткую ссылку на gotoAndPlay()
gp(25);            // Вызвать gotoAndPlay() посредством нашей ссылки
```

Помимо передачи и хранения функций как объектов, «объектность» функций можно использовать, 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.**

Свойства функций обладают возможностями локальных переменных не теряться между вызовами функции. Это полезно, когда нужно вызвать функцию с уникальным идентификатором. Вот, например, общая функция, которая делает копию клипа и дает этой копии уникальные имя и уровень:

```
makeClip.count = 0;     // Определить свойство makeClip() (вспомните,
                        // что makeClip() уже существует, так как
                        // функции определяются до выполнения кода)

  // Сделать копию переданного клипа и присвоить
  // новому клипу имя автоматически
function makeClip (theClip) {
  // Прибавить единицу к счетчику клипов
  makeClip.count++

  // Теперь копировать клип, присвоив ему уникальные имя и глубину
  theClip.duplicateMovieClip(theClip._name + makeClip.count,
makeClip.count);
}

makeClip(square);   // Сделать дубликат square с помощью makeClip()

square1._x += 100;  // Переместить продублированный квадрат вправо
```

# Централизация кода

Вероятно, самым важным свойством функций является возможность их повторного использования: можно создавать многократно используемые участки кода, которые можно выполнять из любого места кода. Так как функции могут выполняться удаленно, их можно хранить в одном общем месте, что облегчает сопровождение и модификацию кода.

Допустим, например, что в трех разных клипах есть три кнопки, имеющие одинаковое назначение. Вместо утроения кода можно поместить один экземпляр кода в функцию на главной временной диаграмме и вызывать его из каждой кнопки по мере надобности. Это экономит время, сокращает возможность появления ошибок и позволяет легко изменять поведение сразу трех кнопок. Благодаря централизации кода тестирование становится более надежным, а выявление ошибок значительно упрощается. Если одна из кнопок работает, а другая – нет, 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 точной 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 ответов код был разбросан по всему фильму. Мы разместили логику вопросника в трех местах: в первом кадре, где мы инициализировали наш вопросник, в кнопках, где мы сохраняли ответы пользователя, и в кадре `quizEnd`, где мы подводили счет правильных ответов. Теперь мы хотим централизовать логику вопросника, выполняя все эти задачи в первом кадре. Для инициализации мы будем использовать тот же ряд простых предложений, что и в первой версии, но для сохранения ответов пользователя и подсчета результата воспользуемся двумя функциями, *answer( )* и *gradeUser( )*.

Пример 9.10 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.

*Пример 9.10. Вопросник с несколькими вариантами выбора, версия 2*

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

//  Инициализировать переменные главной временной диаграммы
var displayTotal;          // Текстовое поле для вывода результата
var numQuestions = 2;      // Количество вопросов
var q1answer;              // Ответ пользователя на вопрос 1
var q2answer;              // Ответ пользователя на вопрос 2
var totalCorrect = 0;      // Количество правильных ответов
var correctAnswer1 = 3;    // Правильный вариант ответа для вопроса 1
var correctAnswer2 = 2;    // Правильный вариант ответа для вопроса 2

//  Функция, регистрирующая ответы пользователя
function answer (choice) {
  answer.currentAnswer++;
  set ("q" + answer.currentAnswer + "answer", choice);
  if (answer.currentAnswer == numQuestions) {
    gotoAndStop ("quizEnd");
  } else {
    gotoAndStop ("q" + (answer.currentAnswer + 1));
  }
}

//  Функция для подсчета результата пользователя
function gradeUser() {
  // Подсчитать количество правильных ответов
  for (i = 1; i <= numQuestions; i++) {
    if (eval("q" + i + "answer") == eval("correctAnswer" + i)) {
      totalCorrect++;
    }
  }

  // Вывести результат пользователя в текстовом поле на экране
  displayTotal = totalCorrect;
}
```

Первым делом останавливаем фильм с помощью функции *stop( )*, чтобы он не прокрутил все вопросы. Затем инициализируем переменные временной диаграммы вопросника. Вы узнаете `displayTotal`, `totalCorrect`, `q1answer` и `q2answer` из первой версии. Мы также добавили `numQuestions` (которую будем использовать в функциях *answer( )* и *gradeUser( )*), а также `correctAnswer1` и `correctAnswer2` (которые будем использовать при оценивании ответов).

Когда переменные инициализированы, можно создать функцию *answer( )*, которая будет регистрировать ответы пользователя и перемещать воспроизводящую головку к следующему вопросу. Функция *answer( )* нуждается в одном параметре, `choice`, который является номером ответа, выбранного пользователем для каждого вопроса, поэтому объявление функции начинается так:

```
function answer (choice) {
```

Когда дан ответ, функция увеличивает currentAnswer – свойство функции, хранящее номер вопроса, на который дается ответ:

```
answer.currentAnswer++;
```

Затем мы записываем выбор пользователя в динамически именуемую переменную временной диаграммы, соответствующую вопросу, на который дается ответ. Далее мы используем значение свойства currentAnswer, чтобы определить имя переменной временной диаграммы (q1answer, q2answer и т. д.):

```
set ("q" + answer.currentAnswer + "answer", choice);
```

Когда ответ пользователя сохранен в соответствующей переменной, мы переходим к концу вопросника, если это был последний вопрос, или к следующему вопросу, находящемуся в кадре с меткой *"q" + (answer.currentAnswer + 1)*:

```
if (answer.currentAnswer == numQuestions) {
  gotoAndStop ("quizEnd");
} else {
  gotoAndStop ("q"+ (answer.currentAnswer + 1));
}
```

Это составляет логику наших вопросов-ответов. Функция *answer( )* готова обработать ответы на любой вопрос вопросника. Теперь построим функцию, оценивающей эти ответы – *gradeUser( )*.

Функция *gradeUser( )* не принимает параметров. Она должна сравнить ответ каждого пользователя с правильным ответом и вывести результат пользователя. Эти сравнения мы производим в цикле *for*: тело цикла проходится столько раз, сколько вопросов в вопроснике:

```
for (i = 1; i <= numQuestions; i++) {
```

Внутри цикла выражение сравнения сравнивает ответы пользователя с правильными ответами. С помощью *eval( )* мы динамически извлекаем значение каждой переменной ответа пользователя и каждую переменную правильного ответа. Если эти две переменные равны, инкрементируем totalCorrect:

```
if (eval("q" + i + "answer") == eval("correctAnswer" + i)) {
  totalCorrect++;
}
```

После завершения цикла totalCorrect будет содержать количество вопросов, на которые пользователь ответил правильно. Мы выводим это число, присваивая динамическому текстовому полю displayTotal значение totalCorrect:

```
  displayTotal = totalCorrect;
```

Вот! Теперь, когда обе функции на месте и переменные вопросника инициализированы, логика нашего вопросника завершена. Вы заметили, насколько проще следить за действием вопросника, когда большая часть кода содержится в одном кадре? Осталось только вызывать функции из надлежащих мест вопросника.

## Вызов функций вопросника

Мы будем вызывать функцию *answer( )* из кнопок, по которым щелкает пользователь при выборе ответа, а функцию *gradeUser( )* – из кадра `quizEnd`.

К кнопке первого ответа в кадре 1 прикрепим такой код:

```
on (release) {
  answer(1);
}
```

Ко второй кнопке – такой:

```
on (release) {
  answer(2);
}
```

А к третьей кнопке прикрепим следующий код:

```
on (release) {
  answer(3);
}
```

Затем точно такой же код мы прикрепим к трем кнопкам в кадре `q2`. Обратите внимание на элегантность процесса. Когда код, требуемый для каждой кнопки, аккуратно оформлен в виде функции, код кнопки сводится к минимуму. Логика ответа и перемещения по вопроснику не дублируется излишне, как было в предыдущей версии вопросника. Кроме того, создание новых кнопок превращается просто в изменение числа, передающегося в функцию *answer( )*. Для последнего вопроса тоже не нужно делать ничего особенного, поскольку функция *answer( )* обрабатывает это автоматически.

Когда код прикреплен к кнопкам, нам остается только оценить пользователя в конце опроса, прикрепив к кадру `quizEnd` следующий код:

```
gradeUser();
```

Опять же, поместив функции оценивания в первый кадр, мы отделили *механику* системы от ее *использования*. В результате мы не потеряем эту механику в той путанице кадров и кнопок, которую часто представляют собой 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 задумывайтесь над тем, какие из изученных вами приемов можно применить к нашему примеру вопросника. Какие наглядные особенности можно добавить для пользователя? Какие невидимые изменения можно сделать в коде, чтобы он стал более гибким и элегантным, даже если ощущения пользователя при этом не изменятся?

# Вперед!

В этой главе рассмотрено 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 служат одним из краеугольных камней интерактивности.

# 10

# События и обработчики событий

Мы много узнали о составлении команд, которые должен выполнять ActionScript. К настоящему времени мы вполне можем сообщать интерпретатору, *что* от него нужно, 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 воспроизводящая головка входит в новый кадр, интерпретатор выполняет код, прикрепленный к этому кадру. После того как выполнен код кадра, обновляется изображение на экране и воспроизводятся звуки. Затем воспроизводящая головка переходит к следующему кадру.

Например, если поместить код непосредственно в кадр 1 фильма, то этот код выполняется перед выводом содержимого кадра 1. Если мы поместим блок кода в ключевой кадр в кадре 5 того же фильма, будут выведены кадры с 1 по 4, затем выполнен код кадра 5 и, наконец, 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 из-за выполнения команд *gotoAndPlay( )* или *gotoAndStop( )*, код всех кадров

выполняется в предсказуемом порядке, синхронно с перемещением воспроизводящей головки.

# Асинхронное выполнение кода, основанное на событиях

Некоторая часть кода выполняется в непредсказуемом порядке. Этот код выполняется, когда интерпретатор ActionScript обнаруживает, что произошло одно из предопределенных событий. Во многих *событиях* участвуют действия пользователя, например, щелчок мыши или нажатие клавиши. Так же, как вход воспроизводящей головки в новый кадр вызывает выполнение синхронного кода, прикрепленного к этому кадру, события могут вызывать выполнение кода, *основанного на событиях* (*event-based*). О коде, основанном на событиях (т. е. 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 собственно интерактивности.

В этой главе исследуется асинхронное (основанное на событиях) программирование во Flash и перечисляются различные события, поддерживаемые ActionScript.

# Типы событий

События можно разбить на две категории:

*Пользовательские события*

Действия, выполняемые пользователем (например, щелчок мыши или нажатие клавиши).

*Системные события*

То, что происходит во время внутреннего воспроизведения фильма (например, появление клипа на рабочем столе или загрузка ряда переменных из внутреннего файла).

Синтаксически ActionScript не различает пользовательские и системные события. Событие, запускаемое фильмом внутри себя, не менее осязаемо, чем щелчок кнопки мыши пользователем. Хотя обычно удаление, скажем, клипа фильма с рабочего стола может не восприниматься как достойное внимания «событие», способность реагировать на системные события дает нам возможность лучше управлять фильмом.

Разделение событий ActionScript на категории можно произвести и более практично – на основе объектов, к которым они относятся. Все события происходят относительно некоторого объекта в среде Flash. Это значит, что интерпретатор не просто сообщает, что пользователь щелкнул мышью, а сообщает, что пользователь щелкнул мышью по *этой кнопке* или что пользователь щелкнул мышью, когда *этот клип* был на рабочем столе. И интерпретатор не говорит, что были получены данные, а сообщает, что *этот клип* получил некоторые данные. Мы определяем код, который реагирует на события, в объектах, с которыми связаны эти события.

Объектами ActionScript, которые могут получать события, являются:

- Клипы
- Кнопки
- Объекты классов *XML* и *XMLSocket*

Как мы увидим в этой главе, в ActionScript фактически есть две разные реализации событий: одна для событий, связанных с клипами и кнопками, и другая для всех остальных типов объектов.

# Обработчики событий

Не все события запускают какой-либо код. Часто они происходят, не оказывая воздействия на фильм. Например, пользователь может генерировать десятки событий, многократно щелкая по кнопке, но эти щелчки могут игнорироваться. Почему? Потому что сами по себе события не могут вызвать выполнение кода: мы должны написать код, явно реагирующий на события. Чтобы приказать интерпретатору выполнить некоторый код в ответ на событие, мы должны вставить так называемый *обработчик события* (*event handler*), в котором описываются действия, совершаемые, когда происходит указанное событие. Обработчики событий названы так потому, что они *обрабатывают,* или *перехватывают,* события в фильме.

Обработчик событий похож на особым образом названную функцию, которая автоматически вызывается при возникновении определенного события. Поэтому создание обработчика события очень похоже на создание функции, но имеет несколько особенностей:

- Обработчики событий имеют предопределенные имена, такие как *keyDown*. Нельзя дать обработчику события произвольное имя: нужно использовать предопределенные имена, которые будут показаны ниже в табл. 10.1 и табл. 10.2.
- Обработчики событий не нужно объявлять с помощью предложения *function.*
- Обработчики событий должны прикрепляться к кнопкам, клипам или объектам, но не к кадрам.

Большинство событий впервые появилось во Flash 5. При экспорте в формат Flash 4 используйте только обработчики событий для кнопок (во Flash 4 поддерживались только события для кнопок) и тщательно проверяйте результат в проигрывателе Flash 4.

# Синтаксис обработчика событий

Названия событий (и соответствующих обработчиков событий) в ActionScript предопределены. Обработчики событий для кнопок определяются как *on* (*eventName*), а обработчики событий клипов определяются как *onClipEvent* (*eventName*), где *eventName* является именем обрабатываемого события.

Поэтому все обработчики событий кнопок (кроме *keyPress*, для которого дополнительно требуется параметр *key*) имеют вид:

```
on (eventName) {
  statements
}
```

Обработчик для одной кнопки может отвечать на несколько событий, разделенных запятыми. Например:

```
on (rollover, rollOut) {
  // Вызвать пользовательскую функцию в ответ
  // на события rollOver и rollOut
  playRandomSound( );
}
```

Все обработчики событий клипов имеют вид:

```
onClipEvent (eventName) {
  statements
}
```

В отличие от обработчиков для кнопок, обработчики для клипов могут реагировать только на одно событие.

# Создание обработчиков событий

Чтобы создать обработчик для события, нужно определить обработчик и прикрепить его к соответствующему объекту. Мы начнем с самых распространенных обработчиков – тех, которые прикрепляются к кнопкам и клипам.

## Прикрепление обработчиков событий к кнопкам и клипам

Чтобы прикрепить обработчик события к кнопке или клипу, нужно физически расположить код функции обработчика в нужных кнопке или клипе. Сделать это можно только в среде разработки Flash, выбрав объект на рабочем столе и введя требуемый код в панели Actions, как показано на рис. 10.1.

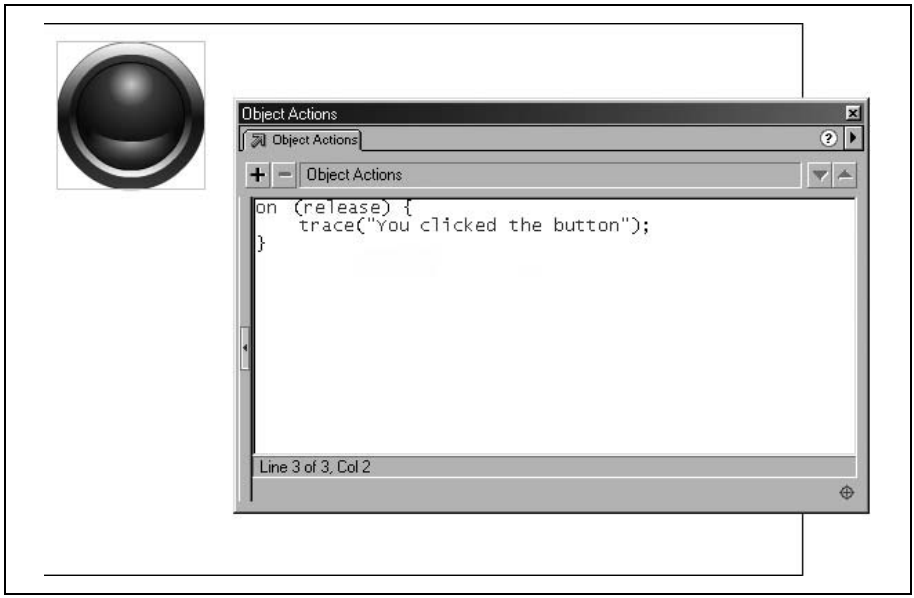


***Рис. 10.1.*** *Прикрепление обработчика событий к кнопке*

Попробуем создать простую функцию обработчика события для кнопки и клипа. Чтобы создать обработчик события кнопки, выполните следующие действия:

1. Начните новый фильм Flash.
2. Создайте кнопку и перетащите ее экземпляр на основной рабочий стол.
3. При выделенной кнопке 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, интерпретатор обнаруживает событие *release*, связанное с отпусканием кнопки, и выполняет обработчик события *on (release)*. Сообщение «You clicked the button» появляется в окне Output при каждом нажатии и отпускании кнопки.

Теперь попробуем создать несколько более интересный обработчик события для клипа. Выполните следующие действия:

1. Начните новый фильм Flash.
2. На главном рабочем столе фильма нарисуйте прямоугольник.
3. Выберите пункт меню Insert → Convert to Symbol
4. В диалоговом окне Symbol Properties присвойте новому символу имя `rectangle` и выберите Movie Clip в качестве Behavior.
5. Щелкните OK и завершите создание клипа `rectangle`.
6. Выделите на рабочем столе клип `rectangle` и введите в панели Actions код:

```
onClipEvent (keyDown) {
  _visible = 0;
}

onClipEvent (keyUp) {
  _visible = 1;
}
```

7. Выберите пункт меню Control → Test Movie.
8. Щелкните по фильму, чтобы активизировать ввод данных с клавиатуры, затем нажмите и удерживайте любую клавишу. При каждом нажатии клавиши клип `rectangle` исчезает. При каждом отпускании нажатой клавиши `rectangle` появляется снова.

Обратите внимание, что мы не выполняем каких-либо предложений, запускающих обработчик события: интерпретатор автоматически запускает обработчик события, когда происходит соответствующее событие.

Flash не позволяет прикреплять и удалять обработчики событий с помощью ActionScript во время воспроизведения фильма. Обработчики событий должны назначаться кнопкам и клипам с помощью среды разработки Flash. Поэтому следующий синтаксис является недопустимым:

```
myClip.onKeyDown = function () { _visible = 0; };
```

Позднее, в разделе «Динамические обработчики событий клипов», мы увидим, как можно обойти это ограничение.

## Прикрепление обработчиков событий к другим объектам

Кроме клипов и кнопок обработчики событий поддерживаются двумя встроенными классами объектов – *XML* и *XMLSocket*. Для этих объектов не нужно добавлять обработчики событий к каким-либо физическим сущностям в среде разработки Flash. Обработчики прикрепляются как методы к экземплярам объектов.

Для объектов *XML* и *XMLsocket* ActionScript использует предопределенные свойства, хранящие имена обработчиков событий. Например, свойство `onLoad` содержит имя обработчика, который должен выполняться при загрузке внешних данных XML.

Чтобы задать свойство `onLoad` для объекта *XML*, используется такой код:

```
myDoc = new XML();
myDoc.onLoad = function () { trace("загрузка завершена!"); };
```

В другом варианте можно сначала определить функцию обработчика, а затем присвоить ее свойству `onLoad` нашего объекта:

```
function doneMsg () {
  trace("загрузка завершена!");
}
myDoc.onLoad = doneMsg;
```

Этот синтаксис весьма похож на синтаксис JavaScript, где можно присваивать функции свойствам обработчика событий, как показано в примере 10.1.

*Пример 10.1 Присваивание обработчика событий в JavaScript*

```
// Присвоить литерал функции обработчику onload в JavaScript
window.onload = function () { alert("конец загрузки"); };

// или, иначе, создать, а затем присвоить функцию свойству onload
function doneMsg () {
  alert("конец загрузки");
}
window.onload = doneMsg;
```

Возможно, в будущем большее количество объектов ActionScript станет поддерживать присваивание обработчиков событий с помощью свойств объектов, поэтому разумно привыкать к такому стилю сразу. Если вы не пользуетесь объектами *XML* и *XMLSocket*, можете все равно попрактиковаться в создании таким способом обработчиков на документах HTML и JavaScript. Прелесть этого подхода в гибкости: любую функцию обработчика можно легко переназначить и даже удалить во время воспроизведения фильма.

Подробнее о прикреплении функций к объектам будет рассказано в главе 12 «Объекты и классы». Данные о событиях, поддерживаемых объектами *XML* и *XMLSocket*, можно найти в части III «Справочник по языку».

Срок жизни обработчиков событий связан с существованием объектов, с которыми они ассоциированы. Когда клип или кнопка удаляются с рабочего стола или прекращает существование объект *XML* или *XMLSocket*, заканчивается и существование всех обработчиков событий, связанных с этими объектами. Объект должен присутствовать на рабочем столе или существовать на временной диаграмме, чтобы его обработчики событий оставались действующими.

# Область видимости обработчиков событий

Как и любые функции, предложения в обработчике событий выполняются в предопределенной области видимости. Область видимости указывает интерпретатору, где разрешать переменные, подфункции или свойства, на которые есть ссылки в теле обработчика. Рассмотрим область видимости обработчика отдельно для событий клипов, событий кнопок и событий других объектов.

## Область видимости обработчика событий клипа

В отличие от обычных функций обработчики событий клипов *не определяют локальную область видимости*! Когда мы прикрепляем обработчик к клипу, областью видимости обработчика становится весь клип, а не сам обработчик. Это означает, что все переменные извлекаются из временной диаграммы клипа. Например, если мы прикрепляем обработчик события *enterFrame* к клипу с именем `navigation` и пишем внутри обработчика *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 в нашем обработчике *var y = 10;*, то переменная `y` будет определена на временной диаграмме `navigation`, несмотря на то, что при использовании в функции ключевое слово *var* обычно объявляет локальную переменную.

Проще всего запомнить правила области видимости обработчика событий, если рассматривать предложения в обработчике так, как если бы они были прикреплены к кадру клипа обработчика. Допустим, напри-

мер, что есть клип по имени ball с переменной xVelocity в нем. Для доступа к xVelocity внутри обработчика события 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 справедливо в отношении свойств и методов: вместо ball._x можно просто писать _x, а вместо *ball.gotoAndStop(5)* – просто *gotoAndStop(5)*. Например:

```
onClipEvent (enterFrame) {
  _x += xVelocity;                  // Переместить шарик
  gotoAndPlay(_currentframe - 1);  // Сделать короткий цикл
}
```

Можно даже определить функцию на клипе ball путем объявления функции в обработчике:

```
onClipEvent (load) {
  function hideMe() {
    _visibility = 0;
  }
}
```

Иногда легко забыть, что область видимости обработчиков событий клипа – *временная диаграмма клипа*, а не локальная область видимости функции обработчика и не *родительская* временная диаграмма клипа (та, 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 же явным образом сослаться на функцию *moveBall( )* в главной временной диаграмме через _root:

```
onClipEvent (enterFrame) {
  _root.moveBall();  // Теперь работает!
}
```

Иногда может потребоваться явная ссылка на объект текущего клипа внутри обработчика события. Сделать это можно с помощью ключево-

го слова this, которое, если используется в обработчике события, представляет собой ссылку на текущий клип. Поэтому следующие ссылки являются синонимами в обработчике события клипа:

```
this._x  // То же, что следующая строка
_x

this.gotoAndStop(12);  // То же, что следующая строка
gotoAndStop(12);
```

Использовать this чаще всего приходится, когда динамически генерируется имя одного из свойств текущего клипа (имя переменной или вложенного клипа). Вот как заставить выполняться один из вложенных клипов ball.stripe1, ball.stripe2,... в зависимости от текущего кадра клипа ball:

```
onClipEvent (enterFrame) {
  this["stripe" + _currentframe].play();
}
```

Ключевое слово this также часто используется 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 использоваться с явной ссылкой на клип. Поэтому ключевое слово this необходимо использовать при вызове следующих функций как методов внутри обработчика событий:

*duplicateMovieClip( )*
*loadMovie( )*
*loadVariables( )*
*print( )*
*printAsBitmap( )*
*removeMovieClip( )*
*startDrag( )*
*unloadMovie( )*

Например:

```
this.duplicateMovieClip("ball2", 1);
this.loadVariables("vars.txt");
this.startDrag(true);
this.unloadMovie();
```

О двойственной природе этих функций будет подробно рассказано в разделе «Проблемы перекрытия методов и глобальных функций» главы 13 «Клипы».

Обратите внимание, что ключевое слово this позволяет ссылаться на текущий клип, даже если этому клипу не присвоено имя экземпляра в среде разработки или имя клипа нам неизвестно. На самом деле, с помощью ключевого слова this можно передать текущий клип в виде

ссылки в функцию, даже не зная имени текущего клипа. Вот иллюстрирующий это вполне допустимый (и достаточно элегантный) код:

```
// 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);
}
```

В build 30 проигрывателя Flash 5 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 r30
}
```

Чтобы обойти эту ошибку, используйте ссылку на клип в таком виде:

```
onClipEvent(load) {
  this.gotoAndStop("intro");
}
```

## Область видимости обработчика событий кнопки

Обработчики событий кнопок имеют в качестве области видимости временную диаграмму, на которой располагается кнопка. 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
}
```

С другой стороны, если поместить клип `ball` на главную временную диаграмму и объявить переменную `speed` в обработчике для `ball`, областью видимости `speed` станет `ball`:

```
// КОД ДЛЯ ОБРАБОТЧИКА ball
on (load) {
  var speed = 10;  // Определяет speed в ball, а не _root
}
```

Внутри обработчика кнопки ключевое слово `this` ссылается на временную диаграмму, на которой расположена кнопка:

```
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;
}
```

## Область видимости обработчиков событий других объектов

В отличие от обработчиков клипов и кнопок обработчики событий, прикрепленные к экземплярам встроенных классов, например *XML* и *XMLSocket*, получают область видимости в точности так же, как функции. Для обработчиков событий *XML* и *XMLSocket* цепочка областей видимости определяется при определении функции обработчика. Более того, обработчики событий *XML* и *XMLSocket* определяют локальную область видимости. Все правила для областей видимости функций, описанные в разделе «Область видимости функций» главы 9 «Функции», непосредственно применимы к функциям обработчиков событий, прикрепленных к объектам, которые не являются кнопками или клипами.

# События кнопок

Данные табл. 10.1 кратко знакомят с различными событиями, доступными для кнопок. С помощью событий кнопок можно легко создать код для навигации, форм, игр и других элементов интерфейса. Исследуем каждое событие кнопки и посмотрим, как можно запрограммировать кнопку, чтобы она реагировала на события мыши и клавиатуры.

Каждое событие кнопки в табл. 10.1 обрабатывается соответствующим обработчиком события кнопки вида *on* (*eventName*). Например, событие *press* (нажатие кнопки) обрабатывается с помощью обработчика события, начинающегося с *on (press)*. Исключение составляет обработчик события *keyPress*, имеющий вид *on* (*keyPress key*), где *key* означает клавишу, нажатие которой должно быть обнаружено. События кнопок посылаются только той кнопке, с которой взаимодействует мышь. Если несколько кнопок перекрываются, все события посылаются самой верхней кнопке, даже если для нее не определены обработ-

чики, а другие кнопки реагировать не могут. В приведенных ниже описаниях активная область (*hit*-область) соответствует физической части кнопки, которая должна находиться под указателем мыши, чтобы кнопка была активизирована. (Активная область кнопки определяется графически во время создания кнопки в среде разработки Flash.)

*Таблица 10.1. События кнопок*

| Название события кнопки | Событие кнопки происходит, когда . . . |
|---|---|
| `press` | Первая кнопка мыши нажата во время нахождения указателя в *активной* области. Нажатие других кнопок мыши не обнаруживается |
| `release` | Первая кнопка мыши нажата, а затем отпущена во время нахождения указателя в *активной* области |
| `releaseOutside` | Первая кнопка мыши нажата во время нахождения указателя в *активной* области, а затем отпущена, когда указатель оказался вне *активной* области |
| `rollOver` | Указатель мыши входит в *активную* область кнопки при ненажатой кнопке мыши |
| `rollOut` | Указатель мыши выходит из *активной* область кнопки при ненажатой кнопке мыши |
| `dragOut` | Первая кнопка мыши нажата во время нахождения указателя в *активной* области, а затем при нажатой кнопке указатель переместился за пределы *активной* области |
| `dragOver` | Первая кнопка мыши нажата во время нахождения указателя в *активной* области, а затем при нажатой кнопке указатель переместился за пределы *активной* области и обратно в нее |
| `keyPress` | Указанная клавиша *key* находится в нажатом состоянии. В большинстве случаев предпочтительнее использовать не это событие, а событие клипа *keyDown* |

## press

Щелчок мышью состоит из двух действий: сначала кнопка нажимается (*press*), а затем отпускается (*release*). Событие *press* происходит, когда указатель мыши находится в активной области кнопки и нажимается первая кнопка мыши. Нажатия остальных кнопок мыши не определяются. События *press* можно использовать с переключателями (radio buttons) или для открытия огня в играх, но для того чтобы разрешить пользователю изменить свое решение до освобождения кнопки, используйте события *release*.

## release

Событие кнопки *release* происходит при обнаружении следующей последовательности:

1. Указатель мыши находится в *активной* области кнопки.
2. Нажата первая кнопка мыши, в то время как указатель мыши продолжает находиться в *активной* области (в этот момент происходит событие *press*).
3. Первая кнопка мыши отпущена, в то время как указатель мыши продолжает находиться в *активной* области исходной кнопки (в этот момент происходит событие *release*).

Используя событие *release* вместо события *press*, вы даете пользователям возможность увести указатель с кнопки даже после щелчка по ней и отменить таким способом свое действие.

## releaseOutside

Событие *releaseOutside* обычно указывает на то, что пользователь изменил свое решение: сначала щелкнул по кнопке, а затем отвел указатель в сторону, прежде чем отпустить кнопку мыши. Это событие генерируется, когда обнаружена такая последовательность:

1. Указатель мыши находится в *активной* области кнопки.
2. Нажата и удерживается первая кнопка мыши (происходит событие *press*).
3. Указатель мыши выходит из *активной* области кнопки (происходит событие *dragOut*).
4. Первая кнопка мыши отпущена, в то время как указатель мыши находится вне *активной* области исходной кнопки.

Вам редко придется обнаруживать события *releaseOutside*, так как обычно они указывают на отказ пользователя совершить какие-либо действия.

## rollOver

Событие *rollOver* происходит, когда указатель мыши входит в *активную* область кнопки, и при этом не нажаты кнопки мыши. Событие *rollOver* редко используется в ActionScript, потому что изменение внешнего вида кнопки задается прямо в среде разработки, а не с помощью сценария. Чтобы создать состояния выделения для кнопок, следует использовать кадры *up*, *over* и *down*, предоставляемые редактором Flash.

Во Flash 5 событие *rollOver* предоставляет удобный способ восстановить выделение в текстовом поле. Подробности можно найти в разделе «Объект Selection» части III.

## rollOut

Событие *rollOut* противоположно событию *rollOver*: оно происходит, когда указатель мыши выходит из активной область кнопки при ненажатых кнопках мыши. Как и *rollOver*, событие *rollOut* используется редко, потому что состояние выделения для кнопок создается прямо в среде разработки, и переключать изображения вручную в ActionScript не требуется.

## dragOut

Событие *dragOut* похоже на *rollOut*, но, в отличие от него, генерируется, если при выходе из активной области кнопка мыши находится в нажатом состоянии. За событием *dragOut* следуют либо событие *releaseOutside* (если пользователь отпускает кнопку мыши), либо событие *dragOver* (если пользователь возвращает указатель в активную область кнопки, не отпустив кнопку мыши).

## dragOver

Событие *dragOver* – это редко встречающийся лесной зверь:

1. Указатель мыши перемещается в *активную* область кнопки (происходит событие *rollOver*).
2. Первая кнопка мыши нажата и удерживается (происходит событие *press*).
3. Указатель мыши покидает *активную* область кнопки (происходит событие *dragOut*).
4. Указатель мыши возвращается в *активную* область кнопки (происходит событие *dragOver*).

Таким образом, событие *dragOver* указывает на то, что пользователь вывел курсор мыши из активной области и снова вернул его в активную область, при этом все время удерживая кнопку мыши нажатой. Обратите внимание, что если при повторном входе указателя в *активную* область кнопки кнопка мыши остается нажатой, генерируется событие *dragover*, а не *rollOver*.

## keyPress

Событие *keyPress* не связано с мышью и запускается нажатием указанной клавиши. Мы рассказываем о нем в этом разделе, потому что это событие использует такой же синтаксис *on* (*eventName*), как другие обработчики событий кнопок ActionScript. Этот обработчик события требует указать клавишу, которая вызывает данное событие:

```
on (keyPress key) {
  statements
}
```

где *key* является строкой, представляющей клавишу, связываемую с событием. Строка может быть символом на клавише (например, «s» или «S») либо ключевым словом, представляющим клавишу в формате "*<Keyword>*". Для каждого обработчика можно задать только одну клавишу. Чтобы перехватывать с помощью *keyPress* несколько клавиш, нужно создать несколько обработчиков события *keyPress*. Например:

```
// Обнаруживает нажатие клавиши "a"
on (keyPress "a") {
  trace("The 'a' key was pressed");
}

// Обнаруживает нажатие клавиши Enter
on (keyPress "<Enter>") {
  trace("The Enter key was pressed");
}

// Обнаруживает нажатие клавиши "стрелка вниз"
on (keyPress "<Down>") {
  trace("The Down Arrow key was pressed");
}
```

Допустимыми значениями *Keyword* являются следующие (обратите внимание, что клавиши F1 . . . F12 не поддерживаются событием *keyPress*, но обнаруживаются с помощью объекта *Key*):

```
<Backspace>
<Delete>
<Down>
<End>
<Enter>
<Home>
<Insert>
<Left>
<PgDn>
<PgUp>
<Right>
<Space>
<Tab>
<Up>
```

Во Flash 4 событие *keyPress* было единственным способом взаимодействия с клавиатурой. Во Flash 5 появился объект *Key*, который в сочетании с событиями клипов *keyDown* и *keyUp* (о которых рассказывается далее) предоставляет значительно большие возможности взаимодействия с клавиатурой. Событие *keyPress* в каждый момент обнаруживает нажатие только одной клавиши, в то время как объект *Key* может обнаруживать одновременное нажатие нескольких клавиш.

# Обзор событий клипов

События клипов генерируются многими явлениями во Flash Player – от щелчков мышью до загрузки данных. События клипов можно разбить на две категории: события, вызываемые пользователем, и события при воспроизведении фильма. События, вызываемые пользователем, связаны с мышью и клавиатурой, в то время как события при воспроизведении фильма связаны с выводом кадров во Flash Player, рождением и смертью клипов и загрузкой данных.

Обратите внимание, что события клипов, вызываемые пользователем, частично пересекаются с функциями, выполняемыми описанными выше событиями кнопок. Например, обработчик события клипа *mouseDown* может обнаружить нажатие кнопки мыши так же, как это может сделать обработчик события кнопки *press*. Однако события клипов не привязаны к каким-либо активным областям, как в случае событий кнопок, и не влияют на вид указателя мыши.

Разберем события клипов ActionScript, сведенные в табл. 10.2. Сначала мы рассмотрим события, связанные с воспроизведением фильмов (*enterFrame*, *load*, *unload* и *data*), а затем посмотрим, как действуют события, связанные с пользователем (*mouseDown*, *mouseUp*, *mouseMove*, *keyDown*, *keyUp*). Каждое событие клипа обрабатывается соответствующим обработчиком события клипа, имеющим вид *onClipEvent* (`eventName`). Например, обработчик события *enterFrame* начинается со слов *onClipEvent (enterFrame)*. За исключением событий *load*, *unload* и *data*, события клипов посылаются во *все* клипы, имеющиеся на рабочем столе, даже если, к примеру, пользователь щелкает мышью по другому клипу (или вообще не по клипу).

*Таблица 10.2. События клипов*

| Название события клипа | Событие клипа происходит, когда . . . |
|---|---|
| *enterFrame* | Воспроизводящая головка входит в кадр (перед выводом кадра во Flash Player) |
| *load* | Клип впервые появляется на рабочем столе |
| *unload* | Клип удаляется с рабочего стола |
| *data* | Закончена загрузка в клип переменных, или часть загруженного фильма загружена в клип |
| *mouseDown* | Первая кнопка мыши нажата во время нахождения клипа на рабочем столе (нажатие других кнопок мыши не обнаруживается) |
| *mouseUp* | Первая кнопка мыши отпущена во время нахождения клипа на рабочем столе |

*Таблица 10.2 (продолжение)*

| Название события клипа | Событие клипа происходит, когда . . . |
|---|---|
| *mouseMove* | Курсор мыши перемещается (хоть самую малость) во время нахождения клипа на рабочем столе, даже если указатель находится не над клипом |
| *keyDown* | Нажата клавиша во время нахождения клипа на рабочем столе |
| *keyUp* | Отпущена клавиша во время нахождения клипа на рабочем столе |

# События клипов, связанные с воспроизведением фильмов

Следующие события генерируются без вмешательства пользователя, когда Flash загружает и воспроизводит фильмы.

## enterFrame

Если вам когда-либо приходилось использовать пустые циклические клипы для запуска сценариев, знайте, что *enterFrame* предлагает желанное избавление от этого. Событие *enterFrame* возникает по одному разу для каждого кадра, который проходит в фильме. Например, следующий код, помещенный в клип, вызывает нарастающее увеличение клипа на 10 пикселов за кадр:

```
onClipEvent (enterFrame) {
  _height += 10;
  _width += 10;
}
```

(Обратите внимание на то, что, как уже указывалось, свойства `_height` и `width` разрешаются в области видимости клипа, к которому прикреплен обработчик события *enterFrame*, поэтому перед `_height` и `_width` не требуется указывать имя экземпляра клипа.)

Событие *enterFrame* генерируется перед выводом каждого кадра, даже если воспроизводящая головка клипа с обработчиком *enterFrame* остановлена. Поэтому событие *enterFrame* генерируется всегда.

При выводе во Flash Player постоянно запущены все фильмы Flash, даже если ничто не движется по экрану, и воспроизводящая головка остановлена в кадре. Поэтому обработчик *enterFrame* отдельного клипа будет выполняться многократно, пока этот клип находится на рабочем

столе, независимо от того, воспроизводится этот клип или остановлен. Если воспроизводящая головка клипа перемещается в результате вызова функции *gotoAndStop( )*, обработчик события клипа *enterFrame* все равно запускается для каждого проходящего кадра. И если все воспроизводящие головки во всем фильме остановлены с помощью функции *stop( )*, все обработчики событий *enterFrame* во всех клипах все равно будут выполняться.

Событие *enterFrame* обычно используется для того, чтобы с течением времени повторно обновлять состояние клипа. Но обработчик события *enterFrame* не обязательно применять непосредственно к содержащему его клипу: *enterFrame* можно применять к пустому клипу, состоящему из одного кадра, для многократного выполнения кода. Этот прием, называемый *циклом события клипа* (*clip event loop*), или, менее формально, *процессом*, продемонстрирован в разделе «Циклы временной диаграммы и событий клипа» главы 8 «Предложения цикла».

Обратите внимание на то, что код в обработчике события *enterFrame* выполняется *раньше* любого кода, имеющегося на временной диаграмме клипа, содержащего обработчик.

При желании можно с помощью *enterFrame* получить чрезвычайно мощные возможности управления клипом. Пример 10.7, приведенный ниже, показывает, как можно расширить наш прежний код, увеличивающий клип, чтобы заставить размеры клипа колебаться в обе стороны.

## load

Событие *load* происходит при рождении клипа, т. е. при первом появлении клипа на рабочем столе. Клип «появляется на сцене» одним из следующих способов:

- Воспроизводящая головка переходит в ключевой кадр, содержащий новый экземпляр клипа, помещенный в среду разработки Flash.
- Клип создается как дубликат другого клипа с помощью функции *duplicateMovieClip( )*.
- Клип помещается на рабочий стол программируемым образом с помощью функции *attachMovie( )*.
- Внешний файл *.swf* загружается в клип с помощью функции *loadMovie( )*.
- Содержимое клипа выгружается с помощью функции *unloadMovie( )*. (Событие *load* генерируется потому, что после удаления из клипа содержимого в него загружается пустой клип-заместитель.)

Тело обработчика событий *load* выполняется *после* любого кода на временной диаграмме, где впервые появляется клип.

Обработчик события *load* часто используется для инициализации переменных клипа или выполнения некоторой задачи настройки (на-

пример, определения размера или положения динамически генерируемого клипа). Обработчик *load* может также служить хорошим способом предотвратить автоматическое воспроизведение клипа:

```
onClipEvent (load) {
  stop();
}
```

Обработчик события *load* может использоваться и для запуска некоторой функции, для правильного выполнения которой требуется существование определенного клипа.

Особенно интересно использовать событие *load* в сочетании с функцией *duplicateMovieClip( )*, создающей новые клипы. В примере 10.2 создается целое созвездие клипов `star` с помощью одного обработчика событий *load*, вызываемого каскадно. Обработчик события загрузки копируется в каждый дублируемый `star`, заставляя его, в свою очередь, дублировать себя. Процесс прекращается после копирования 100-го клипа. Файл *.fla* для примера 10.2 можно получить из сетевого Хранилища кода.

*Пример 10.2. Создание созвездия клипов star с помощью события load*

```
onClipEvent (load) {
  // Поместить текущий клип в случайно выбранную точку
  _x = Math.floor(Math.random() * 550);
  _y = Math.floor(Math.random() * 400);

  // Установить масштаб, чтобы не наследовать его от предыдущего клипа
  _xscale = 100;
  _yscale = 100;

  // Установить случайный масштаб текущего клипа между 50% и 150%
  randScale = Math.floor(Math.random() * 100) - 50;
  _xscale += randScale;
  _yscale += randScale;

  // Если это еще не 100-я звездочка, создать новую
  if (_name != "star100") {
    nextStarNumber = number(_name.substring(4, _name.length)) + 1;
    this.duplicateMovieClip("star" + nextStarNumber, nextStarNumber);
  }
}
```

## unload

Событие *unload* противоположно событию *load*: оно происходит при гибели клипа, т. е. сразу *после* последнего кадра, в котором присутствует клип на рабочем столе (но *перед* первым кадром, в котором клип отсутствует).

Вызвать событие клипа *unload* могут следующие обстоятельства:

- Воспроизводящая головка достигает конца диапазона кадров, в которых размещен клип.
- Клип удален с помощью функции *removeMovieClip( )* (которая уничтожает клипы, созданные посредством использования функций *attachMovie( )* и *duplicateMovieClip( )*).
- Ранее загруженный внешний файл *.swf* удален из клипа с помощью функции *unloadMovie( )*.
- В клип загружен внешний файл *.swf*.

Этот последний способ создания события *unload* может показаться несколько странным, но на самом деле является естественным следствием процедуры загрузки фильмов во Flash. Всякий раз, когда файл *.swf* загружается в клип, прежнее содержание этого клипа замещается, что вызывает событие *unload*. Вот пример, иллюстрирующий поведение событий *load* и *unload* при выполнении функции *loadMovie( )*:

1. Находясь в среде разработки Flash, поместим на рабочий стол пустой клип в кадре 1 главной временной диаграммы. Назовем этот клип `emptyClip`.
2. В кадре 5 главной временной диаграммы загрузим в `emptyClip` фильм *test.swf* с помощью такого кода: `emptyClip.loadMovie("test.swf");`.
3. Воспроизведем фильм, выбрав пункт меню Control → Play movie.

В результате получится следующее:

1. Кадр 1: Появляется клип `emptyClip`, вызывая событие *load*.
2. Кадр 5: Выполняется функция *loadMovie( )* в два этапа:
   a) Содержимое-«заполнитель» (placeholder) `emptyClip` удаляется, чтобы освободить место для поступающего *test.swf*, что вызывает событие *unload*.
   b) Загружается фильм *test.swf*, что вызывает событие *load*.

Событие *unload* обычно используется для запуска кода уборки, т. е. кода очистки рабочего стола или восстановления программной среды каким-либо образом. Обработчик *unload* дает также возможность выполнить некоторое действие по окончании клипа (например, запустить показ другого фильма).

## data

Событие *data* происходит при загрузке в клип внешних данных. Событие *data* может генерироваться в двух совершенно разных случаях в зависимости от типа загружаемых данных. Рассмотрим эти два случая по отдельности.

## Использование обработчика события data с функцией loadVariables()

Если у сервера запрашивается ряд переменных с помощью функции *loadVariables( )*, необходимо дождаться их полной загрузки, прежде чем использовать содержащиеся в них данные (см. часть III).

Когда клип получает всю группу загружаемых переменных, генерируется событие *data*, которое указывает, что можно выполнять код, зависящий от этих переменных.

Допустим, например, что имеется фильм гостевой книги, в которую пользователи вводят свои комментарии, а эти комментарии сохраняются на сервере. Если пользователь хочет просмотреть комментарий, тот запрашивается с сервера с помощью *loadVariables( )*. Но прежде чем начать вывод комментария, необходимо сделать остановку на экране загрузки, пока не станет известно, что запрошенные данные доступны. Обработчик события *data* извещает о том, что данные загружены, после чего можно вывести комментарий пользователю.

В примере 10.3 приведен упрощенный отрывок кода гостевой книги, показывающий использование обработчика события *data* с функцией *loadVariables( )*. В этом примере есть кнопка, с помощью которой две переменные в кодировке URL загружаются из текстового файла в клип. В клипе имеется обработчик события *data*, который выполняется, когда эти две переменные загружены. Обработчик отображает значения этих переменных. Безопасность вывода этих переменных гарантируется тем, что код обработчика выполняется только при генерации события *data* (т. е. после получения данных).

*Пример 10.3. Ожидание события data*

```
// СОДЕРЖИМОЕ ФАЙЛА GUESTBOOK.TXT
name=judith&message=hello

// КНОПКА ВНУТРИ КЛИПА
on (release) {
  this.loadVariables("guestbook.txt");
}

// ОБРАБОТЧИК В КЛИПЕ
onClipEvent (data) {
  trace(name);
  trace(message);
}
```

Мы еще воспользуемся событием *data* при создании формы Flash в главе 17 «Формы Flash».

## Использование обработчика события data с функцией loadMovie()

Второе применение события *data* связано с загрузкой в клипы внешних файлов *.swf* с помощью функции *loadMovie( )*. Когда файл *.swf* загружается в клип, который будет его содержать, то по умолчанию сразу начинается воспроизведение файла, даже если он загружен только частично. Это не всегда желательно: иногда требуется обеспечить загрузку всего файла *.swf* или определенной его части до начала воспроизведения. Гарантировать это можно с помощью обработчика события *data* и некоторого кода предварительной загрузки.

Событие *data* происходит всякий раз, когда принимающий данные клип получает часть внешнего файла *.swf*. Определение того, что составляет «часть», более сложно, чем можно предположить. Для генерации события *data* должен быть загружен хотя бы один новый полный кадр внешнего файла *.swf* , после того как: a) было сгенерировано последнее событие *data* или b) началась загрузка файла *.swf*. (За этот промежуток времени фактически могло быть загружено более одного кадра файла *.swf*, но один кадр является минимальным количеством, необходимым для вызова события *data*.)

Выполнение обработчиков событий *data* связано с выводом кадров в проигрывателе Flash. При выводе каждого кадра интерпретатор проверяет, загружена ли часть внешнего файла *.swf* в клип, содержащий обработчик события *data*. Если в такой клип загружена часть внешнего файла *.swf* и в загруженной порции есть хоть один новый кадр, то запускается обработчик события *data*. Это действие происходит один и только один раз для каждого выводимого кадра (даже если воспроизводящая головка остановлена).

Обратите внимание, что поскольку событие *data* происходит покадрово, предварительные загрузчики фильмов с более высокой скоростью воспроизведения выглядят более сглаженными благодаря тому, что для них чаще обновляются состояния загрузки файлов *.swf*.

Точное количество событий *data*, возникающих во время операции *loadMovie( )*, зависит от распределения содержимого в загружаемом файле *.swf* и скорости соединения. Файл *.swf* с одним кадром запустит только одно событие *data*, как бы велик он ни был. С другой стороны, файл *.swf* с 100 кадрами может запустить до 100 отдельных событий *data* в зависимости от частоты воспроизведения кадров в фильме, количества байтов в каждом кадре и скорости сетевого соединения. Если кадры большие, а соединение медленное, будет запущено больше событий *data*, вплоть до максимально возможного: по одному на каждый кадр. Если кадры маленькие, а соединение быстрое, будет запущено меньше событий *data* (все 100 кадров могут быть переданы в промежутке между выводом проигрывателем двух кадров, породив только одно событие *data*).

Как же применять обработчик события *data* для создания предварительного загрузчика? Всякий раз, когда вследствие вызова функции *loadMovie( )* происходит событие *data*, оно свидетельствует о выполнении загрузки внешнего файла *.swf*. Поэтому, находясь в обработчике события *data*, можно контролировать, достаточно ли велика загруженная часть файла, чтобы разрешить его воспроизведение. Это делается с помощью функций *getBytesLoaded( )* и *getBytesTotal( )*, дающих количество загруженных байтов и общее количество байтов, как показано в примере 10.4. (Можно также воспользоваться свойствами клипа `_framesloaded` и `_totalframes`.)

В примере 10.4 обеспечена также обратная связь во время загрузки фильма. Обратите внимание, что в первом кадре загружаемого файла *.swf* должен быть вызов функции *stop( )*, чтобы не допустить автоматического воспроизведения фильма до полной его загрузки. Вариант примера 10.4 можно загрузить из сетевого Хранилища кода.

*Пример 10.4. Предварительный загрузчик в событии data*

```
onClipEvent (data) {
    trace("data received");          // Вот-вот начнется представление!

    // Включить сигнал передачи данных
    _root.transferIndicator.gotoAndStop("on");

    // Если загрузка завершена, выключить сигнал передачи данных
    // и позволить фильму воспроизводиться
    if (getBytesTotal() > 0 && getBytesLoaded() == getBytesTotal()) {
      _root.transferIndicator.gotoAndStop("off");
      play();
    }

    // Вывести некоторые сведения о передаче данных в переменных
    // текстовых полей на _root
    _root.bytesLoaded = getBytesLoaded();
    _root.bytesTotal = getBytesTotal();
    _root.clipURL = _url.substring(_url.lastIndexOf("/") + 1, _url.length);
}
```

# События клипа, связанные с пользователем

Остальные события клипа относятся к взаимодействию с пользователем. Когда происходит какое-нибудь событие клипа, связанное с вводом пользователя, *все* клипы на рабочем столе (вне зависимости от глубины вложенности в другие клипы) получают это событие. В результате несколько клипов могут реагировать на один щелчок мыши, перемещение мыши или нажатие клавиши.

Чтобы выполнить некоторый код при приближении мыши к определенному клипу, обработчик события должен получить расположение

указателя мыши относительно клипа. Встроенная функция *hitTest( )* предоставляет простой способ проверить, не произошел ли щелчок мыши в определенной области, как показано ниже в примере 10.9.

## mouseDown

Подобно событию кнопки *press*, событие клипа *mouseDown* обнаруживает нажатие кнопки мыши. Событие *mouseDown* происходит всякий раз, когда первая кнопка мыши нажата во время нахождения указателя мыши над *любым участком* рабочего стола.

В отличие от события кнопки *press*, событие *mouseDown* не связано с *активной* областью кнопки. В сочетании с событиями *mouseUp* и *mouseMove*, а также методом *Mouse.hide( )*, событие *mouseDown* можно использовать для создания пользовательского указателя мыши, как будет показано ниже в примере 10.8.

## mouseUp

Событие *mouseUp* является парным к событию *mouseDown*. Оно происходит всякий раз, когда первая кнопка мыши отпускается во время нахождения указателя мыши над любым участком рабочего стола. Как и для *mouseDown*, клип с обработчиком *mouseUp* должен присутствовать на рабочем столе в момент отпускания кнопки мыши, чтобы событие привело к каким-либо результатам. События *mouseUp*, *mouseDown* и *mouseMove* можно использовать для создания богатых возможностей интерактивности, не влияя на курсор мыши (как это делает кнопка).

## mouseMove

Событие *mouseMove* позволяет обнаружить изменение местоположения курсора мыши. Во время перемещения мыши события *mouseMove* происходят многократно с той скоростью, с которой процессор может генерировать события. Чтобы событие *mouseMove* возымело какой-либо эффект, клип с обработчиком *mouseMove* должен присутствовать на рабочем столе во время перемещения мыши.

Событие *mouseMove* полезно связывать с кодом, который активизирует ожидающие приложения, отображает след мыши и создает пользовательские указатели, как будет показано ниже в примере 10.8.

## keyDown

События *keyDown* и *keyUp* являются клавиатурными аналогами событий *mouseDown* и *mouseUp*. Вместе они образуют основные средства кодирования интерактивных действий посредством клавиатуры. Событие *keyDown* происходит при нажатии клавиши на клавиатуре. Ес-

ли клавиша удерживается, событие *keyDown* может происходить многократно в зависимости от операционной системы и настроек клавиатуры. В отличие от события *keyPress*, события клипа *keyDown* происходят при нажатии любой, а не какой-то конкретной, клавиши.

Чтобы *перехватить* (*trap*, *catch*) или обнаружить событие *keyDown*, клип с обработчиком *keyDown* должен присутствовать на рабочем столе в момент нажатия клавиши. Следующий код выполняет такой перехват:

```
onClipEvent (keyDown) {
  trace("Some key was pressed");
}
```

Как можно заметить, этот обработчик *keyDown* не сообщает о том, какая клавиша была нажата. Но обычно требуется связать некоторое действие с конкретной клавишей, например, если нужно, чтобы различные клавиши поворачивали космический корабль в разных направлениях.

Чтобы узнать, какими клавишами вызвано событие *keyDown*, нужно обратиться к встроенному объекту *Key*, который описывает состояние клавиатуры. Тип требуемой информации зависит от того, какого рода интерактивность должна быть реализована. Например, в играх требуется мгновенная и непрерывная реакция на нажатия клавиш, которые могут быть одновременными. Напротив, в интерфейсах для навигации может требоваться только обнаружение нажатия одной клавиши (например, пробела в слайдовой презентации).

Объект *Key* может сообщить, какая клавиша была нажата последней, и нажата ли в данный момент какая-то определенная клавиша. Для определения состояния клавиатуры используется один из четырех методов объекта *Key*:

```
Key.getCode()          // Код последней нажатой клавиши
                       // в виде числа десятичной системы счисления
Key.getAscii()         // ASCII-код последней нажатой клавиши
                       // в виде числа десятичной системы счисления
Key.isDown(keycode)    // Возвращает true, если указанная клавиша
                       // нажата в данный момент
Key.isToggled(keycode) // Определяет состояние переключателя Caps Lock/Num Lock
```

В примере 10.5 приведен обработчик события *keyDown*, который сообщает ASCII-код последней нажатой клавиши.

*Пример 10.5. Проверка последней нажатой клавиши*

```
onClipEvent (keyDown) {
  // Получить ASCII-код последней нажатой клавиши и преобразовать в символ
  lastKeyPressed = String.fromCharCode(Key.getAscii());
  trace("Вы нажали клавишу '" + lastKeyPressed."");
}
```

В примере 10.6 приведен обработчик *keyDown*, который проверяет, не была ли «стрелка вверх» последней нажатой клавишей.

*Пример 10.6. Обнаружение нажатия клавиши «стрелка вверх»*

```
onClipEvent (keyDown) {
  // Проверить, была ли последней нажатой клавишей "стрелка вверх".
  // "Стрелка вверх" представляется свойством Key.UP.
  if (Key.getCode() == Key.UP) {
    trace("Последней нажатой клавишей была "стрелка вверх".");
  }
}
```

Есть несколько способов проверки состояния клавиатуры, и нужно избрать тот, что лучше всего подходит для конкретного приложения. Например, метод *Key.getAscii( )* возвращает ASCII-код символа, связанного с последней нажатой клавишей, который может различаться для клавиатур, использующих разные языки (хотя для английского языка расположение на клавиатуре букв и цифр стандартизовано). Напротив, метод *Key.getCode( )* возвращает значение, связанное с физической клавишей клавиатуры, а не конкретную букву. *Key. getCode( )* может оказаться полезнее для интернациональной или представленной разными платформами аудитории, если желательно, скажем, воспользоваться для навигации четырьмя соседними клавишами независимо от символов, которые они представляют. Дополнительная информация на эту тему есть в разделе «Объект Key» в части III.

Файлы *.fla* с примерами обработки событий *keyDown* и *keyUp* можно найти в сетевом Хранилище кода.

Обработчики событий, реагирующие на нажатие клавиш, выполняются только тогда, когда в фокусе курсора мыши находится проигрыватель Flash. Чтобы активизировать обработчики нажатий клавиш фильма, пользователь должен щелкнуть мышью по рабочему столу фильма. Прежде чем войти в раздел фильма, управляемый с клавиатуры, попробуйте заставить пользователя щелкнуть по кнопке.

## Обработка специальных клавиш

Чтобы отключить команды меню автономного проигрывателя Flash (Open, Close, Fullscreen и т. д.), добавьте в начало своего фильма следующую строчку кода:

```
fscommand("trapallkeys", "true");
```

Эта команда также лишает возможности выхода из полноэкранного режима Projector с помощью клавиши <Escape>. Для перехвата клавиши <Escape> в Projector воспользуйтесь таким обработчиком:

```
onClipEvent (keyDown) {
```

```
  if (Key.getCode() == Key.ESCAPE) {
    // Реакция на нажатие Escape
  }
}
```

Имейте в виду, что клавишу <Escape> можно перехватить не во всех броузерах. Кроме того, нельзя дезактивировать клавишу <Alt> или стандартные для Windows комбинации клавиш <Alt>+<Tab> и <Ctrl>+<Alt>+<Delete>.

Чтобы перехватить нажатие клавиши <Tab>, создайте кнопку с таким обработчиком:

```
on (keyPress "<Tab>") {
  // Реакция на нажатие Tab
}
```

В автономном проигрывателе Flash клавишу <Tab> можно также перехватить с помощью обработчика события клипа вида:

```
onClipEvent (keyDown) {
  if (Key.getCode() == Key.TAB) {
    // Реакция на нажатие Tab
  }
}
```

В некоторых броузерах клавишу <Tab> можно обнаружить только с помощью события кнопки *keyPress*, и может даже потребоваться совместно использовать событие кнопки *keyPress* и событие клипа *keyUp*. В следующем коде сначала перехватывается клавиша <Tab> с помощью *keyPress*, а затем следует реакция на нее в обработчике *keyUp*. Обратите внимание, что мы не используем *keyDown*, потому что *Key.getCode( )* для клавиши <Tab> в Internet Explorer задается только при ее освобождении:

```
// КОД В КНОПКЕ НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
on (keyPress "<Tab>") {
  // Задать фиктивную переменную
  foo = 0;
}

// КОД В КЛИПЕ НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
onClipEvent (keyUp) {
  if (Key.getCode() == Key.TAB) {
    // Поместить курсор в myTextField на _level0
    Selection.setFocus("_level0.myTextField");
  }
}
```

Обычно клавиша <Tab> перехватывается, чтобы переместить точку ввода в определенное текстовое поле формы. Подробности в примере раздела «Объект Key» части III.

Чтобы перехватить сочетание клавиш в стиле сокращенного набора, такое как <Ctrl>+<F>, используйте обработчик *enterFrame* и метод *Key.isDown( )*:

```
onClipEvent (enterFrame) {
  if (Key.isDown(Key.CONTROL) && Key.isDown(70)) {
  // Реакция на комбинацию Ctrl-F
  }
}
```

Чтобы перехватить нажатие клавиши <Enter> (или <Return>), используйте обработчик кнопки вида:

```
on (keyPress "<Enter>") {
  // Реакция на нажатие клавиши Enter (например, подача формы)
}
```

или обработчик *keyDown*, например:

```
onClipEvent (keyDown) {
  if (Key.getCode() == Key.ENTER) {
    // Реакция на нажатие клавиши Enter (например, подача формы)
  }
}
```

Дополнительные сведения о перехвате других специальных клавиш, например, функциональных клавиш (F1, F2 и т. д.) или клавиш на цифровом поле, читайте в разделах «Объект Key» и «Метод Key.getCode()» части III.

## keyUp

Событие *keyUp* генерируется при освобождении нажатой клавиши. Это событие является важным при программировании игр, т. к. позволяет выключать то, что ранее было включено в результате события *keyDown* – классическим примером служит атака, выполняемая космическим кораблем. Еще одним примером является среда разработки Flash, в которой нажатие клавиши пробела переключает на инструмент Hand, а освобождение пробела восстанавливает предыдущий инструмент. Такой способ позволяет показывать и скрывать какие-то элементы приложения, например, временные меню.

Как и в событии *keyDown*, для получения полезных сведений о событии *keyUp* обычно используется объект *Key*:

```
onClipEvent (keyUp) {
  if (!Key.isDown(Key.LEFT)) {
    trace("'Стрелка влево' не нажата");
  }
}
```

Так как метод *Key.isDown( )* позволяет в любой момент проверить состояние любой клавиши, можно воспользоваться циклом события *enterFrame* для контроля нажатия определенной клавиши. Проводить же *опрос* клавиатуры (т. е. повторяющуюся проверку состояния клавиши) менее эффективно, чем ожидать, пока не станет точно известно, что какая-то клавиша нажата, о чем сообщает событие *keyDown*, запускающее обработчик.

Выбираемый подход в конечном счете зависит от типа создаваемой системы. В системе, постоянно находящейся в движении, например в игре, может подойти опрос, поскольку в любом случае в каждом кадре выполняется главный цикл игры. Поэтому во время прохождения этого цикла можно проверить и объект *Key*. Например:

```
// КОД В ПУСТОМ КЛИПЕ
// Здесь поддерживается продолжение процесса игры
onClipEvent (enterFrame) {
  _root.mainLoop();
}

// БАЗОВЫЙ КОД ИГРЫ НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
// Этот код выполняется один раз для каждого кадра
function mainLoop () {
  if (Key.isDown(Key.LEFT)) {
    trace("Нажата клавиша 'стрелка влево'");
    // Повернуть космический корабль влево
  }

  // Проверить остальные клавиши, затем продолжить цикл игры
}
```

В статичном интерфейсе нет необходимости проверять нажатие клавиш в цикле *enterFrame*, если только не требуется обнаружить особые сочетания клавиш (т. е. клавиши, нажатые одновременно). Обычно следует использовать обработчики событий *keyDown* и *keyUp*, которые генерируются ровно один раз для каждого нажатия и освобождения клавиши. При использовании обработчиков событий *keyUp* и *keyDown* не нужно заботиться о том, нажата ли все еще клавиша в каждый данный момент. Благодаря этому нажатия клавиш обнаруживаются точно, даже если пользователь освобождает клавишу между кадрами, а также предотвращает повторную проверку одной и той же клавиши, когда она была нажата лишь один раз. В любом случае для проверки последней нажатой клавиши в обработчиках событий *keyDown* и *keyUp* обычно применяются методы *Key.getCode( )* и *Key.getASCII( )*.

# Порядок выполнения кода

В некоторых фильмах код рассыпан по нескольким временным диаграммам и обработчикам событий клипов. Поэтому нередко один кадр

требует выполнить несколько блоков кода: одни в обработчиках событий, другие в кадрах на временных диаграммах, а некоторые на главных временных диаграммах документов в проигрывателе. 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 в соответствии с замыслом разработчика, нужно познакомиться с тем, в каком порядке выполняются обработчики событий в отношении различных временных диаграмм фильма.

Асинхронные обработчики событий выполняются независимо от кода на временных диаграммах фильма. Например, обработчики событий кнопок выполняются немедленно, когда происходит событие, которое они обрабатывают; это же относится к обработчикам событий *mouseDown*, *mouseUp*, *mouseMove*, *keyDown* и *keyUp*.

Однако обработчики событий, возникающих при воспроизведении фильма, выполняются в очередности, соответствующей ходу фильма, как показано в табл. 10.3.

*Таблица 10.3. Очередность выполнения обработчиков событий фильма*

| **Обработчик события** | **Момент выполнения** |
|---|---|
| *load* | Выполняется в первом кадре, где клип присутствует на рабочем столе, после выполнения кода родительской временной диаграммы, но до выполнения внутреннего кода клипа и вывода кадра. |
| *unload* | Выполняется в первом кадре, где клип отсутствует на рабочем столе, до выполнения кода родительской временной диаграммы. |
| *enterFrame* | Выполняется во втором и всех последующих кадрах, где клип присутствует на рабочем столе. 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 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 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.

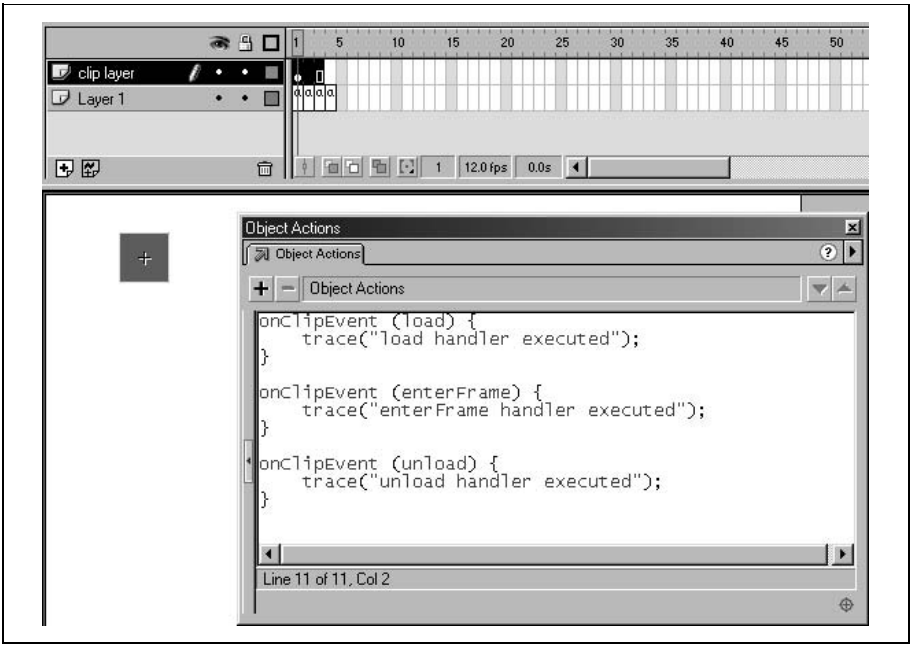


***Рис. 10.2.** Фильм для проверки очередности выполнения кода*

При воспроизведении фильма порядок выполнения кода будет таким:

```
========FRAME 1=======
1) Выполняется код главной временной диаграммы
2) Выполняется обработчик load
3) Выполняется внутренний код клипа, кадр 1

========FRAME 2=======
1) Выполняется обработчик enterFrame
2) Выполняется внутренний код клипа, кадр 2
3) Выполняется код главной временной диаграммы

========FRAME 3=======
1) Выполняется обработчик enterFrame
2) Выполняется внутренний код клипа, кадр 3
3) Выполняется код главной временной диаграммы

========FRAME 4=======
1) Выполняется обработчик unload
2) Выполняется код главной временной диаграммы
```

Очередность выполнения кода в нашем примере фильма демонстрирует некоторые важные правила, о которых нужно помнить при написании обработчиков событий:

- Код обработчика события *load* выполняется раньше, чем внутренний код клипа, поэтому обработчик *load* можно использовать для

инициализации переменных для кадра 1 клипа, в котором он находится.

- Перед созданием экземпляра клипа в кадре выполняется код этого кадра. В результате определяемые пользователем в клипе переменные и функции не станут доступны коду в его родительской временной диаграмме, пока на рабочем столе не появится впервые кадр, следующий *за* клипом, даже если эти переменные и функции объявлены в обработчике *load* клипа.
- 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 рабочем столе и исчезает с него.
- Во всех кадрах код обработчика события клипа *enterFrame* выполняется прежде кода родительской временной диаграммы клипа. Поэтому с помощью 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 клипа копируются, когда копируется клип с помощью функции *duplicateMovieClip( )*. Допустим, например, что на рабочем столе находится клип с именем `square`, в котором определен обработчик события *load*:

```
onClipEvent (load) {
  trace("movie loaded");
}
```

Что произойдет в результате копирования `square` в `square2`?

```
square.duplicateMovieClip("square2", 0);
```

Так как при копировании `square` обработчик *load* копируется в `square2`, появление `square2` влечет выполнение обработчика *load*, который выводит в окно Output сообщение «movie loaded». Пользуясь этим автоматическим сохранением обработчиков, можно создавать хитрые рекурсивные функции для получения мощных результатов. В примере 10.2 показана лишь малая часть открывающихся при этом возможностей.

# Обновление экрана с помощьюupdateAfterEvent

Как нам стало известно из раздела «Очередность выполнения кода», обработчики событий *mouseDown*, *mouseUp*, *mouseMove*, *keyDown* и

*keyUp* выполняются немедленно, как только происходят эти события. Именно *немедленно*, даже если соответствующее событие происходит в промежутке между выводом кадров.

Благодаря такой срочности фильм может обладать очень быстрой реакцией, но это качество легко потерять. По умолчанию зрительные эффекты, создаваемые обработчиками событий *mouseDown*, *mouseUp*, *mouseMove*, *keyDown* и *keyUp*, физически не выводятся проигрывателем Flash, пока не будет выведен следующий кадр. Чтобы увидеть, как это происходит, создайте фильм со скоростью 1 кадр в секунду и поместите на рабочий стол клип со следующим кодом:

```
onClipEvent (mouseDown) {
  _x += 2;
}
```

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, клип переместится вправо на 12 пикселов, когда будет выведен очередной кадр. Если вы щелкнете 3 раза, то клип переместится на 6 пикселов. Между кадрами регистрируется каждый обработчик *mouseDown*, но результаты отображаются только при выводе 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:

```
onClipEvent (mouseDown) {
  _x += 2;
  updateAfterEvent();
}
```

Функцию *updateAfterEvent( )* можно использовать только с событиями *mouseDown*, *mouseUp*, *mouseMove*, *keyDown* и *keyUp*. Часто это важно для того, чтобы действия пользователя вызывали быстрые и плавные зрительные изменения. Ниже, в примере 10.8, мы воспользуемся функцией *updateAfterEvent( )* для плавного вывода пользовательского указателя. Однако для событий кнопок не требуется явно вызывать функцию *updateAfterEvent( )*: кнопки обновляются обычным образом между выводом кадров.

# Повторное использование кода

При использовании событий кнопок и событий клипов не забывайте о принципах централизации кода, рассмотренных в главе 9. Всегда старайтесь избежать лишнего дублирования кода и распределения его по разным элементам фильма. Если вы обнаружите, что помещаете один и тот же код в тело нескольких обработчиков событий, прикрепление этого кода непосредственно к объекту может оказаться неразумным. Попытайтесь обобщить код, оторвав его от объекта и поместив в хранилище кода в каком-либо месте фильма, которым часто оказывается главная временная диаграмма.

Во многих случаях размещение предложений внутри обработчика событий кнопки или клипа – неудачное решение. Помните, что инкапсуляция кода в функции и вызов этой функции из обработчика позволяют повторно использовать код и легко находить его. Особенно это касается кнопок: лично я редко помещаю непосредственно в кнопку что-либо помимо команды вызова функции. С клипами нужно проявлять большую рассудительность, так как код, размещаемый непосредственно в клипах, часто составляет огромную часть хорошо сложенной независимой архитектуры кода. Попробуйте применить различные подходы, пока не найдете правильное соотношение между своими потребностями и уровнем мастерства. В любом случае, всегда полезно помнить о проблемах избыточности и повторного использования кода.

Пример, иллюстрирующий различие между прикреплением кода к кнопкам и вызовом функций из кнопок, можно найти в разделе «Централизация кода» главы 9.

# Динамические обработчики событий клипов

Ранее в этой главе было отмечено, что во Flash есть два вида событий: прикрепляемые к клипам и кнопкам и прикрепляемые к другим объектам данных, таким как *XML* и *XMLSocket*. Чтобы создать обработчик события для объекта данных, нужно назначить имя функции в качестве свойства объекта. Напомним синтаксис динамического добавления функции:

```
myXMLDoc.onLoad = function () { trace("all done loading!"); };
```

Динамическое присваивание функции позволяет изменить поведение обработчика во время воспроизведения фильма. Для этого необходимо лишь заново присвоить свойство обработчика:

```
myXMLDoc.onLoad = function () { gotoAndPlay("displayData"); };
```

Можно даже вообще отключить обработчик:

```
myXMLDoc.onLoad = function () { return; };
```

К сожалению, обработчики событий клипов и кнопок далеко не такие гибкие: их нельзя изменять или удалять во время воспроизведения фильма. Кроме того, обработчики событий клипов нельзя прикрепить к главной временной диаграмме какого-либо фильма! Так, невозможно напрямую создать обработчик события для клипа `_root`.

Чтобы обойти эти ограничения, можно – в случае событий *enterFrame* и событий, связанных с вводом пользователя, – использовать пустые клипы фильмов и моделировать динамическое удаление или изменение обработчика событий. Пустые клипы даже позволяют моделировать события уровня `_root`. Этот прием уже рассматривался в главе 8, где мы узнали, как создавать следующий цикл события:

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) {
_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 задайте 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( )*.

Пошаговые инструкции относительно использования этого приема с событием *enterFrame* можно найти в разделе «Циклы событий клипов Flash 5» главы 8.

# Применение обработчиков событий

Завершают наше изучение событий и обработчиков событий ActionScript несколько практических примеров. Это простые приложения, но они позволяют почувствовать, насколько гибким может оказаться программирование на основе событий. Два последние примера можно загрузить из сетевого Хранилища кода.

В примере 10.7 осуществляется сжатие и расширение клипа.

*Пример 10.7. Колебательное изменение размеров клипа*

```
onClipEvent (load) {
```

```
  var shrinking = false;
  var maxHeight = 300;
  var minHeight = 30;
}

onClipEvent (enterFrame) {
  if (_height < maxHeight && shrinking == false) {
    _height += 10;
    _width  += 10;
  } else {
    shrinking = true;
  }

  if (shrinking == true) {
    if (_height > minHeight) {
      _height -= 10;
      _width  -= 10;
    } else {
      shrinking = false;
      _height += 10;      // Инкрементировать, чтобы
      _width  += 10;      // не пропустить цикл
    }
  }
}
```

Пример 10.8 моделирует пользовательский курсор мыши, пряча обычный системный курсор и заставляя клип следовать за положением мыши на экране. В этом примере обработчики *mouseDown* и *mouseUp* несколько изменяют размер пользовательского указателя, чтобы обозначить щелчок мыши.

*Пример 10.8. Пользовательский указатель мыши*

```
onClipEvent (load) {
  Mouse.hide();
}

onClipEvent (mouseMove) {
  _x = _root._xmouse;
  _y = _root._ymouse;
  updateAfterEvent();
}

onClipEvent (mouseDown) {
  _width  *= .5;
  _height *= .5;
  updateAfterEvent();
}

onClipEvent (mouseUp) {
  _width  *= 2;
  _height *= 2;
  updateAfterEvent();
}
```

Наконец, просто для показа мощи обработчиков событий клипа ActionScript в примере 10.9 клип превращается в пользовательскую кнопку, при этом событие *mouseMove* используется для проверки нахождения курсора над кнопкой, события *mouseDown* и *mouseUp* используются для проверки щелчка мышью, а функция *hitTest( )* предельно упрощает проверку нахождения над активной областью. В этом примере предполагается, что в клипе с обработчиком есть три ключевых кадра с метками `up`, `down` и `over` (соответствующими обычным состояниям кнопки).

*Пример 10.9. Кнопка-клип*

```
onClipEvent (load) {
  stop();
}

onClipEvent (mouseMove) {
  if (hitTest(_root._xmouse, _root._ymouse, true) && !buttonDown) {
    this.gotoAndStop("over");
  } else if (!hitTest(_root._xmouse, _root._ymouse, true) && !buttonDown) {
    this.gotoAndStop("up");
  }
  updateAfterEvent();
}

onClipEvent (mouseDown) {
  if (hitTest(_root._xmouse, _root._ymouse, true)) {
    buttonDown = true;
    this.gotoAndStop("down");
  }
  updateAfterEvent();
}

onClipEvent (mouseUp) {
  buttonDown = false;
  if (!hitTest(_root._xmouse, _root._ymouse, true)) {
    this.gotoAndStop("up");
  } else {
    this.gotoAndStop("over");
  }
  updateAfterEvent();
}
```

# Вперед!

Освоив предложения, 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

Ранее в главе 3 «Данные и типы данных» говорилось 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 составным типом данных для изучения станет *массив* (*array*). Массивы используются для хранения упорядоченных списков и работы с ними, и потому являются базовым средством для последовательного программирования с многократными повторениями. Они применяются к таким задачам, как хранение значений, полученных из формы, заполненной пользователем, создание выпадающих меню, слежение за неприятельским космическим кораблем в игре и т. д. В наиболее общем 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 – это 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 сначала показаны две строки по отдельности, а потом массив, содержащий две строки:

```
"oranges"              // Одно элементарное строковое значение
"apples"               // Еще одно элементарное строковое значение
["oranges", "apples"]  // Один массив, содержащий две строки
```

Массив является контейнером 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 каждого товара, который вы собираетесь приобрести:

```
["oranges", 6, "apples", 4, "bananas", 3]
```

Вот еще одна аналогия, позволяющая сделать понятие массива более осязаемым. Представьте себе комод с ящиками. В каждом отдельном ящике есть некоторое содержимое (носки, рубашки и т. д.). Но в самом комоде нет *содержимого*, он содержит в себе только *ящики*. Содержимое хранят ящики. Комод объединяет ящики в единое целое.

При работе с массивом (комодом) нас обычно интересуют значения, хранящиеся в массиве (содержимое ящиков). Значения, хранимые в массиве, являются информацией, которой мы хотим распоряжаться. Но можно работать и с самой структурой хранения. Можно, например, изменить размер массива (добавить или вытащить ящики) или переупорядочить значения (обменять ящики содержимым).

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, считается единой строкой, и число из нескольких цифр считается одним числом. В качестве одиночного элемента данных массив можно присвоить переменной или использовать 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 можно на него ссылаться.

## Элементы массива

Как и переменная, каждый элемент массива может хранить любой допустимый элемент данных. Поэтому массив в целом похож на совокупность последовательно поименованных переменных, но вместо того, чтобы давать каждому отдельному элементу имя, он его нумерует (при этом номер первого элемента 0, а не 1).

На рис. 11.1 схематически изображена структура массива из трех элементов. Элемент 0 содержит значение «Эрика», элемент 1 – «Славик», а элемент 2 – «Гэри».

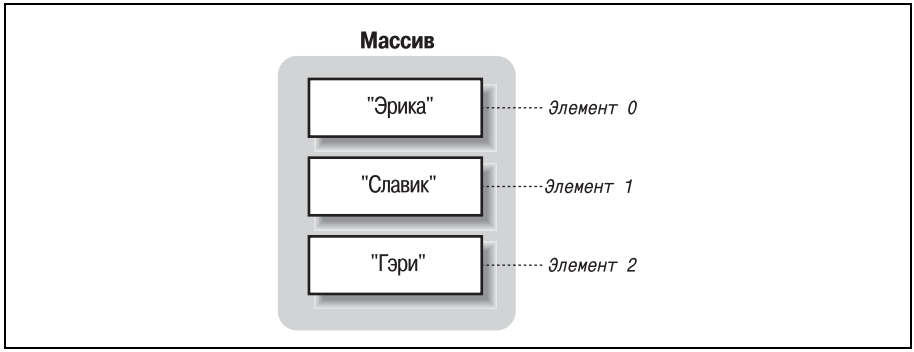


***Рис. 11.1.*** *Пример структуры массива*

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 с помощью его индекса (в данном случае индекс равен 6). Индекс элемента используется для присвоения элементу значения или извлечения значения, либо работы с элементом каким-либо другим способом. Например, некоторые функции при работе с массивами используют индексы массивов для задания диапазона элементов, подлежащих обработке.

Кроме того, можно вставлять и удалять элементы в начало, конец и даже середину массива. В массиве могут быть пустые промежутки (т. е. некоторые элементы могут отсутствовать). Например, в позициях 0 и 4 могут быть элементы, а в позициях 1, 2 и 3 – отсутствовать. Массивы с пустыми промежутками называются *разреженными* (*sparse*).

## Размер массива

В каждом массиве в любой данный момент срока его жизни содержится определенное количество элементов. Количество элементов, которые могут быть размещены в массиве, называется его *длиной* (*length*), о которой мы поговорим позднее.

# Создание массивов

Новый массив можно создать с помощью литерала данных (т. е. просто набрав в тексте все элементы) или с помощью специальной встроенной функции – конструктора массивов *Array( )*.

## Конструктор массивов

Для создания массива с помощью конструктора *Array( )* используется оператор *new*, за которым следуют ключевое слово *Array* и круглые скобки, в результате чего получается пустой (т. е. не содержащий элементы) массив. Обычно вновь созданный массив присваивается переменной или другому контейнеру данных, которые позволяют обращаться к массиву в будущем. Например:

```
var myList = new Array();// Сохранить пустой массив в переменной myList
```

Часто нужно 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( )* нечислового аргумента или просто нескольких аргументов каждый аргумент становится одним из значений элементов нового массива, например:

```
var frameLabels = new Array("intro", "section1", "section2", "home");
```

Массив, хранящийся в `frameLabels`, будет содержать следующие элементы:

```
0: "intro"
1: "section1"
2: "section2"
3: "home"
```

При передаче конструктору *Array( )* ровно одного числового аргумента создается массив с указанным числом 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 и 1
```

### Массивы в других языках программирования

Почти любой язык программирования высокого уровня поддерживает тот или иной тип массивов 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 либо числа, либо строки, но не одновременно. Интересно, что в языке C отсутствует элементарный тип данных *string*. Вместо этого в C поддерживается тип односимвольных данных с именем *char*; 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 – это называется заданием 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 одном массиве данные разных типов и при необходимости автоматически изменяет их размер. В отличие от ActionScript и C, в которых массив нумеруется с нуля (так называемый *0-относительный* (*zero-relative*) массив), в Lingo он нумеруется с единицы (*1-относительный* (*one-relative*) массив).

Языки различаются и по тому, что происходит при попытке доступа к элементу, находящемуся за границами массива. ActionScript и Lingo при попытке присвоить значение элементу за пределами массива просто добавят этот элемент. Если же попытаться получить значение элемента за пределами массива, ActionScript возвратит `undefined`, а в Lingo произойдет ошибка. Язык C не проверяет, является ли допустимым номер элемента, к которому происходит обращение. Он позволяет извлекать и задавать значения элементов за пределами массива, что обычно влечет изменение в памяти других данных или чтение бессмысленных данных, не принадлежащих массиву (C вообще часто предоставляет веревку, на которой можно повеситься).

## Литералы массивов

Иногда удобнее создавать массивы с помощью литералов, а не конструктора *Array( )*. Напомним, что *литерал* является прямым представлением фиксированного участка определенных данных. Например:

```
"beaver"      // Строковый литерал
234.2034      // Числовой литерал
true          // Булев литерал
```

В литерале массива о начале и конце массива сигнализируют квадратные скобки. Внутри квадратных скобок задаются значения элементов массива в виде списка выражений, разделяемых запятыми. Вот общий синтаксис:

```
[expression1, expression2, expression3]
```

Выражения вычисляются, и полученные значения сохраняются в виде элементов описываемого массива. Могут использоваться любые допустимые выражения, в том числе вызовы функций, переменные, литералы и даже другие массивы (массив, находящийся внутри другого массива, называется *вложенным*). 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
```

Сравним, как то же самое делается 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))
```

Обратите внимание, что в ActionScript элементы одного массива могут содержать данные разных типов, как было отмечено выше во врезке.

# Обращение к элементам массива

Когда массив создан, неизбежно потребуется извлечь или изменить значения его элементов. Для этого можно использовать квадратные скобки `[]` (т. е. *оператор доступа к массиву, array access operator*), которые рассматривались в главе 5 «Операторы».

## Извлечение значения элемента

Чтобы получить значение элемента, нужно просто сослаться на него, указав его индекс в квадратных скобках, например:

```
arrayName[elementNumber]
```

где *arrayName* представляет массив, а *elementNumber* может быть любым выражением, дающим числовое значение. Номер первого элемента – 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 переменной 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 соответствует третьему элементу!!)
var favoriteTree = trees[2];  // favoriteTree получает значение "oak"
```

Теперь интересное. Так как индекс элемента можно задать в виде любого выражения, дающего число, то в качестве индекса можно использовать и переменную, например:

```
var i = 3;
var lastTree = trees[i];  // Присвоить lastTree значение "cedar"
```

Возможно даже использование в этом качестве выражений, содержащих вызовы функций, которые возвращают числовые значения:

```
// Присвоить переменной 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 массиву и доступом к значению переменной. Элементы массива можно включать в состав сложных выражений, например, так:

```
var myNums = [12, 4, 155, 90];
  // Сложить элементы массива
var myTotal = myNums[0] + myNums[1] + myNums[2] + myNums[3];
```

Использованный в предыдущем примере подход к суммированию значений элементов массива не служит эталоном оптимального кода. Ниже будет показан значительно более быстрый и удобный способ последовательного доступа к элементам массива.

## Присваивание элементу значения

Чтобы присвоить значение элементу массива, в качестве левого операнда в выражении присвоения используется *arrayName*[*elementNumber*]:

```
// Создать массив
var cities = ["Toronto", "Montreal", "Vancouver", "Waterloo"];
```

```
// Теперь в cities: ["Toronto", "Montreal", "Vancouver", "Waterloo"]

// Присвоить значение первому элементу массива
// cities становится ["London", "Montreal", "Vancouver", "Waterloo"]
cities[0] = "London";

// Присвоить значение четвертому элементу массива
// cities стал ["London", "Montreal", "Vancouver", "Hamburg"]
cities[3] = "Hamburg";

// Присвоить значение третьему элементу массива
// cities стал ["London", "Montreal", 293.3, "Hamburg"]
cities[2] = 293.3;  // Обратите внимание, что можно изменять тип данных
```

Обратите внимание, что при присваивании значения элементу массива можно использовать в качестве индекса любое числовое выражение:

```
var i = 1;
// Set the value of element i
// cities стал ["London", "Prague", 293.3, "Hamburg"]
cities[i] = "Prague";
```

# Определение размера массива

У всех массивов есть встроенное свойство `length`, показывающее текущее количество элементов (включая пустые). Для доступа к свойству `length` массива используется оператор точка, например:

```
arrayName.length
```

Свойство массива `length` сообщает о том, сколько элементов находится в массиве. 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. Неужели 51? Да, это так: несмотря на то что индексы с 3 по 49 пусты, они учитываются в длине массива. Последним элементом массива всегда является *myArray*[*myArray*.length - 1], т. к. 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 этому свойству значения, тем самым добавляя или удаляя элементы в конце массива.

Может возникнуть вопрос, какая польза от свойства length. С помощью свойства length можно создать цикл для перебора всех элементов массива, подобного показанному в примере 8.1. Перебор элементов массива является базовой задачей, возникающей в программировании. Чтобы получить представление о возможностях, возникающих при комбинировании циклов и массивов, изучите пример 11.1, где код рыщет по массиву soundtracks в поисках элемента со значением «hip hop». Цикл *for* должен быть вам знаком по главе 8 «Предложения цикла», а оператор инкрементирования – по главе 5. Что касается написания кода для доступа к массиву, 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 драгоценных моментов, когда тяжелые усилия привели к созданию единой системы. Тривиальные с виду отдельные операции в своем сочетании могут образовывать чрезвычайно мощные и гибкие программы. Из строительных элементов, находящихся в вашем распоряжении, можно создавать любые мыслимые конструкции, как из букв алфавита или последовательностей аминокислот в цепочке ДНК. В оставшейся части главы более подробно рассказывается о технике работы с массивами, в том числе об использовании встроенных функций, уже готовых к тому, чтобы вместо вас выполнять часто встречающиеся задачи. Если возникшую задачу нельзя решить с помощью встроенных функций, можно написать собственную функцию, такую, например, как функция *searchArray( )* из предыдущего примера.

# Именованные элементы массивов

Обычно элементы массивов нумеруются, но можно также давать им имена. С помощью именованных элементов можно моделировать так называемые *ассоциативные массивы* (*associative arrays*), или *хеши* (*hashes*). Отметим, что именованные элементы массива нельзя обрабатывать методами класса Array (*push( )*, *pop( )* и т. д., о которых будет рассказано ниже) и они не входят в состав списка нумерованных элементов. Массив из одного именованного элемента и двух нумерованных имеет длину 2, а не 3. Поэтому для перебора всех именованных элементов массива нужно применять цикл *for-in* (о котором рассказывалось в главе 8), пробегающий как именованные, так и нумерованные элементы.

## Создание именованных элементов массива и обращение к ним

Для добавления в массив элемента, который потом можно будет извлекать по имени, к существующему массиву применяются уже знакомые нам квадратные скобки, содержащие строку вместо числа:

```
arrayName[elementName] = expression
```

Здесь *elementName* является строкой. Например:

```
var importantDates = new Array();
importantDates["dadsBirthday"] = "June 1";
importantDates["mumsBirthday"] = "January 16";
```

Можно также воспользоваться оператором «точка»:

```
arrayName.elementName = expression
```

В этом случае *elementName* является идентификатором, а *не* строкой. Например:

```
var importantDates = new Array();
importantDates.dadsBirthday = "June 1";
importantDates.mumsBirthday = "January 16";
```

В предположении, что идентификатор элемента известен, (например, dadsBirthday в массиве importantDates), можно обратиться к нему одним из двух способов:

```
var goShopping = importantDates["dadsBirthday"];
var goShopping = importantDates.dadsBirthday;
```

Как в случае присвоения значения именованному элементу, так и в случае чтения его значения, при использовании квадратных скобок *elementName* является строкой или выражением, возвращающим строку. Аналогично, при использовании оператора *точки* *elementName* является *идентификатором* (т. е. именем элемента без кавычек), а не строкой.

## Удаление именованных элементов

Чтобы избавить массив от ненужного именованного элемента, применяется оператор *delete*, о котором сообщалось 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. (Сравните это с поведением оператора *delete* при удалении *нумерованных* элементов, который удаляет только значение, а контейнер сохраняет).

# 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

Чтобы добавить в существующий массив новый элемент по заданному индексу, можно просто присвоить этому элементу значение:

```
// 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 пустые элементы для промежуточных индексов:

```
// Оставить пустыми индексы с 4 по 38
myList[39] = "grapes";

trace (myList[12]); // Вывод пуст, так как элемент 12 не определен
```

## Добавление новых элементов с помощью свойства length

Чтобы расширить массив, не присваивая значений новым элементам, можно просто 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 выделить место для данных, которые вы рассчитываете получить, например, для результатов сдачи экзаменов студентами.

## Добавление элементов с помощью методов класса Array

С помощью встроенных методов массивов можно осуществлять выполнение более сложных сценариев добавления элементов. (В главе 12 «Объекты и классы» будет объяснено, что *метод* – это функция, действующая над объектом.)

### Метод push()

Метод *push( )* добавляет один или более элементов в конец массива. Он автоматически дописывает данные вслед за последним элементом массива, поэтому нет необходимости беспокоиться о реальном количестве существующих элементов. Метод *push( )* может и добавлять в массив одновременно 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* составляют разделяемый запятыми список элементов, добавляемых в конец массива в качестве новых элементов. Вот несколько примеров:

```
// Создать массив из двух элементов
var menuItems = ["home", "quit"];

// Добавить один элемент
// menuItems теперь имеет вид ["home", "quit", "products"]
menuItems.push("products");

// Добавить еще два элемента
// menuItems становится ["home", "quit", "products", "services", "contact"]
menuItems.push("services", "contact");
```

При вызове без аргументов *push( )* добавляет один пустой элемент:

```
menuItems.push();  // Увеличить длину массива на 1

// То же, что
menuItems.length++;
```

Метод *push( )* возвращает новую длину изменившегося массива:

```
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 методом *push( )*, но элементы (один или несколько) добавляются в *начало* массива, проталкивая дальше все уже существующие элементы (т. е. индексы существующих элементов увеличиваются, чтобы дать место новым элементам в начале массива). Синтаксис *unshift( )* следует тому же стилю, что и все методы массивов:

```
arrayName.unshift(item1, item2,...itemn);
```

Здесь *item1*, *item2*,...*itemn* составляют разделяемый запятыми список элементов, добавляемых в начало массива в качестве новых элементов. Вот несколько примеров:

```
var flashVersions = new Array();
```

```
flashVersions[0] = 5;
flashVersions.unshift(4);    // flashVersions теперь [4, 5]
flashVersions.unshift(2,3);  // flashVersions теперь [2, 3, 4, 5]
```

Метод *unshift( )*, как и *push( )*, возвращает новую длину увеличившегося массива.

## Стеки и методы push и pop

Имя метода *push( )* берет начало от понятия программирования, называемого *стеком*. Стек можно представить себе в виде вертикального массива, похожего на стопку тарелок. Если вы бываете в кафетериях и ресторанах с буфетом, то вам должны быть знакомы стойки для тарелок. Новые чистые тарелки буквально *проталкиваются* (*pushed*) на самый верх стопки (stack) а старые тарелки проваливаются глубже в стойку. Когда клиент *выталкивает* (*pop*) тарелку из стопки, он берет ту тарелку, которая была помещена в стопку последней. Это называют стеком – «последним пришел – первым ушел» (last-in-first-out, LIFO) и обычно используют в таких вещах, как буферы команд. Например, если вы в броузере щелкнете по кнопке «Назад», он переместит вас на предыдущую страницу, где вы побывали. Если снова щелкнуть по этой кнопке, вы переместитесь назад еще на одну страницу и т. д. Это осуществляется путем проталкивания (*pushing*) адреса каждой посещаемой страницы на стек и выталкивания (*popping*) ее со стека при щелчке по кнопке «Назад».

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 первым сдал свой багаж, 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 действует так, чтобы первым обслужить того, кто пришел первым. Стек FIFO – это как очередь в местном банке. Вместо того чтобы взять последний элемент массива, стек FIFO в качестве очередного элемента обрабатывает первый элемент массива. Затем он удаляет первый элемент, и 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 устал ждать). Поэтому слово *push* обычно подразумевает, что используется стек типа LIFO, в то время как *append* подразумевает, что используется стек типа FIFO. В любом случае элементы добавляются в «конец» стека. Разница заключается 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 используется 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 этих 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* является числом, задающим индекс, с которого должны начаться удаление элементов и, возможно, их вставка (помните, что индекс первого элемента равен 0); *deleteCount* является необязательным аргументом, определяющим количество удаляемых элементов (включая элемент с индексом *startIndex*). Если *deleteCount* опущен, удаляются все элементы, начиная с *startIndex* включительно. Необязательные параметры *item1*, *item2*,...*itemn* составляют разделяемый запятыми список элементов, добавляемых к массиву, начиная с *startIndex*.

Пример 11.3 демонстрирует многосторонность метода *splice( )*.

*Пример 11.3. Использование метода массива splice( )*

```
// Составить массив...
months = new Array("January", "Friday", "April", "May", "Sunday", "Monday", "July");
// Н-да. Что-то с нашим массивом не так. Нужно 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 (т. е. с "April")
months.splice(3); // Теперь months: ["January", "February", "March"]
```

Другая полезная особенность метода *splice( )* состоит в том, что он возвращает массив тех элементов, которые *удалил*. Поэтому его можно использовать 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.

## Метод concat()

Подобно методу *push( )*, метод *concat( )* добавляет элементы в конец массива. Но, в отличие от *push( )*, *concat( )* не изменяет массив, к которому был применен – новый массив он только возвращает. Кроме того, *concat( )* может разбирать массивы, переданные ему в качестве аргументов, на отдельные элементы и объединять два массива в один новый. Вот синтаксис метода *concat( )*:

```
origArray.concat(elementList)
```

Метод *concat( )* добавляет элементы, содержащиеся в *elementList*, по одному в конец массива *origArray* и возвращает результат как *новый* массив, оставляя *origArray* в неприкосновенности. Обычно возвращаемый массив сохраняется в переменной. Ниже в качестве элементов, добавляемых в массив, используются 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( )* используется 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( )* выделил элементы в массиве `guests`, чего не стал бы делать *push( )*. Если выполнить такой код:

```
var allUsers = registeredPlayers.push(guests);
```

то получится следующий вложенный массив:

```
["Gray", "Shift", "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

Элементы можно удалить из массива с помощью оператора *delete* либо как результат уменьшения свойства массива `length`, либо с помощью одного из встроенных методов массивов.

## Удаление элементов с помощью оператора delete

Оператор *delete* присваивает элементу массива значение `undefined` с использованием следующего синтаксиса:

```
delete arrayName[index]
```

Здесь *arrayName* является любым массивом, а *index* – номер или имя элемента, значению которого требуется присвоить `undefined`. Честно говоря, название оператора *delete* (удалить) вводит в заблуждение. Метод *не* удаляет элемент из массива, а только присваивает ему значение `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
```

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 со свойствами объектов и именованными элементами, нежели с нумерованными элементами. Применение *delete* к первым из них навсегда разрушает свойства и именованные элементы, не оставляя никаких следов.

## Удаление элементов с помощью свойства length

Ранее свойство `length` было использовано для добавления элементов в массив. Можно также присвоить свойству массива `length` значение, меньшее текущей длины, чтобы удалить элементы из массива (т. е. выполнить 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 последние)/
```

# Удаление элементов с помощью методов класса Array

Существует несколько встроенных методов удаления элементов массива. Уже было показано, как с помощью метода *splice( )* удалить последовательность элементов из середины массива. Методы *pop( )* и *shift( )* используются для удаления элементов из конца или 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» массива выглядит смешно.) Тем не менее, *pop( )* уменьшает длину массива на 1 и возвращает удаленный элемент. Например:

```
x = [56, 57, 58];
x.pop();     // x теперь [56, 57]
```

Как уже отмечалось, *pop( )* часто используется в сочетании с *push( )* в операциях со стеком LIFO. В примере 11.4 массив `siteHistory` используется для отслеживания перемещений пользователя по сайту. Когда пользователь переходит в новый кадр, его местонахождение добавляется в массив с помощью метода *ush( )*. 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);
    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
    gotoAndStop(siteHistory[siteHistory.length - 1]);
  } else {
    // ...иначе идем домой
    gotoAndStop("home");
  }
  trace(siteHistory);
}

// КОД В КНОПКЕ НАВИГАЦИИ
on (release) {
  goto("gallery");
}

// КОД В КНОПКЕ  НАЗАД
on (release) {
  goBack();
}
```

## Метод shift()

Помните *unshift( )*, метод, применявшийся для добавления элемента в начало массива? Познакомьтесь с его парой – методом *shift( )*, который удаляет элемент из начала массива:

```
arrayName.shift()
```

Не так смешно, как использование *pop*, и вполне хорошо.

Как и *pop( )*, *shift( )* возвращает значение удаленного им элемента. Оставшиеся элементы перемещаются вверх в начало массива с сохранением порядка. Напимер:

```
var sports = ["hackey sack", "snowboarding", "inline skating"];
sports.shift();  // Теперь ["snowboarding", "inline skating"]
sports.shift();  // Теперь ["inline skating"]
```

Так как *shift( )* действительно удаляет один элемент, этот метод полезнее применять для удаления первого элемента массива, чем *delete*. Можно также применять *shift( )* для ограничения диапазона списка. Допустим, например, что мы рассчитываем частоту смены кадров фильма. После вывода каждого кадра мы отправляем текущее время в массив с помощью метода *push( )*. Чтобы ограничить размер массива последними 10 значениями времени, при необходимости выполняем *shift( )* для самого старого отсчета. Чтобы найти скорость воспроизве-

дения фильма, усредняем значения, хранящиеся в массиве. Этот способ продемонстрирован в примере 11.5.

*Пример 11.5. Расчет скорости воспроизведения фильма*

```
// Создать массив значений времени в клипе
onClipEvent(load) {
  var elapsedTime = new Array();
}

// Используем событие клипа enterFrame для определения времени после
// каждого кадра
onClipEvent(enterFrame) {
  // Добавляем текущее время в elapsedTime
  elapsedTime.push(getTimer());

  // Если чисел достаточно для вычисления среднего...
  if (elapsedTime.length > 10) {
    // ...удалить из elapsedTime самый старый отсчет
    elapsedTime.shift();

    // Вычислить среднее число миллисекунд на кадр
    elapsedAverage = (elapsedTime[elapsedTime.length - 1] -
                      elapsedTime[0]) / elapsedTime.length;

    // Число кадров в секунду: делим 1 на среднее время воспроизведения
    fps = 1000 / elapsedAverage;
    trace("current fps " + fps);
  }
}
```

### Метод splice()

Ранее отмечалось, что метод *splice( )* может как удалять элементы из массива, так и добавлять их. Так как метод *splice( )* уже был подробно рассмотрен, не будем здесь повторно возвращаться к его изучению. Однако в данном контексте специально 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

Ранее было показано, как применять методы массивов для удаления элементов из массивов и добавления их. ActionScript предлагает также встроенные методы для переупорядочения 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. Вот его синтаксис:

```
arrayName.reverse()
```

А вот впечатляющие 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 инверсии порядка элементов в массиве. Это не только труднее, чем кажется: скорее всего, встроенный метод *reverse( )* окажется существенно быстрее.

## Метод sort()

Метод *sort( )* переупорядочивает последовательность элементов массива в соответствии с любым заданным правилом. Если правило не задано, то *sort( )* по умолчанию размещает элементы, грубо говоря, в алфавитном порядке. Отсортировать массив по алфавиту действительно просто:

```
arrayName.sort()
```

При вызове метода массива *sort( )* без аргументов его элементы временно преобразуются в строки и сортируются соответственно кодовым позициям символов, приведенным в приложении B «Набор символов Latin 1 и коды клавиш» (важные детали также можно найти в разделе «Сравнение по порядку символов и алфавиту» главы 4 «Элементарные типы данных»):

```
// Это работает, как ожидалось...
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 использовать *sort( )* для организации элементов массива в соответствии с собственным правилом. Это более сложный в работе прием, но весьма мощный. Сначала нужно создать *функцию сравнения* (*compare function*), определяющую, как интерпретатор должен сортировать любые два элемента массива. Затем эта функция передается методу *sort( )* при его вызове, например:

```
arrayName.sort(compareFunction)
```

где *compareFunction* является именем функции, сообщающей интерпретатору, как принимать решение при сортировке.

Чтобы построить функцию сравнения, нужно создать новую функцию, которая принимает два аргумента (представляющие любые два элемента массива). В теле функции нужным образом определяется, который из элементов должен появиться раньше в списке элементов после сортировки методом *sort( )*. Если нужно, чтобы первый элемент появился *прежде* второго, функция должна возвратить *отрицательное* число. Если нужно, чтобы первый элемент появился *после* второго, функция должна возвратить *положительное* число. Если нужно оставить элементы в том же положении, функция должна возвратить 0. На псевдокоде этот подход в общем выглядит так:

```
function compareElements (element1, element2) {
  if (условие для появления element 1 раньше element2) {
    return -1;
  } else if (условие для появления element1 после element2) {
    return 1;
  } else {
    return 0;  // Оставить элементы в покое
  }
}
```

Например, чтобы отсортировать элементы по возрастанию числовых значений, можно использовать такую функцию:

```
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 оба числа равны. Теперь это более элегантно! Вот версия 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 безотносительно к регистру.

*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. Сделаем сортировку массива 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!

## Метод slice()

Будучи своего рода подмножеством *splice( )*, метод *slice( )* возвращает из массива ряд элементов. В отличие от *splice( )*, метод *slice( )* только

*возвращает* элементы. Он создает новый массив и не оказывает воздействия на массив, для которого был вызван. Синтаксис метода *slice( )* следующий:

```
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 пара примеров:

```
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, изображающей все элементы массива. Метод *join( )* начинает работу с преобразования каждого элемента указанного массива в строку. Пустые 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* является строкой, служащей для разделения преобразованных элементов массива *arrayName*. Если *delimiter* не задан, по умолчанию используется запятая. Действие предложения *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 *join( )* не изменяет массива, к которому применен, а возвращает строку, получающуюся из этого массива.

При вызове без аргумента *join( )* ведет себя в точности как *toString( )*. Так как *toString( )* не добавляет пробела после запятой, используемой им для разделения элементов, то для получения красиво форматированной выдачи более удобно применение метода *join( )*:

```
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, *сами* являющиеся 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 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"
```

## Метод toString()

Как будет показано в главе 12, *toString( )* служит универсальным для некоторых объектов методом, возвращающим строковое представление любого объекта, с которым он вызывается. В случае объекта *Array* метод *toString( )* возвращает список элементов массива, преобразованных в строки и разделенных запятыми. Метод *toString( )* можно вызвать явно:

```
arrayName.toString()
```

Но обычно метод *toString( )* не используется явным образом, а вызывается автоматически, когда *arrayName* используется в строковом контексте. Например, когда мы пишем *trace(arrayName)*, в окне Output появляется список значений, разделенных запятыми; запись *trace(arrayName)* эквивалентна *trace(arrayName.toString( ))*. Метод *toString( )* часто удобно использовать 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 = "the sites array is " + sites.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 массивам можно добавлять свойства. При добавлении к массиву именованных элементов в действительности происходит просто добавление свойств к объекту *Array*. Однако, как было отмечено выше, для использования встроенных методов класса *Array* нужно работать с нумерованными элементами.

В главе 9 «Функции» был рассмотрен полезный гибрид «массив-объект» – объект *arguments*, в котором нумерованные элементы сочета-

лись со свойствами. (Вспомните, что у объекта *arguments* есть свойство `callee`, и в нем также хранится массив параметров в виде массива элементов.) Теперь можно понять, что все массивы хранят пронумерованный список элементов плюс встроенное свойство `length`, плюс любые свойства, добавленные пользователем.

# 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 товара, их количество и цену. Требуется моделировать таблицу из трех строк (по одной для каждого наименования) и двух столбцов (один для количества, 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 (один для строки, другой для столбца). Выражение `spreadsheet[0]`, к примеру, представляет массив из двух столбцов первой строки. Поэтому для доступа ко второму столбцу первой строки таблицы `spreadsheet` используется `spreadsheet[0][1]`:

```
// Создать переменную для хранения суммарной стоимости заказа
var total;

// Теперь найти стоимость заказа. Для каждой строки перемножить
// столбцы и прибавить 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, с успехом можно использовать все методы класса *Array*.

# Вопросник с вариантами выбора: подход № 3

В примере 9.10 был переиначен вопросник с вариантами выбора из примера 1.1. Тогда мы усовершенствовали код, создав многократно используемые централизованные функции. После изучения массивов можно вернуться к вопроснику и сделать несколько усовершенствований. На этот раз посмотрим, как с помощью массивов хранить ответы пользователя и правильные ответы на каждый вопрос, что сделает код более кратким, разборчивым и легко расширяемым, если нужно будет добавить новые вопросы.

Все изменения, которые нужно произвести в вопроснике, будут теперь касаться только слоя *scripts* кадра 1, где расположен основной код вопросника. Как предыдущую, так и данную обновленную версию вопросника можно найти в сетевом Хранилище кода. Чтобы обнаружить, какие изменения будут сделаны на этот раз, прочтите код примера 11.7, уделив особое внимание комментариям.

*Пример 11.7. Опросник с вариантами выбора, версия 3*

```
stop();
//  *** Init main timeline variables
var displayTotal;               // Текстовое поле для вывода результата
var numQuestions = 2;           // Число вопросов
var totalCorrect = 0;           // Число правильных ответов
var userAnswers = new Array();  // Массив ответов пользователя
var correctAnswers = [3, 2];    // Массив правильных ответов
// Обратите внимание, что больше не нужны два длинные списка переменных с
// именами q1answer, q2answer ...., и correctAnswer1, correctAnswer2 ...
// Теперь нам удобно хранить эти данные в массиве.

// *** Функция для регистрации ответов пользователя
function answer (choice) {
  // Так как теперь можно проверить, сколько ответов дано, с помощью
  // свойства length массива userAnswers, нет нужды вручную отслеживать
  // эти данные с помощью свойства answer.currentAnswer. Заметим также, что
  // мы избавились от неуклюжего предложения set, которое ранее присваивало
  // ответ пользователя динамически именованной переменной.

  // Поместим ответ пользователя в наш массив
  userAnswers.push(choice);
```

```
  // Двигайся, детка
  if ((userAnswers.length) == numQuestions) {
    gotoAndStop ("quizEnd");
  } else {
    gotoAndStop ("q"+ (userAnswers.length + 1));
  }
}

// *** Функция для подсчета результата пользователя
function gradeUser() {
  // Подсчитать число правильных ответов. Наше сравнение элементов
  // userAnswer/correctAnswer гораздо аккуратнее змеиной ямы с функциями
  // eval(), использовавшимися для той же цели в прежней версии.
  for (var i = 0; i < userAnswers.length; i++) {
    if (userAnswers[i] == correctAnswers[i]) {
      totalCorrect++;
    }
  }
  // Вывести результат пользователя в динамическом текстовом поле
  displayTotal = totalCorrect;
}
```

**Упражнение для читателя: попробуйте обобщить версию вопросника, использующую массивы, воспользовавшись** `correctAnswers.length` **вместо переменной** `numQuestions`**.**

# Вперед!

В этой главе вы познакомились со своим первым составным типом данных – массивами – и значительно попрактиковались в их использовании. Теперь, когда вы знаете, как несколько элементов данных можно представить в одном контейнере, ваша подготовка стала достаточной для того, чтобы приступить к следующей главе. В ней мы исследуем еще более мощный тип составных данных и фундаментальную составляющую ActionScript – *объекты*.

# 12

# Объекты и классы

В этой главе рассказывается о так называемом *объектно-ориентированном программировании* (*ООП*), которое для многих читателей является неизведанной областью. Мы немного расскажем об используемой в нем технологии и приведем для большей конкретности несколько практических примеров. Возможно, вы слышали о таинственности ООП и трудности его понимания. На самом деле, идеи ООП весьма интуитивны, и процесс ООП значительно легче, чем вас пытались убедить. По сути, ООП просто означает, что некоторые части программы рассматриваются как независимые объекты. Эту идею легко уловить, если понять, что все, с чем вы имеете дело в реальном мире, является независимыми объектами. Ваша собака, родители, машина и компьютер – все это независимые объекты. Имеется в виду, что некоторые вещи они выполняют независимо, а другие – по вашей просьбе, даже если вам не известен внутренний механизм их работы.

Не нужно быть биологом, чтобы заставить свою собаку принести палку. Не нужно быть механиком, чтобы вести машину. Не нужно быть психоаналитиком, чтобы общаться с родителями, и, вопреки слухам, не нужно быть специалистом в вычислительной технике, чтобы получать электронную почту. Все, что требуется знать, это команды, которым готов подчиняться объект (называемые *методами*), и результаты выполнения этих команд. Например, если нажать на педаль газа, можно предполагать, что машина увеличит скорость. Если вы даете собаке команду «сидеть!», то можете предполагать, что она сядет. Вооружившись такими житейскими представлениями об объектах реального мира, посмотрим, какую связь имеют эти представления с ActionScript.

Классическим примером программного *объекта* служит прыгающий шарик. Как и у настоящего шарика, у объекта ball могут иметься

*свойства*, соответствующие его атрибутам, такие как радиус, цвет, масса, местонахождение и прыгучесть (упругость). Для представления прыгающего шарика в программе создадим объект `ball` со свойствами `radius` и прочими. Свойства объекта отображают его состояние в любой заданный момент, и некоторые из них со временем меняются. Например, чтобы заставить шарик двигаться, нужно 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 псевдоуравнения:

```
ball.xPosition += ball.xVelocity * (elapsedTime)
```

Это уравнение начинает с текущего положения шарика и добавляет пройденное расстояние (исходя из его скорости и прошедшего времени), чтобы получить новое положение. *Поведением* объекта называют просто правила, по которым он управляется, вроде нашего уравнения для расчета положения шарика в зависимости от времени. Обычно для такого поведения создаются оболочки в виде так называемых *методов*, которые являются просто *функциями*, *реализующими поведение объекта*. Например, можно создать метод *move( )*, реализующий предыдущее уравнение.

Методы, таким образом, можно рассматривать как команды, которым подчиняется объект. Конечно, у объекта может быть несколько методов. Допустим, нужно заставить шарик отскакивать. Можно создать метод *bounce( )*, который меняет направление движения шарика на противоположное и уменьшает его скорость (шарик не является абсолютно упругим). Метод *bounce( )* может реализовывать такое уравнение:

```
ball.xVelocity =  -(ball.xVelocity) * 0.95   // Упругость шарика 95%
```

Прежде чем проникнуть в тайны создания объектов, добавления свойств и реализации методов, формализуем некоторые из определений. *Объект* формально является структурой данных, объединяющей связанные между собой *свойства* и *методы* (функции). Обычно объект *инкапсулирует* свое поведение: это означает, что внутренние подробности выполнения им своих функций могут быть не видны снаружи объекта. Вместо этого программа взаимодействует с объектом посредством так называемых *интерфейсов* (т. е. методов, доступ к которым открыт вне объекта). Другим частям программы обычно нечего волноваться по поводу того, каким образом объект делает то, что он делает: программа просто поставляет объекту входные данные и проверяет выходные (результаты), если нужно. Вы также услышите о *классах* и *экземплярах (instances)*. *Класс* является родовой категорией объектов, а *экземпляр* является просто конкретным случаем (т. е. экземпляром) объекта. Например, ваша конкретная домашняя собака является экземпляром общего класса *Dog*. Все собаки класса *Dog* лают, и у них

по четыре ноги, но у вашей конкретной собаки собственные значения свойств роста, веса и масти, используемых для описания собак.

Объектно-ориентированное программирование (ООП) – это всего лишь название для программ, которые используют объекты. Объекты и ООП настолько присущи ActionScript, что мы уже пользовались ими, возможно, не понимая этого. Знакомым объектом Flash является клип, реализованный 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 именами методов и свойств, а один экземпляр класса отличается от другого *значениями* этих свойств.

Независимо от того, какие объекты используются – собственные или встроенные в ActionScript, ООП четко отделяет компоненты программы один от другого (*инкапсулирует*) и позволяет им взаимодействовать между собой, не зная подробностей устройства друг друга. Это позволяет изменять внутреннее функционирование объекта без отрицательного воздействия на те части программы, которые используют объект, если только не меняются методы объекта (т. е. его интерфейсы на внешний мир). Если вернуться, например, к нашему объекту `ball`, нас не интересует то, что законы физики изменяют характер движения нашего шарика. Мы лишь вызываем метод *move( )*, а о деталях пусть заботится сам объект.

Другой приятной особенностью ООП является возможность обращаться с *разными* объектами, обладающими *разным поведением*, единообразно, если в них реализованы методы с одинаковыми именами. Допустим, например, что есть объекты `circle` (круг) и `square` (квадрат). Если в обоих объектах реализован метод *area( )*, возвращающий площадь фигуры, то можно вызывать их методы *area( )*, не беспокоясь о том, как каждый из объектов вычисляет свою площадь.

В этой главе мы научимся создавать базовые объекты и определять категории объектов (т. е. классы). Освоив основы, мы узнаем, как использовать в классах и объектах общие характеристики (т. е. создавать родословное дерево) с помощью наследования. 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 и теплокровность. Наконец, мы узнаем, как с помощью ООП управлять средой Flash посредством встроенных объектов и классов ActionScript.

Надеюсь, что это введение пролило некоторый свет на объекты и ООП. Займемся глубже спецификой.

# Анатомия объекта

Как и массив, отдельный объект служит контейнером для контейнеров. Массив содержит несколько значений данных в отдельных *элементах*; аналогично этому, объект содержит несколько значений данных в отдельных *свойствах*. Однако свойства объекта *именуются*, а не нумеруются. Массив хранит группу элементов в нумерованном списке, а объект хранит свойства в соответствии с уникальными идентификаторами, которые никак не упорядочены. Для доступа к элементу массива нужно знать номер его позиции, а для доступа к свойству объекта требуется знать его имя (т. е. идентификатор).

На рис. 12.1 изображены свойства примера объекта с именем `ball`. У объекта `ball` есть два свойства: `radius` и `color`. Значениями этих свойств являются соответственно 50 и 0xFF0000 (шестнадцатеричное значение красного цвета). Свойства названы уникальными идентификаторами, как переменные. Хотя у каждого свойства есть собственное имя, все они хранятся в едином охватывающем их объекте, `ball`.

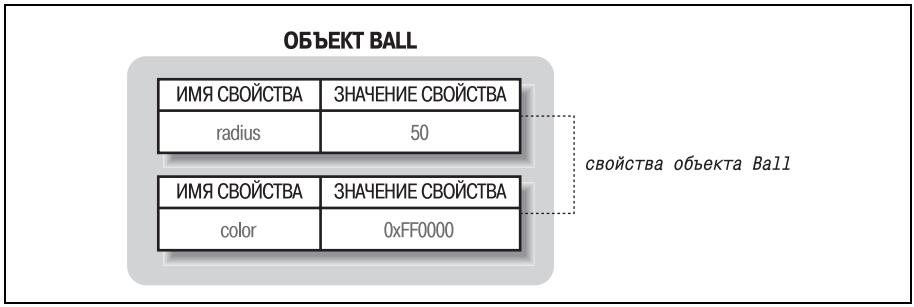


***Рис. 12.1.*** *Пример структуры объекта*

Очевидно, что объект обычно определяет свойства, которые связаны с ним по смыслу. Более конкретно, свойства объекта нужно выбирать так, чтобы с их помощью можно было отличить один объект от другого. Объекты клипов, к примеру, обладают специфическими для клипов свойствами, такими как число кадров в них (`_totalframes`) и положение (`_x` и `_y`).

Так как свойства объектов именуются, а не нумеруются, для объектов не существует таких средств управления элементами, какие есть в массивах (*shift( )*, *unshift( )*, *push( )*, *splice( )* и т. д.). Свойства объектов принято задавать с помощью методов объекта, чтобы не нарушать его инкапсуляции. Это означает, что в строгом ООП объект должен устанавливать свои собственные свойства. Если внешняя сущность хочет задать свойство объекта, это должно быть сделано путем вызова соответствующего метода объекта. Например, пуристу не понравилось бы непосредственное присваивание значения свойству массива `length`. Он стал бы доказывать, что лучше всего дать объекту Array возмож-

ность самому задавать свойство length, а код, находящийся вне объекта, должен делать это только косвенно путем вызова одного из методов объекта Array. В этом случае тот, кто сопровождает класс Array, может поменять имя свойства length на len, что никак не отразится на других пользователях.

# Создание экземпляров объектов

Пусть это покажется откатом назад, но допустим, что класс объектов уже создан. Это не слишком нелепое предположение, 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 предоставляет класс объектов *Array*, но создание конкретных массивов предоставляется пользователю.

Для создания экземпляра объекта используется оператор *new* и *функция конструктора*, с помощью которой инициализируется объект. Общий синтаксис при этом следующий:

```
new ConstructorFunction()
```

Создадим экземпляр нашего первого объекта с помощью конструктора, который уже встроен в ActionScript – конструктора *Object( )*. Конструктор *Object( )* создает самый общий объект (поэтому он так и назван), базовый тип объекта для всех других типов объектов. Приведенный ниже код создает новый родовой объект:

```
new Object()
```

При создании экземпляра объекта полученный экземпляр обычно сохраняется в переменной, элементе массива или свойстве объекта, чтобы в дальнейшем к нему можно было обращаться. Например:

```
var myObject = new Object();
```

При создании объекта таким способом получается пустой объект без свойств и только с двумя самыми общими методами. Родовые объекты, следовательно, имеют ограниченную область применения: действительная мощь объектов проявляется тогда, когда создаются специализированные классы объектов. Прежде чем научиться создавать собственные классы, посмотрим, как использовать свойства и методы объектов существующих классов.

# Свойства объектов

Свойства являются именованными контейнерами данных, связанными с объектом. Они определяются классом объекта и задаются индиви-

дуально для каждого экземпляра объекта. Как и переменные, свойства объектов могут содержать данные любого вида: строки, числа, булевы величины, null, undefined, функции, массивы, клипы фильмов и даже другие объекты.

## Обращение к свойствам

Доступ к свойствам объекта предоставляет знакомый нам уже оператор точки. Точка отделяет имя свойства от объекта, которому оно принадлежит, например:

```
objectName.propertyName
```

где *objectName* является именем объекта, а *propertyName* – допустимый идентификатор, соответствующий имени некоторого свойства *objectName*.

Например, если есть экземпляр объекта ball со свойством radius, обратиться к radius можно таким образом:

```
ball.radius
```

В альтернативном способе обращения к свойству применяется оператор [ ], например:

```
objectName[propertyName]
```

Оператор ‘‘‘‘‘‘‘ позволяет образовать имя свойства с помощью любого выражения, значением которого является строка, например:

```
trace(ball["radius"]);

var prop = "radius";
trace(ball[prop]);  // prop разрешается в "radius"
```

Доступ ко встроенным свойствам ActionScript осуществляется точно таким же способом. Вспомните синтаксис получения значения числа «пи»:

```
Math.PI
```

В этом выражении происходит обращение к встроенному свойству PI объекта Math. Однако в чистом ООП почти никогда к свойству объекта не обращаются напрямую, вместо этого используются методы. Например, для получения свойства volume экземпляра встроенного класса *Sound* пишут:

```
trace(mySound.getVolume());
```

а не:

```
trace(mySound.volume);
```

## Использование цикла for–in для доступа к свойствам объекта

В главе 8 «Предложения цикла» было показано, как цикл *for-in* может использоваться для перечисления свойств объекта. Сейчас, когда об объектах стало известно несколько больше, стоит на короткое время вернуться к предложению *for-in* и снова рассмотреть его использование для работы со свойствами объекта.

Подобно всем циклам, предложение *for-in* состоит из заголовка и тела. Тело предложения *for-in* автоматически выполняется один раз для каждого свойства указанного объекта. Не нужно знать количество свойств или их имена, поскольку при каждой итерации цикла переменная «итератора» автоматически становится равной имени очередного свойства. Поэтому можно обращаться к свойствам объекта так:

```
// Перечислить свойства объекта ball
for (var prop in ball) {
  trace("Свойство " + prop + " имеет значение " + 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 используемый для доступа к нумерованным элементам массива, с циклом *for-in*, используемым для доступа к свойствам объекта. Сведения о циклах *for-in* можно найти в главе 8.

# Методы

Методы – это функции, связанные с объектом и обычно используемые для выполнения задач или доступа к данным объекта. Методы вызываются с помощью оператора вызова функций, **()**, при этом имя метода отделяется от имени его объекта с помощью оператора точки:

```
objectName.methodName()
```

Например:

```
ball.getArea();  // Вызвать метод getArea() объекта ball
```

Как и свойства, методы определяются для класса и вызываются для отдельных экземпляров объектов. Однако, прежде чем рассмотреть создание методов в классе, подчинимся правилам хорошего объектно-ориентированного программирования и прикрепим метод к отдельному экземпляру встроенного класса *Object*. Поняв, как один метод действует с изолированным объектом, мы правильно применим идею к собственному классу.

Метод, в сущности, является функцией, хранимой в свойстве объекта. В результате присваивания функции свойству объекта она становится методом этого объекта:

```
// Создать объект
myObject = new Object();

// Создать функцию
function greet () {
  trace("hello world");
}

// теперь присвоить значение greet свойству sayHello
myObject.sayHello = greet;
```

Когда функция помещена в свойство объекта, можно вызывать ее как метод, например:

```
myObject.sayHello();  // Выводит: "hello world"
```

По правде говоря, можно присвоить функцию и элементу массива (а не свойству объекта) и вызывать ее с помощью оператора вызова, вот так:

```
var myList = new Array();
myList[0] = greet;
myList[0]();          // Выводит: "hello world"
```

Но когда функция вызывается в качестве *метода* объекта, с ней происходит нечто особенное: она обретает способность извлекать или задавать свойства объекта, к которому прикреплена. Посмотрим, как это действует с нашим свойством `sayHello`. Во-первых, добавим свойство `msg` к нашему родовому объекту (и снова в показательных целях мы повинуемся правилам ООП... Считается дурным тоном прикреплять свое свойство к отдельному экземпляру объекта):

```
myObject.msg = "Nice day, isn't it?";
```

Теперь подправим *greet( )*, чтобы выводилось значение `msg`:

```
function greet () {
  trace(this.msg);
}

// Теперь снова вызовем sayHello
myObject.sayHello();  // Выводит: "Nice day, isn't it?"
```

Обратили внимание на ключевое слово `this` в конце строки 2? Оно соединяет нас с объектом `myObject`. При выполнении *sayHello( )* в качестве метода объекта `myObject`, в этот метод фактически передается невидимый аргумент с именем `this`, который содержит ссылку на `myObject`. В теле функции `this` используется для доступа к свойству `msg` объекта `myObject` в виде `this.msg`. При вызове *myObject.sayHello( )* выражение

this.msg разрешается в myObject.msg и становится таким: «Nice day, isn't it?».

Методы могут как извлекать, так и задавать значения свойств содержащего их объекта. В примере 12.1 показан метод, который извлекает значения двух свойств объекта, выполняет над ними вычисления, а затем задает значение нового третьего свойства. (И снова, поскольку в центре нашего внимания находятся методы, используемый нами объект является всего лишь экземпляром объекта. В следующем разделе мы узнаем, как правильно добавлять новые методы объектов через класс.)

*Пример 12.1. Реализация метода, задающего свойство*

```
// Создать новый объект и сохранить его в rectangle
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;

// Определить функцию для вычисления площади прямоугольника и задать соответст-
// вующее свойство. Обратите внимание на использование ключевого слова this.
function rectArea() {
  this.area = this.width * this.height;
}

// Назначить функцию методом объекта rectangle
rectangle.setArea = rectArea;

// Теперь вызвать метод setArea() объекта rectangle
rectangle.setArea();    // Присваивает площади прямоугольника значение 50

// Наконец, посмотрим, какое новое свойство создал setArea()
trace(rectangle.area);  // Выводит: 50
```

Чаще метод не задает значение свойства, а возвращает значение. Переделаем метод *setArea( )* объекта rectangle, чтобы он так и поступал – см. пример 12.2.

*Пример 12.2. Возвращение методом значения*

```
// Создать и настроить прямоугольник. Обратите внимание на недопустимость
// присваивания метода area до появления в коде функции rectArea
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;
rectangle.area = rectArea;

// На этот раз просто возвратить площадь, не присваивая ее свойству
function rectArea() {
  return this.width * this.height;
}

// Теперь ее даже проще использовать
trace("Площадь прямоугольника равна: " + rectangle.area());
```

```
// Выводит: "Площадь прямоугольника равна: 50"
```

Ключевое слово this – хитрая вещь, поскольку требует от интерпретатора некоторой ловкости рук. Чтобы проиллюстрировать, как работает предыдущий код, рассмотрим пример 12.3, выявляющий скрытую связь между this и его объектом и методом. (Пример 12.3 вымышлен, так как this является ключевым словом и при использовании в таком виде вызовет ошибку.)

*Пример 12.3. Фиктивный пример, явным образом использующий this*

```
// Создать и настроить объект прямоугольника
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;
rectangle.area = rectArea;

// Специально потребуем this в качестве аргумента функции rectArea()
function rectArea(this) {
  return this.width * this.height;
}

// Вызов метода с явной ссылкой на объект rectangle
trace("The area of rectangle is: " + rectangle.area(rectangle));
```

Методы могут также присваиваться свойствам объектов в виде литералов функций. Использование литерала функции для создания метода избавляет от необходимости сначала создавать функцию, а затем присваивать ее свойству. В примере 12.4 объект примера 12.2 переработан так, чтобы показать использование литерала функции.

*Пример 12.4. Реализация методов с помощью литералов функций*

```
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;

// Далее идет присваивание литерала функции
rectangle.area = function () {
  return this.width * this.height;
};  // Рекомендуется использовать точку с запятой в конце литерала
    //  функции

// Вызывать функцию можно, как и прежде
trace("The area of rectangle is: " + rectangle.area());
```

Методы предоставляют огромные возможности для контроля над объектами. Использование методов со встроенными объектами является одним из главных средств управления средой фильмов Flash.

В этом разделе мы познакомились с тем, как записать функцию в свойство (тем самым сделав ее методом) и как вызывать функцию с по-

мощью свойства. У этих двух действий схожий синтаксис, но совершенно разные результаты. Очень важно понимать различие между ними.

Во время присваивания метода свойству круглые скобки должны быть *опущены*:

```
myObj.myMethod = someFunction;
```

При вызове метода круглые скобки должны присутствовать:

```
myObj.myMethod();
```

Различие в синтаксисе тонкое, так что следите за его соблюдением.

# 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 нехороших субъектов, обладающих искусственным интеллектом, в видеоигре. Для облегчения создания объектов и задания иерархий объектов используются классы объектов. Класс служит определением в виде шаблона для целой категории объектов. Как говорилось во введении, классы описывают родовые особенности определенных пород объектов, типа наличия у всех собак четырех ног.

## Классы объектов

Прежде чем рассматривать, как используются классы, посмотрим, как обстоят дела, если их не использовать. Допустим, нужен объект `ball`, но вместо того чтобы создать его с помощью класса, мы просто попробуем приспособить для этого общий объект из встроенного класса *Object*. Дадим этому объекту свойства: `radius`, `color`, `xPosition`, и `yPosition`. Затем добавим два метода – *moveTo( )* и *area( )* – для перемещения объекта и определения размера занимаемой им площади.

Воспользуемся таким кодом:

```
var ball = new Object();
ball.radius = 10;
ball.color = 0xFF0000;
ball.xPosition = 59;
ball.yPosition = 15;
ball.moveTo = function (x, y) { this.xPosition = x; this.yPosition = y; };
ball.area = function () { return Math.PI * (this.radius * this.radius); };
```

Этот подход приводит к нужной цели, но имеет ограничения: каждый раз, когда требуется новый объект шарика, приходится повторять

весь код инициализации для ball, что утомительно и чревато ошибками. Кроме того, при создании таким способом многочисленных объектов шариков будет происходить избыточное дублирование одинакового кода функций *moveTo( )* и *area( )* для каждого объекта, что приведет к ненужному расходу памяти.

Для эффективного создания ряда объектов с общими свойствами следует создать класс. С помощью класса можно определить свойства, которыми должны обладать все объекты ball. Кроме того, все свойства с фиксированными значениями можно сделать общими для всех экземпляров класса ball. На обычном языке класс *Ball* можно описать интерпретатору так:

> *Ball* является типом объекта. У всех экземпляров объекта ball есть свойства radius, color, xPosition, yPosition, которые задаются отдельно для каждого ball. Все объекты ball также используют совместно методы *moveTo( )* и *area( )*, одинаковые для всех членов класса *Ball*.

Посмотрим, как эта теория действует на практике.

## Создание класса

В ActionScript не существует специального средства для «объявления класса»: нет предложения «class», которое создавало бы новый класс подобно предложению *var*, создающему новые переменные. Вместо этого определяется 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 обозначения того, что это конструктор класса, принято писать его с заглавной буквы. Имя конструктора должно описывать класс создаваемых им объектов, например, *Ball*, *Product* или *Vector2d*. Содержащиеся в конструкторе *statements* инициализируют создаваемые им объекты.

Для начала сделаем функцию конструктора *Ball* как можно более простой. Пока нам нужно только чтобы она создавала пустые объекты:

```
// Сделать Ball конструктором
function Ball () {
  // Здесь что-то выполняется
}
```

Это было не трудно. Теперь посмотрим, как с помощью функции конструктора *Ball* создавать объекты `ball`.

## Создание членов класса

Функция конструктора определяет класс и используется для создания новых экземпляров класса. Как отмечалось выше, при вызове 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 class
```

Поэтому для создания объекта `ball` (*экземпляра*) с помощью класса *Ball* пишут:

```
myBall = new Ball();  // Сохраняет экземпляр класса Ball в myBall
```

Класс *Ball* пока не добавляет свойств или методов к создаваемым им объектам. Этим мы теперь и займемся.

## Присвоение новых свойств объектам класса

Для создания новых свойств на этапе создания объекта снова обратимся к специальному ключевому слову `this`. Внутри функции конструктора в ключевом слове `this` хранится ссылка на создаваемый в данный момент объект. С помощью этой ссылки можно присваивать зарождающемуся объекту любые нужные свойства. Общая технология этого описывается следующим синтаксисом:

```
function Constructor () {
  this.propertyName = value;
}
```

Здесь `this` указывает на создаваемый объект, *propertyName* является свойством, которое нужно прикрепить к объекту, а *value* является значением данных, которое присваивается этому свойству.

Применим эту технику в нашем примере с *Ball*. Ранее предлагалось сделать свойствами класса *Ball* `radius`, `color`, `xPosition` и `yPosition`. Вот как конструктор класса *Ball* может присваивать эти свойства своим экземплярам (обратите внимание на использование ключевого слова `this`):

```
function Ball () {
  this.radius = 10;
  this.color = 0xFF0000;
  this.xPosition = 59;
  this.yPosition = 15;
}
```

С помощью такого конструктора *Ball* можно создавать экземпляры объектов (т. е. члены класса) с предопределенными свойствами – `radius`,

color, xPosition и yPosition — путем вызова *Ball( )* с оператором *new*, так же, как и ранее. Например:

```
// Создать новый экземпляр Ball
bouncyBall = new Ball();

// Теперь выведем значения свойств bouncyBall, присвоенные
// при создании его конструктором Ball().
trace(bouncyBall.radius);     // Выводит: 10
trace(bouncyBall.color);      // Выводит: 16711680
trace(bouncyBall.xPosition);  // Выводит: 59
trace(bouncyBall.yPosition);  // Выводит: 15
```

Ну, не здорово ли?

К сожалению, конструктор *Ball( )* использует фиксированные величины, присваивая значения свойствам создаваемых им объектов (например, this.radius = 10). Поэтому у всех объектов класса *Ball* будут одинаковые значения свойств, что прямо противоположно задачам объектно-ориентированного программирования (нет никакой надобности в классе, который порождает одинаковые объекты: именно различия делают их интересными).

Чтобы динамически присваивать значения свойств экземплярам класса, изменим функцию конструктора так, чтобы она принимала аргументы. Рассмотрим общий синтаксис, а затем вернемся к примеру класса *Ball*:

```
function Constructor (value1, value2, value3) {
  this.property1 = 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 нужно создать новый уникальный член класса, функции конструктора передаются начальные значения свойств:

```
myObject = new Constructor (value1, value2, value3);
```

Посмотрим на примере 12.5, как применить это к классу *Ball*.

*Пример 12.5. Обобщенный класс Ball*

```
// Пусть конструктор Ball() принимает в качестве аргументов значения свойств
function Ball (radius, color, xPosition, yPosition) {
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
```

```
  this.yPosition = yPosition;
}

// Вызвать конструктор с передачей ему аргументов, используемых
// в качестве значений свойств объекта
myBall = new Ball(10, 0x00FF00, 59, 15);

// Посмотрим, получилось ли...
trace(myBall.radius);  // Выводит: 10,  :) Здорово...
```

Создание класса *Ball* почти завершено. Но можно заметить, что все еще отсутствуют методы *moveTo( )* и *area( )*, о которых говорилось ранее. На практике есть два способа прикрепления методов к объектам класса. Сейчас будет рассмотрен простой и менее эффективный способ, а после рассмотрения темы наследования мы вернемся к вопросу о создании методов.

## Присвоение методов объектам класса

Проще всего добавить в класс метод, присвоив классу в конструкторе свойство, которое содержит функцию. Вот общий синтаксис:

```
function Constructor() {
  this.methodName = function;
}
```

*function* можно предоставить несколькими способами, что мы изучим на примере добавления метода *area( )* в наш класс *Ball*. Обратите внимание, что для лучшей ясности мы убрали из класса свойства `color`, `xPosition` и `yPosition`.

*function* может являться литералом функции, например:

```
function Ball (radius) {
  this.radius = radius;
  // Добавить метод area...
  this.area = function () { return Math.PI * this.radius * this.radius; };
}
```

В другом варианте *function* можно объявить *внутри* конструктора:

```
function Ball (radius) {
  this.radius = radius;
  // Добавить метод area...
  this.area = getArea;
  function getArea () {
    return Math.PI * this.radius * this.radius;
  }
}
```

Наконец, *function* можно объявить вне конструктора, но присвоить свойству внутри конструктора:

```
// Объявить функцию getArea()
function getArea () {
  return Math.PI * this.radius * this.radius;
}

function Ball (radius) {
  this.radius = radius;
  // Добавить метод area...
  this.area = getArea;
}
```

Существенной разницы между этими тремя подходами нет: все они совершенно допустимы. Обычно удобнее всего использовать литералы функций, но они не так способствуют многократному использованию кода, как функции, определяемые вне конструктора, благодаря чему их могут использовать другие конструкторы. Тем не менее, все три способа недостаточно эффективны.

До сих пор мы прикрепляли уникальные значения свойств каждому объекту класса *Ball*. У каждого объекта `ball` должны быть собственные значения свойств `radius` и `color`, чтобы один объект `ball` можно было отличить от другого. Но если с помощью описанной технологии присваивать объектам класса фиксированный метод, возникнет ненужное дублирование метода в каждом объекте класса. Формула *area (площади)* одна и та же для каждого `ball`, поэтому код, выполняющий данную задачу, должен быть централизован и обобщен.

Для более эффективного прикрепления к классу свойства или метода с фиксированным значением используется наследование, которое будет следующей изучаемой нами темой.

## Наследование свойств объектов

*Наследуемые* свойства не прикрепляются к отдельным экземплярам объектов класса. Они прикрепляются один раз к конструктору класса и при необходимости заимствуются объектами. Эти свойства называют наследуемыми, потому что они передаются создаваемым объектам, а не определяются в каждом из них. Наследуемое свойство кажется принадлежащим объекту, через который к нему обращаются, но в действительности входит в конструктор класса.

На рис. 12.2 показана общая модель наследования свойств на примере класса *Ball*. Так как методы *moveTo( )* и *area( )* класса *Ball* одинаковы для всех его экземпляров, лучше всего реализовать их в виде наследуемых методов. Они принадлежат самому классу, и каждый объект `ball` обращается к ним только по ссылке. Напротив, свойства `radius`, `color`, `xPosition` и `yPosition` класса *Ball* присваиваются объектам как обычные свойства, потому что каждому объекту `ball` требуются собственные значения этих свойств.

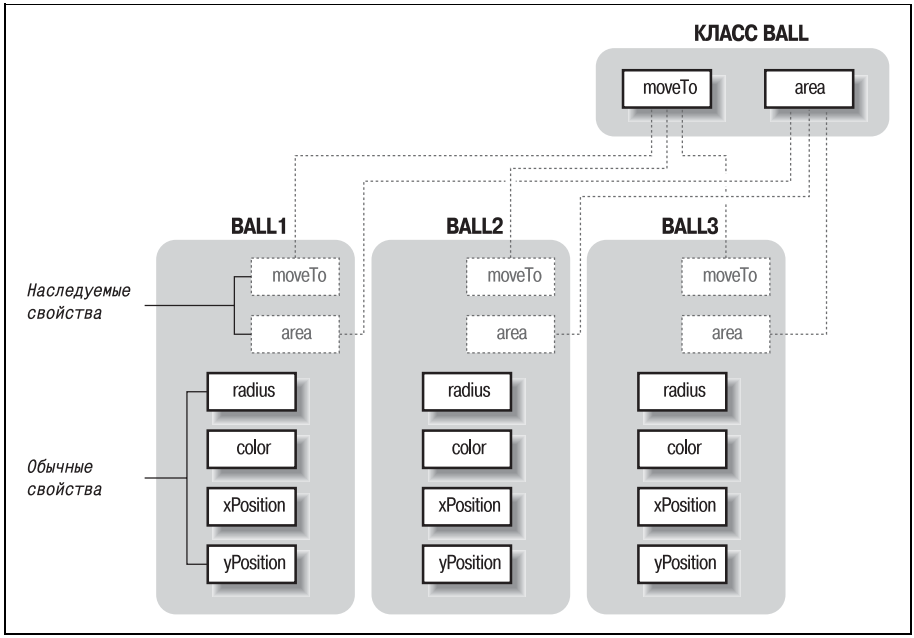


***Рис. 12.2.** Наследуемые и обычные свойства*

Наследование происходит в иерархической цепочке подобно родословному дереву. При вызове метода отдельного объекта интерпретатор проверяет, реализует ли данный объект этот метод. Если метод не найден, интерпретатор ищет метод в классе.

Например, при выполнении метода `ball1.area()` интерпретатор проверяет, определен ли в `ball1` метод *area( )*. Если в объекте `ball1` метод *area( )* отсутствует, интерпретатор проверяет, определен ли метод *area( )* в классе *Ball*. Если да, то интерпретатор вызывает *area( )*, как если бы это был метод объекта `ball1`, а не класса *Ball*. Это позволяет методу действовать над объектом `ball1` (а не классом), при необходимости извлекая или задавая свойства `ball1`. В этом состоит одно из преимуществ ООП: определив функцию в одном месте (классе *Ball*), можно пользоваться ей во многих местах (любом объекте `ball`).

В отличие от обычных свойств, унаследованные свойства могут только извлекаться через объект, но не задаваться.

Пора привести маленький код, чтобы оживить все эти понятия.

## Создание наследуемых свойств с помощью свойства prototype

Начнем процесс создания наследуемых свойств с создания функции конструктора класса, такой, как у класса *Ball* из примера 12.5:

```
function Ball (radius, color, xPosition, yPosition) {
```

```
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}
```

Вспомните из главы 9 «Функции», что функции могут рассматриваться как объекты и поэтому иметь свойства.

При создании функции конструктора интерпретатор автоматически задает ей свойство с именем `prototype`. В свойство `prototype` интерпретатор помещает родовой объект. Все свойства, прикрепляемые к объекту `prototype`, наследуются всеми экземплярами класса функции конструктора.

Чтобы создать свойство, которое будет унаследовано всеми объектами класса, это свойство нужно просто присвоить готовому объекту `prototype` функции конструктора класса. Вот общий синтаксис для этого:

```
Constructor.prototype.propName = value;
```

Здесь *`Constructor`* является функцией конструктора класса (в данном примере, *Ball*); `prototype` является автоматически создаваемым свойством, используемым для хранения наследуемых свойств; *`propName`* является именем наследуемого свойства, а *`value`* является значением этого наследуемого свойства. Например, вот как можно добавить глобальное свойство `gravity` (сила тяжести) ко всему классу объектов *Ball*:

```
Ball.prototype.gravity = 9.8;
```

После размещения свойства `gravity` можно обращаться к нему из любого члена класса *Ball*:

```
// Создать новый экземпляр класса Ball
myBall = new Ball(5, 0x003300, 34, 220);

// Теперь вывести значение унаследованного свойства gravity
trace(myBall.gravity);  // Displays: 9.8
```

К свойству `gravity` можно обращаться через `myBall`, потому что `myBall` наследует свойства объекта `prototype` класса *Ball*.

Поскольку одни и те же методы обычно используются всеми экземплярами класса, их часто хранят в наследуемых свойствах. Добавим наследуемый метод *area( )* в класс *Ball*:

```
Ball.prototype.area = function () {
  return Math.PI * this.radius * this.radius;
};  // Точка с запятой обязательны, поскольку это литерал функции
```

Это было здорово; теперь давайте добавим наследуемый метод *moveTo( )*, но на этот раз не с помощью литерала функции, а с помощью предопределенной функции:

```
function moveTo (x, y) {
  this.xPosition = x;
  this.yPosition = y;
}

Ball.prototype.moveTo = moveTo;
```

Если функция определена как наследуемое свойство, ее можно вызывать, как и всякий другой метод:

```
// Создать новый ball
myBall = new Ball(15, 0x33FFCC, 100, 50);

// Теперь вызвать унаследованный метод area() объекта myBall
trace(myBall.area());  // Выводит: 706.858347057703
```

Обратите внимание, что объект конструктора `prototype` можно целиком заменить новым объектом и в результате одним махом добавить много наследуемых свойств. Однако при этом изменится цепочка наследования, о чем будет рассказано ниже в разделе «Надклассы и подклассы».

## Замена унаследованных свойств

Чтобы изменить унаследованное свойство для одного объекта, можно назвать свойство этого объекта тем же именем, что и унаследованное свойство. Например, можно задать для одного шарика силу тяжести меньше, чем для других шариков:

```
// Создадим конструктор класса Ball
function Ball (radius, color, xPosition, yPosition )
  { ... } // Не показана

// Присвоение унаследованному свойству gravity
Ball.prototype.gravity = 9.8

// Создать объект Ball
lowGravBall = new Ball ( 200, 0x22DD99, 35, 100 );

// Заменить унаследованное свойство gravity
lowGravBall.gravity = 4.5;
```

Задаваемое для объекта свойство всегда заменяет унаследованное свойство с тем же именем. Это просто следствие характера действия цепочки наследования. Аналогично, определения методов, являющиеся локальными для объекта, заменяют имеющиеся в его классе. Если мы выполняем `ball1.area()`, и `ball1` определяет метод *area( )*, выпол-

няется последний метод: интерпретатор не станет смотреть prototype класса *Ball*, чтобы узнать, не определен ли и там метод *area( )*.

## Свойство constructor

Когда создается объект конструктора prototype, интерпретатор автоматически присваивает ему особое свойство с именем constructor. Свойство constructor является ссылкой на функцию конструктора класса prototype. Например, оба следующие выражения дают ссылку на функцию конструктора *Ball*:

```
trace(Ball);                          // Выводит: [Function]
trace(Ball.prototype.constructor);    // Тоже выводит: [Function]
                                      // Такая же ссылка, как первая
```

Обратите внимание, что свойство constructor содержит ссылку на функцию конструктора, а не на строку, представляющую имя функции.

## Свойство __proto__

При создании любого объекта интерпретатор автоматически присваивает ему особое свойство с именем __proto__ (обратите внимание на два символа подчеркивания с каждой стороны имени). Свойство объекта __proto__ является ссылкой на свойство prototype функции конструктора этого объекта. Например, при создании экземпляра *Ball* с именем myBall, myBall.__proto__ становится равным Ball.prototype:

```
myBall = new Ball(6, 0x00FF00, 145, 200);
trace(myBall.__proto__ == Ball.prototype);  // Выводит: true
```

Свойство __proto__ используется, главным образом, интерпретатором ActionScript для нахождения унаследованных свойств объекта. Например, при вызове унаследованного метода *area( )* через myBall, как в *myBall.area( )*, ActionScript обращается к методу *Ball.prototype.area* через myBall.__proto__.

Можно использовать __proto__ непосредственно, чтобы проверить принадлежность объекта определенному классу, как в примере 12.6.

*Пример 12.6. Определение класса объекта*

```
function MyClass (prop) {
  this.prop = prop;
}
myObj = new MyClass();
if (myObj.__proto__ == MyClass.prototype) {
  trace("myObj is an instance of MyClass");
}
```

# Надклассы и подклассы

Одной из важнейших характеристик классов в продвинутом объектно-ориентированном программировании является их способность совместно использовать свойства. Это означает, что целые классы могут наследовать свойства от одних классов и передавать их другим. В сложных ситуациях способность к множественному наследованию может оказаться необходимой. (Даже если сами вы не станете использовать множественное наследование, понимание его поможет в работе со встроенными классами ActionScript.)

Уже было показано, как объекты наследуют свойства объекта `prototype` конструктора своего класса. Наследование не ограничивается лишь одной связью объект/класс. Сам класс может наследовать свои свойства от других классов. Например, может иметься класс *Circle* (круг), определяющий общий метод *area( )* определения площади всех круглых объектов. Вместо определения отдельного метода *area( )* в классе *Ball* можно просто заставить *Ball* наследовать метод *area( )*, уже присутствующий в *Circle*. Поэтому экземпляры *Ball* наследуют имеющийся в *Circle* метод *area( )* через *Ball*. Обратите внимание на иерархию: простейший класс, *Circle*, определяет самые общие методы и свойства. Другой класс, *Ball*, основывается на этом простом классе *Circle*, добавляя специфические для экземпляров класса *Ball* особенности, но беря от *Circle* базовые атрибуты, имеющиеся у всех круглых объектов. В обычном объектно-ориентированном программировании говорят, что класс *Ball расширяет* класс *Circle*. Это означает, что *Ball* является *подклассом Circle*, а *Circle* является *надклассом Ball*.

## Создание надкласса

Выше мы научились определять наследуемые свойства объекта `prototype` функции конструктора класса. Чтобы создать *надкласс* для заданного класса, полностью *заменим* объект `prototype` класса нужным экземпляром требуемого надкласса. Вот общий синтаксис:

```
Constructor.prototype = new SuperClass();
```

Вследствие замены объекта `prototype` функции *Constructor* экземпляром *SuperClass* все экземпляры *Constructor* вынуждены наследовать те свойства, которые определены в экземплярах *SuperClass*. В примере 12.7 сначала создается класс *Circle*, который присваивает метод *area( )* всем своим экземплярам. Затем экземпляр *Circle* присваивается `Ball.prototype`, в результате чего все объекты `ball` наследуют *area( )* из *Circle*.

*Пример 12.7. Создание надкласса*

```
// Создать конструктор Circle (надкласса)
function Circle() {
  this.area = 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;
}

// Здесь создается надкласс путем присваивания экземпляра Circle
// свойству prototype конструктора класса Ball
Ball.prototype = new Circle();

// Создадим экземпляр Ball и проверим его свойства
myBall = new Ball ( 16, 0x445599, 34, 5);
trace(myBall.xPosition);   // 34, обычное свойство Ball
trace(myBall.area());      // 804.24..., area() унаследован от Circle
```

Однако у нашей иерархии классов довольно бедная структура: *Ball* определяет `radius`, но `radius` фактически является общим свойством для всех окружностей, поэтому оно должно быть в классе *Circle*. То же 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.)

Идейно схему пересмотренных конструкторов *Circle* и *Ball* можно представить следующим образом:

```
// Создать конструктор Circle (надкласса)
function Circle ( radius, xPosition, yPosition ) {
  this.area = function () { return Math.PI * this.radius * this.radius; };
  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*? Необходимо сделать еще одно изменение в коде конструктора *Ball*. Во-первых, заставим *Ball* принимать все необходимые свойства в виде параметров:

```
function Ball ( color, radius, xPosition, yPosition ) {
```

Затем в конструкторе *Ball* определим конструктор *Circle* как метод создаваемого объекта `ball`:

```
  this.superClass = Circle;
```

Наконец, вызовем конструктор *Circle* с объектом `ball` и передадим ему значения свойств `radius`, `xPosition` и `yPosition`:

```
this.superClass(radius, xPosition, yPosition);
```

Полный код класса/надкласса показан в примере 12.8.

*Пример 12.8. Класс и его надкласс*

```
// Создать конструктор Circle (надкласса)
function Circle ( radius, xPosition, yPosition) {
  this.area = function () { return Math.PI * this.radius * this.radius; };
  this.radius = radius;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Создать конструктор класса Ball
function Ball ( color, radius, xPosition, yPosition ) {
  // Определить надкласс Circle как метод создаваемого объекта шарика
  this.superClass = Circle;
  // Вызвать конструктор Circle как метод объекта шарика, передав ему
  // значения, полученные как аргументы конструктора Ball
  this.superClass(radius, xPosition, yPosition);
  // Задать цвет шарика
  this.color = color;
}

// Присвоить экземпляр надкласса Circle свойству prototype конструктора
// класса Ball
Ball.prototype = new Circle();

// Теперь создадим экземпляр Ball и проверим его свойства
myBall = new Ball ( 0x445599, 16, 34, 5);
trace(myBall.xPosition);   // 34
trace(myBall.area());      // 804.24...
trace(myBall.color);       // 447836
```

Заметьте, что слово `superClass` в *Ball* не является зарезервированным или специальным. Это просто удобное имя для функции конструктора надкласса. Кроме того, метод *area( )* класса *Circle* можно было бы определить в `Circle.prototype`. Начав практически программировать с помощью классов и объектов, вы несомненно заметите определенную гибкость, которую предоставляют инструменты ActionScript при создании иерархий классов и реализации наследования. Возможно, потребуется адаптировать описанные в данной главе подходы, чтобы они наилучшим образом соответствовали тонким особенностям конкретного приложения.

## Полиморфизм

Наследование делает возможным другое ключевое понятие ООП – *полиморфизм*. Слово «полиморфизм» означает «множество форм». Это значит, что объекту сообщается, что он должен сделать, но не говорится как. Лучше всего проиллюстрировать это примером. Допустим, разрабатывается игра про полицейских и грабителей. Одновременно на экране рисуется много 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 этих лиц создается класс объектов:

```
function Cop() { ... }
function Robber() { ... }
function Bystander() { ... }
```

Кроме того, создается надкласс *Person*, наследниками которого являются 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 двигаться согласно правилам для своего класса. Для этого определим в каждом объекте метод *move( )* (метод *move( )* настраивается для каждого класса):

```
Person.prototype.move = function () { ... движение по умолчанию ... }
Cop.prototype.move = function () { ... движение в преследовании ... }
Robber.prototype.move = function () { ... движение при убегании ... }
Bystander.prototype.move = function () { ... случайное движение ... }
```

Требуется, чтобы в каждом кадре Flash все персонажи перемещались по экрану. Для управления всеми действующими лицами создадим главный массив объектов *Person*. Вот пример того, как можно заполнить массив `persons`:

```
// Создать полицейских
cop1 = new Cop();
cop2 = new Cop();

// Создать грабителей
robber1 = new Robber();
robber2 = new Robber();
robber3 = new Robber();

// Создать прохожих
bystander1 = new Bystander();
bystander2 = new Bystander();
```

```
// Создать массив, заполненный полицейскими, грабителями и прохожими
persons = [cop1, cop2, robber1, robber2, robber3, bystander1, bystander2];
```

В каждом кадре фильма Flash вызывается функция *moveAllPersons( )* (переместить всех), которая определена так:

```
function moveAllPersons() {
  for (var i=0; i < persons.length; i++) {
    persons[i].move();
  }
}
```

При вызове *moveAllPersons( )* все полицейские, грабители и прохожие переместятся по отдельным правилам, которые определены в их классах в методах *move( )*. Это и есть полиморфизм в действии: объекты с общими характеристиками можно организовать вместе, но при этом они сохраняют свою индивидуальность. У полицейских, грабителей и прохожих много общего, что воплощено в надклассе *Person*. У них есть общие действия, например, они умеют перемещаться. Однако реализовывать эти общие действия они могут по-разному, и могут иметь другие специфичные для каждого класса действия и данные. Полиморфизм позволяет единообразно обращаться с разнородными объектами. Функция *move( )* применяется для перемещения всех лиц, хотя в каждом классе есть своя функция *move( )*.

## Определение принадлежности объекта к надклассу

Предположим, что имеется класс *Shape* (фигура) и класс *Rectangle* (прямоугольник), для которого *Shape* является надклассом. Чтобы проверить, является ли объект потомком *Shape*, нужно улучшить метод примера 12.6, чтобы он проходил по цепочке объектов прототипов (называемой цепочкой прототипов (*prototype chain*)). Эта техника продемонстрирована в примере 12.9.

*Пример 12.9. 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, является ли myRect наследником класса Shape
trace (objectInClass(myRect, Shape)); // Выводит: true
```

## Окончание цепочки наследования

Все объекты ведут свое происхождение от класса верхнего уровня *Object*. Поэтому все объекты наследуют свойства, определенные в конструкторе *Object*, а именно, методы *toString( )* и *valueOf( )*. Поэтому можно добавить новые свойства к каждому объекту фильма путем введения новых свойств в класс *Object*. Свойства, прикрепленные к `Object.prototype`, будут распространяться по всей иерархии классов, включая все внутренние объекты и объекты клипов фильмов! В результате можно создавать действительно глобальные переменные или методы. Например, в следующем коде ширина и высота рабочего стола определяются `prototype` класса *Object*. После этого данная информация становится доступна в любом клипе:

```
Object.prototype.mainstageWidth = 550;
Object.prototype.mainstageHeight = 400;
trace(anyClip.mainstageWidth);  // Displays: 550
```

Свойства `Object.prototype` наследуются не только клипами, но и всеми объектами, поэтому даже пользовательские объекты и экземпляры других встроенных классов, например, *Date* и *Sound*, наследуют свойства, присваиваемые `Object.prototype`.

Упражнение для читателя. Прикрепить новые свойства и методы можно к любому встроенному классу. Попробуйте прикрепить нечувствительную к регистру функцию сортировки по алфавиту из примера 11.6 к классу *Array* в качестве наследуемого метода.

## Сравнение с терминологией Java

В нашем обзоре классов и иерархий классов мы сталкивались со свойствами, определяемыми для объектов, конструкторов классов и прототипов классов. В Java и C++ существуют специальные имена для различных типов свойств классов и объектов. Для удобства программистов на Java в табл. 12.1 приведены приблизительные эквиваленты между Java и ActionScript.

*Таблица 12.1. Эквивалентные свойства в Java и ActionScript*

| Java | Описание | ActionScript | Пример для ActionScript |
|---|---|---|---|
| Переменные экземпляра | Локальные переменные экземпляра объекта | Свойства, определенные в функции конструктора класса и копируемые в объекты | `function Square (side) {`<br>`  this.side = side;`<br>`}` |
| Методы экземпляра | Методы, вызываемые на экземпляре объекта | Методы, определенные в объекте `prototype` функции конструктора класса, доступ к которым при вызове с объектом автоматически осуществляется через `prototype` | `Square.prototype.area = squareArea;`<br><br>`mySquare.area();` |

*Таблица 12.1 (продолжение)*

| Java | Описание | ActionScript | Пример для ActionScript |
|---|---|---|---|
| Переменные класса | Переменные, имеющие одно и то же значение для всех экземпляров объектов класса | Свойства, определенные как свойства функции для функции конструктора класса | `Square.numSides = 4;` |
| Методы класса | Методы, вызываемые через класс | Методы, определенные как свойства функции для функции конструктора класса | `Square.findSmaller = function (square1, square2) { ... }` |

## Краткое резюме по объектно-ориентированному программированию

Классы и наследование открывают целый мир возможностей совместного использования информации многими объектами. Объектно-ориентированное программирование служит основой всего ActionScript. Независимо от того, в какой мере вы используете возможности классов и объектов в своих сценариях, понимание их общих принципов составляет важную часть понимания среды программирования Flash. В последнем разделе этой главы будет показано, что уверенное владение объектно-ориентированным программированием существенно влияет на уверенность в ActionScript в целом.

Для более глубокого ознакомления с объектно-ориентированным программированием на языках, производных от ECMA-262, обратитесь к документации Netscape по JavaScript, «Details of the Object Model»:

*http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm*

В каноническом руководстве Дэвида Фланагана (David Flanagan) «JavaScript: The Definitive Guide» (O'Reilly & Associates, Inc.) также содержится ценная информация по ООП в JavaScript. Общее введение в ООП с точки зрения Java можно найти в «Object Oriented Programming Concepts» (из «The Java™ Tutorial») фирмы Sun:

*http://java.sun.com/docs/books/tutorial/java/concepts*

# Встроенные классы и объекты ActionScript

Мы действительно далеко ушли вперед после главы 1 «Легкое введение для непрограммистов». Наше первое столкновение с компонента-

ми ActionScript было просто чтением элементов под кнопкой + в панели Actions. С того момента мы узнали о данных и выражениях, операторах, предложениях и функциях, а сейчас изучили 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; *предложения* дают инструкции; *функции* объединяют инструкции в переносимые команды. Это средства, и всего лишь средства. Это грамматика, используемая при написании инструкций в сценариях. Нам все еще не хватает *темы для обсуждения*. Сейчас мы отлично знаем, как говорить на ActionScript, но нам не о чем говорить. Этот пробел заполняют встроенные классы и объекты ActionScript.

## Встроенные классы

Точно так же, как мы определяем свои классы для описания объектов и действий с ними, ActionScript определяет собственные классы данных. Множество готовых классов, включая класс *Object*, встроено непосредственно в язык ActionScript. Встроенные классы могут управлять физической средой фильма Flash.

Например, один из встроенных классов, *Color*, определяет методы, способные определять или задавать цвет клипа. Чтобы использовать эти методы, сначала нужно создать объект *Color* с помощью конструктора класса *Color( )*, например:

```
clipColor = new Color(target);
```

где `clipColor` является переменной, элементом массива или свойством объекта, в которых хранится объект *Color*. Конструктор *Color( )* принимает один аргумент, `target`, который задает имя клипа, цвет которого нужно узнать или задать.

Допустим, есть клип с именем `square`, и нужно задать его цвет. Создадим новый объект *Color*:

```
squareColor = new Color(square);
```

Чтобы задать цвет клипа `square`, вызываем один из методов *Color* с объектом `squareColor`:

```
squareColor.setRGB(0x999999);
```

Метод *setRGB( )* задает значение цвета RGB для `target` объекта *Color*, которым в данном случае является `square`, поэтому предшествующий вызов метода делает цвет `square` серым.

Поскольку объект *Color* является встроенным классом, он может непосредственно задавать цвет клипа. Другие классы позволяют управлять

звуками, датой и документами XML. Все, что касается встроенных классов, будет рассказано в части III «Справочник 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 использования этих свойств и методов не нужно создавать объект *Key*, его можно прямо использовать. Встроенные объекты автоматически создаются интерпретатором при запуске Flash Player и доступны на протяжении всего фильма.

Выведем, например, код нажатой в данный момент клавиши с помощью метода *getCode( )* объекта *Key*:

```
trace(Key.getCode());
```

А теперь проверим, нажата ли клавиша пробела, с помощью метода *isDown( )*, которому передается в качестве аргумента код клавиши пробела:

```
trace(Key.isDown(Key.SPACE));
```

В части III будет рассказано о других встроенных объектах – *Math*, *Mouse* и *Selection*, которые предоставляют доступ к математическим данным, указателю мыши и выделению в текстовых полях.

## 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 части III. Однако необязательно изучать все классы и объекты в один присест. Начните изучение с клипов, а по мере необходимости обращайтесь к другим темам. Со временем вы узнаете, что нужно 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.

# Вперед!

Мы, конечно, прошли большой путь, и я надеюсь, что вы, как и я, не утратили энтузиазма. Следующим мы изучим самый важный из классов объектов, имеющихся в ActionScript – клип.

# 13

# Клипы

В каждом документе Flash содержится рабочий стол, на который помещаются фигуры, текст и другие видимые элементы, а также временная диаграмма, с помощью которой задается изменение содержимого рабочего стола в зависимости от времени. Рабочий стол (т. е. *главный фильм – main movie*) может содержать независимые подфильмы, называемые *клипами* (*clips*). У каждого клипа есть собственная независимая временная диаграмма и холст (*canvas*). Рабочий стол (Stage) является холстом для главного фильма и может содержать другие клипы. Клип, который содержит в себе другой клип, называется *клипом-хозяином* (*host clip*), или *родительским клипом* (*parent clip*).

Документ Flash может включать в себя иерархию взаимосвязанных клипов. Например, главный фильм может содержать горный ландшафт. В отдельном клипе может находиться анимация персонажа, который можно перемещать по ландшафту, создавая впечатление ходьбы. Еще один клип внутри клипа персонажа можно независимым образом использовать для анимации моргающих глаз. Когда независимые элементы персонажа мультипликации воспроизводятся вместе, они кажутся единым элементом содержимого. Кроме того, компоненты могут разумным образом взаимодействовать друг с другом: можно сделать так, чтобы глаза моргали, когда персонаж останавливается, или ноги изображали движение, когда персонаж начинает перемещаться.

ActionScript предоставляет детальный контроль над клипами: можно воспроизводить клип, можно останавливать его, перемещать воспроизводящую головку вдоль временной диаграммы, программным образом задавать свойства (такие как размер, угол поворота, степень прозрачности и положение на рабочем столе) и работать с ним, как с насто-

ящим 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 сайта или просто как контейнер для фонового звука мультипликации. В конце этой главы будет показано применение клипов в качестве стрелок часов и ответов в вопроснике с вариантами выбора.

# «Объектность» клипов

Во Flash 5 с клипами можно работать как с объектами, о которых рассказывалось в главе 12 «Объекты и классы». Можно получать и задавать свойства клипа, а также вызывать встроенные или пользовательские методы клипа. От других объектов клипы отличаются тем, что производимые с ними операции могут иметь в проигрывателе Flash видимые или слышимые результаты.

В действительности клипы не совсем объекты: нет класса или конструктора *MovieClip*, как нельзя и использовать литерал объекта для создания экземпляра клипа в коде. 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 различие между клипами и настоящими объектами заключается в способе выделения для них памяти (создание) и освобождения памяти (удаление). Детали можно найти в главе 15 «Более сложные темы». Однако, помимо этих технических особенностей, работа с клипами почти всегда происходит точно так же, как с объектами.

Какое же влияние «объектность» клипов оказывает на их использование в ActionScript? Прежде всего, она определяет способ управления клипами и изучения их свойств. Клипами можно непосредственно управлять с помощью встроенных методов, например:

```
eyes.play();
```

Свойства клипов можно извлекать и задавать с помощью оператора точки, как это делается для любого объекта:

```
ball._xscale = 90;
var radius = ball._width / 2;
```

Переменная в клипе просто является свойством этого клипа, и с помощью оператора точки значения переменных можно задавать и извлекать:

```
myClip.myVariable = 14;
x = myClip.myVariable;
```

Вложенные клипы можно считать свойствами родительского клипа как объекта. 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 контроля за воспроизведением. Но использование клипов как объектов позволяет также работать с клипами как с данными: можно сохранить 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 moveClip (clip, x, y) {
  clip._x = x;
  clip._y = y;
}
moveClip(ball, 14, 399);
```

В оставшейся части главы будет рассказано о том, как ссылаться на клипы, управлять ими и обрабатывать их как объекты данных.

# Типы клипов

Не все клипы создаются одинаковыми. На самом деле, во Flash есть три разных типа клипов:

- Главные фильмы
- Обычные клипы
- Интеллектуальные клипы (Smart Clips)

Помимо этих трех официальных разновидностей можно определить еще четыре подкатегории, исходя из применения обычных клипов:

- Клипы обработки
- Клипы сценариев
- Связанные клипы
- Порождающие клипы

Эти последние четыре неофициальные категории не входят в число формальных терминов, используемых 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 содержит основную временную диаграмму и рабочий стол, присутствующие в каждом документе. Главный фильм служит основанием для всего содержимого документа, включая остальные клипы. Иногда главный фильм называют главной временной диаграммой (*main timeline*), *временной диаграммой главного фильма* (*main movie timeline*), *главным рабочим столом* (*main Stage*), или просто *корнем* (*root*).

С главными фильмами можно обращаться почти так же, как с обычными клипами, тем не менее:

- 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( )*, *swapDepths( )*.
- К главному фильму нельзя прикреплять обработчики событий.
- Ссылаться на главный фильм можно посредством встроенных глобальных свойств `_root` и `_leveln`.

Обратите внимание, что, хотя файл *.swf* содержит только один главный фильм, одновременно в проигрывателе Flash могут находиться несколько документов: можно загрузить несколько документов *.swf* (и, следовательно, несколько главных фильмов) на стек *уровней* с помощью функций *loadMovie( )* и *unloadMovie( )*, что будет рассмотрено ниже.

## Обычные клипы фильмов

*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.

## Интеллектуальные клипы

Введенные во Flash 5 интеллектуальные клипы (*Smart Clip*) являются обычными клипами, которые содержат графический интерфейс пользователя, используемый для настройки свойств клипа в среде разработки. Интеллектуальные клипы обычно разрабатываются опытными программистами, чтобы предоставить менее опытным разработчикам документов Flash возможность настройки поведения клипа без знания

того, как работает код. Подробно об интеллектуальных клипах мы расскажем в главе 16 «Среда разработки ActionScript».

## 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 главе 8 «Предложения цикла».

Клипы обработки служат в ActionScript неофициальной альтернативой методам *setTimeout( )* и *setInterval( )* объекта окна JavaScript.

## Клипы сценариев

Как и клип обработки, клип сценария (*script clip*) является пустым клипом, используемым не ради содержания, а для слежения за некоторой переменной или выполнения некоторого сценария. Например, клип сценария можно использовать для хранения обработчиков событий, реагирующих на нажатия клавиш, или события, связанные с мышью.

## Связанные клипы

Связанный клип (*linked clip*) осуществляет 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 используются для динамического генерирования экземпляра клипа непосредственно из символа библиотеки с помощью метода клипа *attachMovie( )*, как будет показано ниже.

## Порождающие клипы

До того как во Flash 5 был введен метод *attachMovie( )*, новые клипы создавались на основе некоего существующего клипа, называемого *порождающим клипом* (*seed clip*), при помощи функции *duplicateMovieClip( )*. *Порождающий клип* находится на рабочем столе исключительно для того, чтобы копироваться посредством *duplicateMovieClip( )*. С появлением метода *attachMovie( )* потребность в порождающих клипах уменьшилась. Однако порождающие клипы и *duplicateMovieClip( )* попрежнему используются, если нужно сохранить в процессе копирования обработчики событий клипа и преобразования.

В фильме, интенсивно использующем *duplicateMovieClip( )* для динамической генерации содержимого, часто можно увидеть ряд порождающих клипов на краю холста фильма. Порождающие клипы исполь-

зуются только для производства клипов-дубликатов и потому хранятся вне рабочего стола.

# Создание клипов фильмов

Обычно с клипами обращаются точно так же, как с объектами данных: их свойства задаются с помощью оператора точки, их методы вызываются с помощью оператора вызова функций (круглые скобки), и хранятся они в переменных, элементах массивов и свойствах объектов. Однако создание клипов происходит не так, как создание объектов данных. Нельзя описать в коде клип в виде литерала, как можно описать объект с помощью литерала объекта. Нельзя также создать клип с помощью функции конструктора клипа типа

```
myClip = new MovieClip();  // Славная попытка, но она не сработает
```

Вместо этого клипы создаются прямо в среде разработки, вручную. Когда клип создан, можно создавать новые независимые дубликаты его с помощью таких команд, как *duplicateMovieClip( )* и *attachMovie( )*.

## Символы и экземпляры клипов

Так же, как все экземпляры объектов основываются на том или ином классе, все экземпляры клипов основываются на клипе-шаблоне, называемом *символом* (*symbol*), или иногда *определением* (*definition*). Символ клипа выступает в качестве модели содержимого и структуры клипа. Перед генерацией конкретного объекта клипа необходимо располагать символом клипа. С помощью символа можно как вручную, так и программным образом создавать клипы, которые должны выводиться в фильме.

Клип, изображаемый на рабочем столе, 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, следует выполнить следующие шаги:

1. Выберите пункт меню Insert → New Symbol. Появится диалоговое окно Symbol Properties.
2. В поле Name введите идентификатор символа.
3. Выберите переключатель Movie Clip.
4. Щелкните OK.

Обычно после этого содержимым клипа заполняются холст символа и временная диаграмма. После своего создания символ располагается в библиотеке и ожидает использования в качестве формы для действительного экземпляра клипа. Однако имеется также возможность пре-

образовать в символ клипа группу фигур и объектов, уже существующих на рабочем столе. Для этого нужно выполнить следующие шаги:

1. Выделить нужные фигуры и объекты.
2. Выбрать пункт меню Insert → Convert to Symbol.
3. В поле Name ввести идентификатор символа.
4. Выбрать переключатель Movie Clip.
5. Щелкнуть OK.

Выделенные для создания нового символа клипа фигуры и объекты будут заменены безымянным экземпляром этого нового клипа. Соответствующий символ клипа появится в библиотеке, готовый к использованию при создании новых экземпляров.

## Создание экземпляров

Существует три способа создания нового экземпляра на основе символа клипа. Два из них программные, а третий является сугубо ручным и используется в среде разработки Flash.

### Создание экземпляров вручную

Экземпляры клипов могут создаваться вручную с помощью библиотеки в среде разработки Flash. При физическом перетаскивании символа клипа из библиотеки на рабочий стол создается новый экземпляр. Созданному таким способом экземпляру нужно вручную дать имя через панель Instance. (К именам экземпляров мы еще вернемся ниже.) Если вам не доводилось работать во Flash с клипами, обратитесь к разделу «Using Symbols and Instances» в справочной системе Macromedia Flash.

### Создание экземпляров с помощью duplicateMovieClip()

Любой экземпляр, уже находящийся на рабочем столе фильма Flash, можно копировать с помощью ActionScript. После этого с новым независимым экземпляром можно обращаться как с совершенно отдельным клипом. Дублировать можно экземпляры клипов, созданные как вручную, так и программным образом. Иными словами, можно дублировать дубликаты.

На практике существуют два способа дублирования экземпляров с помощью *duplicateMovieClip( )*:

- Можно вызвать *duplicateMovieClip( )* в качестве глобальной функции, используя синтаксис:

```
duplicateMovieClip(target, newName, depth);
```

  Здесь `target` является строкой, задающей имя экземпляра, который требуется дублировать. Параметр `newName` является строкой, задаю-

щей идентификатор нового экземпляра, а `depth` является целым числом, определяющим место в стеке программно генерируемых клипов, куда должен быть помещен новый экземпляр.

- Возможен также вызов *duplicateMovieClip( )* в качестве метода существующего экземпляра:

```
myClip.duplicateMovieClip(newName, depth);
```

Здесь *myClip* является именем клипа, дубликат которого хочется создать, а *newName* и *depth* имеют те же значения, что и прежде.

Если экземпляр создан с помощью *duplicateMovieClip( )*, то первоначально он помещается прямо поверх своего порождающего клипа. Поэтому первой задачей после дубликации является перемещение дублированного клипа в новое положение. Например:

```
ball.duplicateMovieClip("ball2", 0);
ball2._x += 100;
ball2._y += 50;
```

Дублированные экземпляры, порождающие клипы которых были преобразованы (например, изменены цвет, угол поворота или размер) с помощью ActionScript или вручную в среде разработки Flash, наследуют начальное преобразование своих порождающих клипов. Последующие преобразования порождающего клипа не влияют на продублированные экземпляры. Аналогично, каждый экземпляр можно изменить независимым образом. Например, если порождающий клип повернуть на 45 градусов, а затем создать дубликат экземпляра, его начальным углом поворота будет 45 градусов:

```
seed._rotation = 45;
seed.duplicateMovieClip("newClip", 0);
trace(newClip._rotation);  // Выводит: 45
```

Если после этого повернуть экземпляр на 10 градусов, угол поворота составит 55 градусов, но угол поворота порождающего клипа по-прежнему останется равным 45 градусам:

```
newClip._rotation += 10;
trace(newClip._rotation);  // Выводит: 55
trace(seed._rotation);     // Выводит: 45
```

Путем дублирования нескольких экземпляров подряд и незначительного преобразования каждого дубликата можно получить интересные и сложные преобразования (это показано для события *load* в примере 10.2).

Использование метода *duplicateMovieClip( )* для дублирования клипов с помощью ActionScript дает и другие преимущества перед размещением клипов в фильме вручную – например, предоставляет следующие возможности:

- Точно управлять появлением клипа на рабочем столе по ходу выполнения программы.
- Точно управлять удалением клипа с рабочего стола по ходу выполнения программы.
- Назначать глубину слоя дублированного клипа относительно других дублированных клипов. (Это было более важно во Flash 4, где не позволялось изменять стек слоев фильма.)
- Копировать обработчики событий клипа.

Такие возможности обеспечивают усиленное программное управление содержанием фильма. Ярким примером служит игра в звездные войны, в которой при нажатии кнопки открытия огня космическим кораблем можно дублировать клип ракеты. Такой клип ракеты можно переместить программным образом, затем поместить позади препятствия в фильме и, наконец, удалить после столкновения с неприятельским кораблем. Клипы, создаваемые вручную, не обладают такой степенью гибкости. Для клипа, созданного вручную, рождение и смерть должны быть предопределены с помощью временной диаграммы, и во Flash 4 нельзя было изменить слой клипа.

## Создание экземпляров с помощью attachMovie()

Подобно *duplicateMovieClip( )*, метод *attachMovie( )* позволяет создать экземпляр клипа; однако, в отличие от *duplicateMovieClip( )*, ранее созданный экземпляр не нужен – новый создается прямо из символа библиотеки фильма. Для использования *attachMovie( )* с целью создания экземпляра символа необходимо сперва экспортировать этот символ из библиотеки. Вот как это делается:

1. Выбрать в библиотеке нужный символ.
2. В меню библиотеки Options выбрать пункт Linkage. Появляется окно диалога Symbol Linkage Properties.
3. Выбрать переключатель Export This Symbol.
4. В поле Identifier ввести уникальное имя символа клипа. Имя может быть любой строкой – часто совпадающей с именем самого символа, но должно отличаться от всех других экспортированных символов клипов.
5. Щелкнуть OK.

Когда символ клипа экспортирован, можно прикреплять новые экземпляры этого символа к существующему клипу путем вызова *attachMovie( )* со следующим синтаксисом:

```
myClip.attachMovie(symbolIdentifier, newName, depth);
```

Здесь *myClip* является именем клипа, к которому нужно прикрепить новый экземпляр. Если опустить *myClip*, то *attachMovie( )* прикрепит новый экземпляр к текущему клипу (тому, в котором находится пред-

ложение *attachMovie( )*). Параметр *symbolIdentifier* является строкой, содержащей имя символа, используемого для генерации экземпляра, как оно указано в поле Identifier параметров Linkage библиотеки; *newName* является строкой, определяющей идентификатор вновь создаваемого экземпляра; *depth* является целым числом, обозначающим место в стеке слоев родительского клипа, куда должен быть помещен новый экземпляр.

Когда экземпляр прикрепляется к другому клипу, он помещается в центре этого клипа, в «слоеном» стеке клипа (о стеках клипов скоро будет рассказано). Когда экземпляр прикрепляется к главному фильму документа, он помещается левый верхний угол рабочего стола, имеющий координаты (0, 0).

## Имена экземпляров

При создании экземпляров им присваиваются идентификаторы, или *имена экземпляров*, чтобы позднее на них можно было ссылаться. Обратите внимание на возникающее при этом отличие от обычных объектов. При создании обычного объекта данных (не клипа) нужно присвоить 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
var thing = new Object();  // Ссылка на объект сохраняется в thing, и в
                           // потом к нему можно обращаться как к thing
```

Экземпляры клипов не обязательно сохранять в переменных, чтобы иметь возможность к ним обращаться. В отличие от обычных объектов данных, экземпляры клипов доступны в ActionScript через имена экземпляров сразу после создания, например:

```
ball._y = 200;
```

Имя каждого экземпляра клипа хранится в его встроенном свойстве _name, которое можно как читать, так и задавать:

```
ball._name = "circle";  // Изменить имя ball на circle
```

При изменении свойства экземпляра _name все дальнейшие ссылки на экземпляр должны использовать новое имя. Например, после выполнения приведенного выше кода ссылка ball прекращает существование, а в будущем для ссылки на экземпляр необходимо использовать circle.

Способ получения экземпляром своего имени зависит от способа его создания. Программно генерируемые экземпляры получают свои имена от функции, которая их создает. Экземплярам, создаваемым вруч-

ную, обычно явно присваиваются имена экземпляров в среде разработки через панель Instance, для чего нужно:

1. Выделить экземпляр на рабочем столе.
2. Выбрать пункт меню Modify → Instance.
3. Ввести имя экземпляра в поле Name.

Если созданному вручную экземпляру не дать имени, оно автоматически присваивается проигрывателем Flash во время выполнения. Автоматические имена экземпляров имеют вид последовательности `instance1`, `instance2`, `instance3`...`instance`*n*, но эти имена не дают осмысленного описания содержимого клипа (а о том, какое автоматическое имя было присвоено, приходится гадать).

Так как имена экземпляров являются идентификаторами, их нужно образовывать согласно правилам создания допустимых идентификаторов, описанным в главе 14 «Лексическая структура». В первую очередь, имена экземпляров не должны начинаться с цифры, а также содержать дефисы и пробелы.

## Импорт внешних фильмов

Мы рассказали о создании экземпляров клипов внутри одного документа, но Flash Player может одновременно отображать несколько документов *.swf*. С помощью *loadMovie( )* (используемой 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 (т. е. на передний план относительно основного фильма). Благодаря расположению содержимого в отдельных файлах обретается точный контроль над процессом загрузки. 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 смогут загружаться в произвольном порядке, и пользователь получит прямой доступ к каждому их них.

Когда внешний файл *.swf* загружается на уровень, его главная временная диаграмма становится корневой 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.

Подобно *duplicateMovieClip( )*, *loadMovie( )* можно вызывать как самостоятельную функцию или как метод экземпляра. При самостоятельном использовании синтаксис *loadMovie( )* такой:

```
loadMovie(URL, location)
```

Здесь *URL* задает адрес внешнего файла *.swf*, который нужно загрузить. Параметр *location* является строкой, указывающей путь к существующему клипу или уровню документа, куда должен быть помещен новый файл *.swf*, т. е. загружаемый фильм. Например:

```
loadMovie("circle.swf", "_level1");
loadMovie("photos.swf", "viewClip");
```

Так как ссылка на клип при использовании в строковом контексте преобразуется в путь, можно задавать *location* в виде ссылки на клип, т. е. `_level1` , а не `"_level1"`. Однако при использовании ссылок будьте осторожны. Если передаваемая ссылка не указывает на реальный клип, функция *loadMovie( )* ведет себя неожиданным образом: она загружает файл *.swf* в *текущую* временную диаграмму. Подробности ищите в части III «Справочник по языку» или в разделе «Проблемы, связанные с перекрытием методов и глобальных функций» ниже в данной главе.

Вариант *loadMovie( )* как метода клипа использует следующий синтаксис:

```
myClip.loadMovie(URL);
```

При использовании в качестве метода *loadMovie( )* предполагает, что внешний *.swf* загружается в *myClip*, поэтому параметр *location*, требуемый для самостоятельной функции *loadMovie( )*, не нужен. Таким образом, задается только путь к загружаемому файлу *.swf* через параметр *URL*. Конечно, *URL* может быть именем локального 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 *myClip* 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 предложений. На самом деле, она выполняется только после того, как будут выполнены *все* другие предложения в блоке.

Нельзя обращаться к свойствам или методам загружаемого извне фильма в том же самом блоке предложений, 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 всегда нужно проверить, закончена ли передача фильма в проигрыватель. Это делается при помощи простого фрагмента кода, часто называемого предварительным загрузчиком (*preloader*), который проверяет, какая часть файла загружена, прежде чем разрешить выполнение каких-либо действий. Предварительные загрузчики можно создавать с помощью свойств клипов `_totalframes` и `_framesloaded` и методов клипов *getBytesLoaded( )* и *getBytesTotal( )*. Примеры кода можно найти в соответствующих статьях части III. Смотрите также пример 10.4, в котором показано, как создать предварительный загрузчик с помощью события клипа *data*.

## Использование loadMovie() с attachMovie()

Загрузка внешнего файла *.swf* в экземпляр клипа с помощью *loadMovie( )* приводит к неожиданному результату: становится нельзя прикреплять к этому клипу экземпляры с помощью *attachMovie( )*. Если в клип загружен внешний файл *.swf*, он больше не принимает прикрепляемых фильмов из библиотеки, от которой он произошел. Например, если *movie1.swf* содержит экземпляр с именем `clipA`, и в `clipA` загружается *movie2.swf*, становится невозможным прикрепление к `clipA` экземпляров из библиотеки *movie1.swf*.

Почему это происходит? 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( )*, возможность прикрепления к клипу фильмов из библиотеки его документа восстановится.

Аналогично, в результате загрузки в клип файла *.swf* с помощью *loadMovie( )* утрачивается возможность копирования этого клипа с помощью *duplicateMovieClip( )*.

# Порядок расположения фильмов и экземпляров в стеке

Все экземпляры клипов и загружаемых извне фильмов, выводимые 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» из двух) всегда закрывает другой клип («нижний» из двух). В принципе, достаточно просто, но при этом главный стек, содержащий *все* экземпляры и файлы *.swf*, фактически дробится на множество меньших подстеков. Сначала мы рассмотрим эти подстеки отдельно, а потом разберемся, как они объединяются, образуя главный стек. (В данном изложении стек не имеет прямого отношения к стекам LIFO и FIFO, обсуждавшимся в главе 11 «Массивы».)

## Внутренний стек слоев

Экземпляры, созданные *вручную* в среде разработки Flash, располагаются в стеке, называемом *внутренним стеком слоев* (*internal layer stack*). Порядок в этом стеке управляется фактическими слоями временной диаграммы фильма: если перекрываются два созданных вручную экземпляра с различных слоев временной диаграммы, то экземпляр, находящийся на более высоком уровне, закрывает экземпляр, находящийся на менее высоком уровне.

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.

Во Flash 5 можно поменять местами положения двух экземпляров во внутреннем стеке слоев с помощью метода *swapDepths( )*, при условии, что они находятся на одной и той же временной диаграмме (т. е. значения свойства `_parent` должны быть одинаковы для обоих клипов). В предшествующих версиях Flash нельзя было изменять внутренний стек слоев посредством ActionScript.

## Стек клипов, создаваемых программно

*Программно создаваемые* экземпляры хранятся в стеке отдельно от экземпляров, создаваемых *вручную* и содержащихся во внутреннем стеке слоев. Каждый экземпляр обладает собственным *стеком программно генерируемых клипов*, содержащим клипы, созданные с помощью *duplicateMovieClip( )* и *attachMovie( )*. Порядок таких клипов в стеке может различаться в зависимости от способа их создания.

## Добавление в стек клипов, созданных с помощью attachMovie()

Новый экземпляр, созданный с помощью *attachMovie( )*, всегда кладется сверху (т. е. на передний план) по отношению к клипу, к которому он был прикреплен. Допустим, к примеру, что в структуре внутренних слоев фильма есть два клипа – X и Y, и что X располагается в слое, лежащем выше Y. Предположим также, что к X прикрепляется новый клип A, а к Y прикрепляется новый клип B:

```
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 содержимым для новых программно генерируемых клипов. Это значит, что можно прикреплять клипы к прикрепленным клипам.

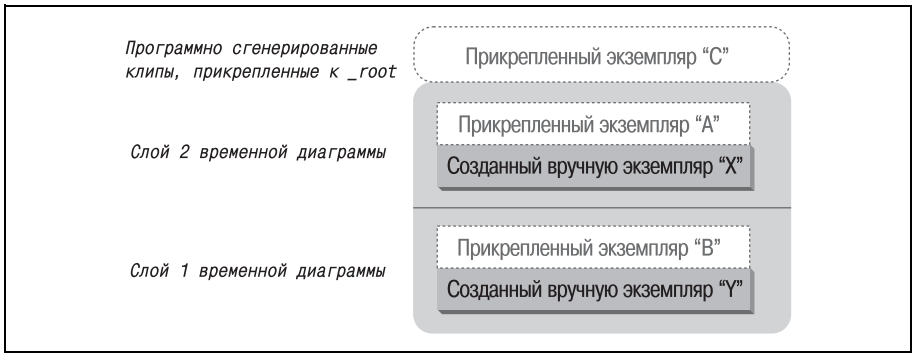


***Рис. 13.2.** Стек экземпляров, показывающий клип, прикрепленный к _root*

Клипы, прикрепленные к фильму _root документа Flash, помещаются в программно генерируемый стек клипов фильма _root, который нахо-

дится впереди *всех* клипов фильма _root, даже тех, которые имеют программно генерируемое содержимое.

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.

## Добавление в стек клипов, созданных с помощью duplicateMovieClip()

Каждый экземпляр, копируемый с помощью *duplicateMovieClip( )*, попадает в программный стек в соответствии с тем, каким образом был создан порождающий клип этого экземпляра:

- Если порождающий клип экземпляра был создан вручную (или был дублирован *duplicateMovieClip( )* из созданного вручную клипа), то новый экземпляр помещается в стек над _root.
- Если же порождающий клип экземпляра был создан с помощью *attachMovie( )*, то новый экземпляр помещается в стек порождающего клипа.

Вернемся к нашему примеру и посмотрим, как он работает. Если создать клип D путем дублирования клипа X (созданного вручную), то клип D помещается в стек над _root, с клипом C. Аналогично, если создать клип E путем дублирования клипа D (полученного из клипа X, созданного вручную), 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 стек над X, с клипом A. Лучше всяких слов это иллюстрирует рис. 13.3.

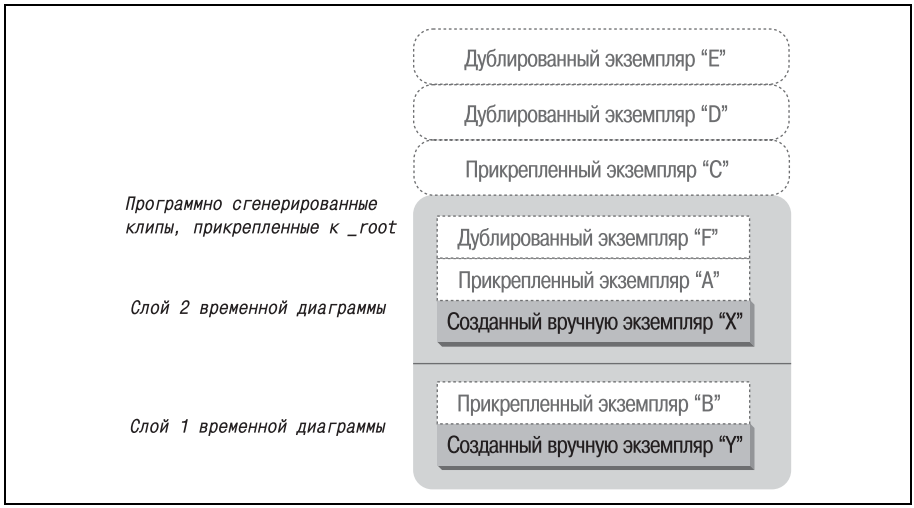


***Рис. 13.3.*** *Стек экземпляров, показывающий различные дублированные клипы*

### Задание глубины для экземпляров в стеке программно генерируемых клипов

Читателя может заинтересовать, 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 функциям *attachMovie( )* 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 (*depth*) может быть любым целым числом и измеряется снизу вверх, так что −1 ниже, чем 0; 1 выше, чем (т. е. находится впереди) глубина 0; глубина 2 еще выше и так далее. Если два программно сгенерированных клипа занимают на экране одну и ту же область, тот из них, который имеет большее значение глубины, выводится впереди другого.

Каждый слой предназначен только для одного жильца. Только один клип может занимать слой стека в некоторый момент времени: если поместить клип в слой, который уже занят, предыдущий клип, занимавший его, удаляется.

Вполне допускается иметь разрывы в глубине клипов: один клип может находиться на глубине 0, другой – на глубине 500, а третий – на глубине 1000. Наличие разрывов не влечет уменьшения производительности или увеличения расхода памяти.

## Стек «_level» документа .swf

Помимо внутреннего стека слоев и стека программно генерируемых клипов 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, присвоив им в стеке документов уровень выше `_level0`. Все содержимое документов, имеющих более высокий уровень в стеке документов, появляется перед документами с более низким уровнем, независимо от порядка в стеках клипов внутри каждого документа.

Так же как стек программно генерируемых клипов разрешает иметь в каждом слое только один клип, 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 документом. Например, можно заменить исходный документ, загрузив новый файл *.swf* на `_level0`. Загрузка нового файла *.swf* на `_level1` приведет к видимому закрытию фильма на `_level0`, но не удалит его из проигрывателя.

Рис. 13.4 подытоживает отношения, существующие между различными стеками, поддерживаемыми Flash Player.

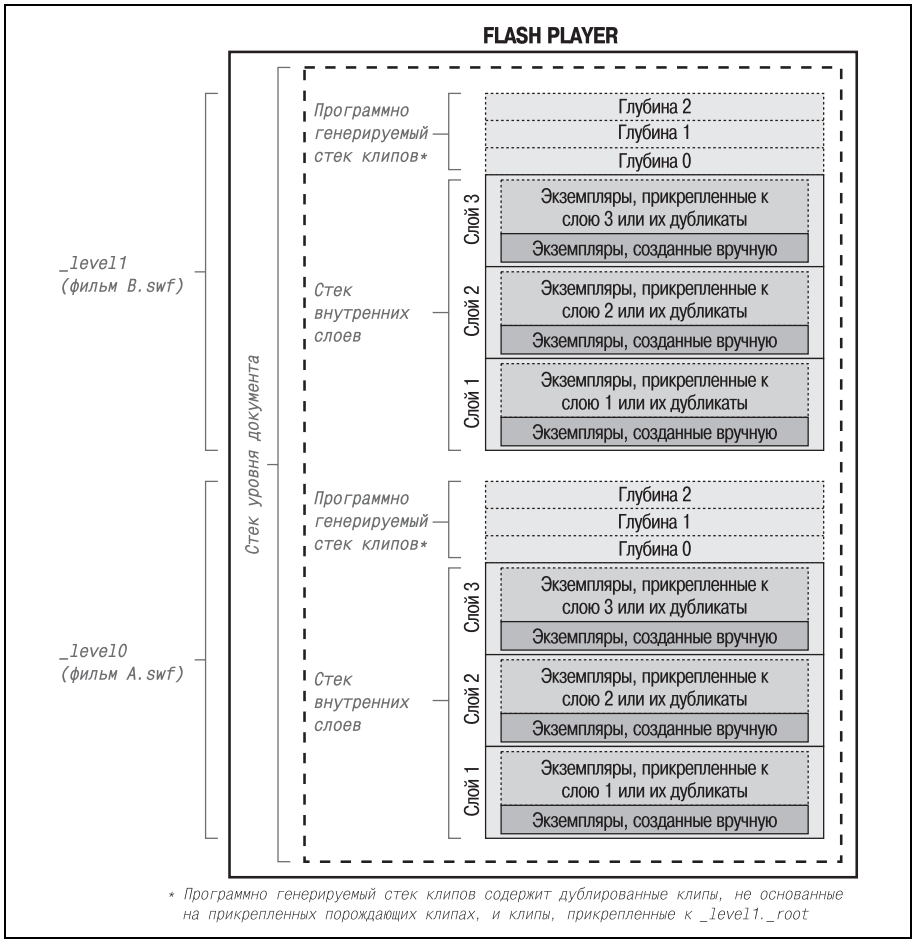


***Рис. 13.4.** Полный стек клипов Flash Player*

## Стеки и порядок выполнения кода

Распределение клипов по слоям и слои временных диаграмм оказывают влияние на порядок выполнения кода. Здесь действуют следующие правила:

- Код в кадрах различных слоев временной диаграммы всегда выполняется сверху вниз.
- При начальной загрузке экземпляров, созданных вручную, код их временных диаграмм и обработчиков событий *load* 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, например, 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 нужно избегать кода, зависящего от их относительного положения. Следует стремиться к созданию такого кода, который можно безбоязненно выполнять вне зависимости от порядка выполнения клипов в стеке. Уберечься от некоторых проблем, обусловленных стеком выполнения, можно, размещая весь код в слое *scripts*, лежащем сверху в каждой содержащей код временной диаграммы.

# Ссылки на экземпляры и главные фильмы

В предыдущих разделах было показано, как создавать и распределять по слоям экземпляры клипов и внешние файлы *.swf* для Flash Player. Чтобы успешно управлять содержимым с помощью ActionScript, нужно 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 нужно сослаться на клип или фильм, достаточно просты, способы создания ссылок многочисленны и разнообраз-

ны. На протяжении оставшейся части данного раздела будут рассмотрены средства ActionScript для создания ссылок на экземпляры и фильмы.

## Использование имен экземпляров

Ранее отмечалось, что ссылаться на клипы можно по *именам* их *экземпляров*, например:

```
trace(myVariable);  // Ссылка на переменную
trace(myClip);      // Ссылка на клип
```

Для непосредственной ссылки на экземпляр (как в приведенном примере) экземпляр должен располагаться на временной диаграмме, к которой прикреплен код. Например, если экземпляр с именем `clouds` помещен на главную временную диаграмму документа, можно обращаться к `clouds` из кода, прикрепленного к главной временной диаграмме:

```
// Установить свойство экземпляра
clouds._alpha = 60;
// Вызвать метод экземпляра
clouds.play();
// Поместить экземпляр в массив с другими родственными экземплярами
var background = [clouds, sky, mountains];
```

Если экземпляр, на который нужно сослаться, находится не на той временной диаграмме, на которой код, необходимо использовать более развитый синтаксис, описываемый ниже в разделе.

## Ссылка на текущий экземпляр фильма

При ссылке на клип не всегда обязательно указывать имя экземпляра. Код, прикрепленный к кадру в экземпляре, может ссылаться на свойства и методы этого экземпляра непосредственно, без указания имени экземпляра.

Например, чтобы задать свойство `_alpha` клипа `cloud`, можно поместить в кадр на временной диаграмме `cloud` такой код:

```
_alpha = 60;
```

Аналогично, для вызова метода *play( )* клипа `cloud` из кадра на временной диаграмме `cloud` можно просто выполнить:

```
play();
```

Этот прием можно использовать на любых временных диаграммах, в том числе временных диаграммах главных фильмов. Например, следующие два предложения являются синонимами, если они прикреплены к кадрам на главной временной диаграмме документа Flash.

Первое ссылается на главный фильм неявным образом, в то время как второе делает это явно с помощью глобального свойства _root:

```
gotoAndStop(20);
_root.gotoAndStop(20);
```

Как было отмечено в главе 10 «События и обработчики событий», код в обработчике события экземпляра может, как и код временной диаграммы, непосредственно ссылаться на свойства и методы. Например, к cloud можно прикрепить следующий обработчик события, который задает свойство cloud и затем вызывает его метод, не ссылаясь явно на экземпляр cloud:

```
onClipEvent (load) {
  _alpha = 60;
  stop();
}
```

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 нужно использовать явную ссылку. Ниже нам еще придется подробнее поговорить о конфликтах методов и глобальных функций в разделе «Проблемы, связанные с перекрытием методов и глобальных функций».

## Возвратные ссылки с помощью ключевого слова this

Если нужно 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 предложения окажутся равносильными, если они прикреплены к кадру на временной диаграмме экземпляра cloud:

```
_alpha = 60;       // Неявная ссылка на текущую временную диаграмму
this._alpha = 60;  // Явная ссылка на текущую временную диаграмму
```

Есть две причины для использования this при ссылке на клип, даже если на клип можно сослаться непосредственно. 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, задающего «целевой» клип. Для решения этой проблемы используется this, например:

```
this.duplicateMovieClip("newClouds", 0);  // Вызвать метод экземпляра

// Если опустить ссылку this, возникнет ошибка
duplicateMovieClip("newClouds", 0);  // Ой!
```

С помощью this удобно передавать ссылки на текущую временную диаграмму в функции, действующие над клипами:

```
// Вот функция, обрабатывающая клипы
function moveTo (theClip, x, y) {
  theClip._x = x;
  theClip._y = y;
}

// Теперь вызовем ее с текущей временной диаграммой
moveTo(this, 150, 125);
```

Если вы интенсивно используете объектно-ориентированное программирование, будьте осторожны с использованием ключевого слова this при ссылках на экземпляры и фильмы. Помните, что внутри пользовательского метода или конструктора объекта слово this имеет совсем другой смысл и не является ссылкой на текущую временную диаграмму. За подробностями обращайтесь к 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 игре клип spaceship (космический корабль) может содержать экземпляр клипа blinkingLights (мерцающих огней) или клипа burningFuel (горящее топливо). Либо клип персонажа face (лицо) может содержать отдельные клипы eyes, nose и mouth (глаза, нос и рот).

Ранее было вкратце рассмотрено, каким образом 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 можно 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 точки). Допустим, например, что clipB помещен на холст clipA. Для доступа к clipB из кадра на временной диаграмме clipA используется непосредственная ссылка на clipB:

```
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 развивать до бесконечности. Так как каждый экземпляр клипа, помещенный на временную диаграмму другого клипа, становится свойством приютившего его клипа, можно обходить иерархию, разделяя экземпляры операторами точки, например:

```
clipA.clipB.clipC.clipD.gotoAndStop(5);
```

Теперь, когда мы увидели, как перемещаться вниз по иерархии экземпляров, посмотрим, как можно 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:

```
myClip._parent
```

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 оператора точки ссылаться на различные клипы в иерархии. Допустим, что в кадр временной диаграммы clipB помещен такой код:

```
_parent       // Ссылка на clipA
this          // 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 clipB), которым в
                 // данном случае является главная временная диаграмма
```

Заметьте, что нет необходимости (хотя и допустимо) проделывать окольный путь 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. Такие окружные ссылки необязательны, но демонстрируют гибкость точечной записи:

```
clipC._parent                  // Окружная ссылка на clipB
                               // (текущая временная диаграмма)
clipC._parent._parent._parent  // Окружная ссылка на главную
                               // временную диаграмму
```

Обратите внимание на использование оператора точки для спуска по иерархии клипов и свойства _parent для восхождения по ней. Если для вас это в новинку, то, вероятно, следует попробовать построить иерархию clipA, clipB, clipC во Flash и использовать код нашего примера. Умение правильно ссылаться на экземпляры составляет одну из основ мастерства хорошего программиста 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 (т. е. 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). См. рис. 13.5.

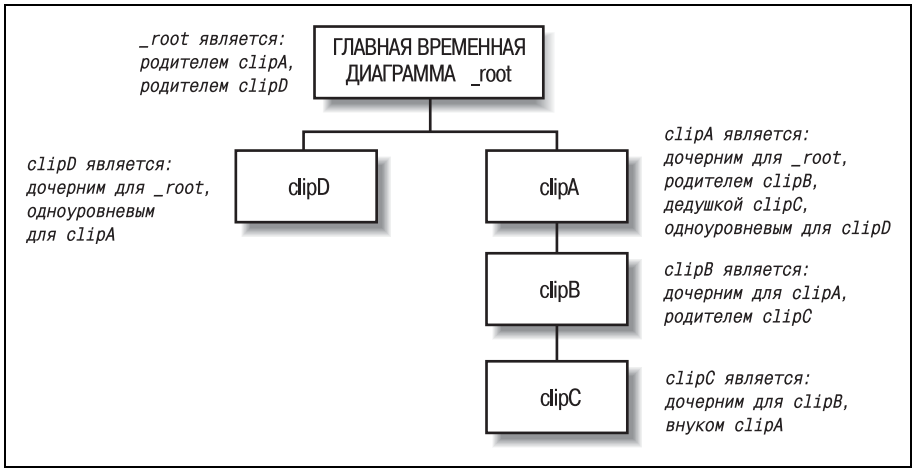


***Рис. 13.5.** Пример иерархии клипов*

Поэтому, как низко ни опустишься по родословному дереву, вернувшись назад на то же число шагов, обязательно попадешь в исходную точку. Таким образом, 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 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): ссылка на to `_root. _parent` возвращает `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 методы 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 глубоко в 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` имеет одно и то же значение, независимо от того, из какого клипа к нему обращаются. Следующие ссылки эквивалентны одна другой:

```
_parent._root
this._root
_root
```

Поэтому можно и нужно использовать `_root`, если неизвестно положение данного клипа в иерархии. Рассмотрим, к примеру, такую иерархию, в которой `circle` является дочерним для главной временной диаграммы фильма, а `square` является дочерним для `circle`:

```
main timeline
  circle
    square
```

Пусть теперь 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. Если этот же код выполнить в `square`, клип `circle` (а не главный фильм) будет перемещен на 10 пикселов вправо. Чтобы сценарий перемещал на 10 пикселов главный фильм вне зависимости от места своего выполнения, нужно 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 успешно 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. Не требуется гадать, на что они ссылаются.

## Ссылки на другие документы в проигрывателе Flash с помощью _leveln

Если в стек документов проигрывателя Flash 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 уровень документа, на который нужно сослаться.

Поэтому `_level0` представляет документ на самом нижнем уровне стека документов (документы на более высоких уровнях будут выводиться на переднем плане). Если на `_level0` с помощью *loadMovie( ) не был загружен никакой фильм*, то `_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 точки:

```
_level1.clipA.stop();
```

Как и ссылки `_root`, ссылки на свойства `_level`*n* называются *абсолютными*, так как приводят в одно и то же место из любой точки документа.

Обратите внимание на то, что `_leveln` 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 свойство _level*n* является ссылкой на главную временную диаграмму фильма на указанном 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, с 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. В редакторе ActionScript есть инструмент Insert Target Path (показанный на рис. 13.6), который позволяет визуально создавать ссылку на клип, освобождая от ручного труда.

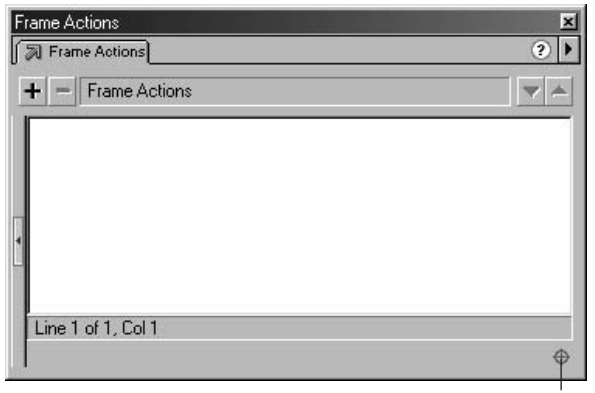


***Рис. 13.6.*** *Кнопка Insert Target Path*

Чтобы воспользоваться средством Insert Target Path, нужно выполнить следующие шаги:

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 нужно сослаться.

4. Выбрать тип ссылки: *absolute reference*, чтобы начать ссылку с `_root`, или *relative reference*, чтобы выразить ссылку на целевой клип относительно клипа, в котором содержится данный код.
5. Если осуществляется экспорт в формат Flash 4, выбрать кнопку Slashes Notation для совместимости с Flash 4. (Выбираемая по умолчанию кнопка Dot Notation составляет ссылки, не работающие во Flash 4). Смотрите табл. 2.1.

Инструмент Insert Target Path не позволяет генерировать 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 вручную вводить такие ссылки в код при помощи 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 нужно динамически создавать ссылки на клипы на этапе исполнения.

## Использование оператора доступа к элементам массива

Как было показано в главе 5 «Операторы» и главе 12 «Объекты и классы», свойства объектов можно извлекать с помощью оператора точки или оператора доступа к элементам массива, `[]`. Например, следующие два предложения эквивалентны:

```
myObject.myProperty = 10;
myObject["myProperty"] = 10;
```

У оператора доступа к элементам массива есть одно важное свойство, отсутствующее у оператора точки: он позволяет (и даже требует) ссылаться на свойство посредством *строкового выражения*, а не *идентификатора*. Вот, например, выражение конкатенации строк, которое дает законную ссылку на свойство `myProperty`:

```
myObject["myProp" + "erty"];
```

Этот же прием можно использовать 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 использовался оператор точки. 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, можно 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 при ссылке на clipB с помощью оператора [] имя clipB передается как строка, а не как идентификатор. Ссылка в виде строки может быть любым допустимым выражением, возвращающим строку. Вот, например, ссылка на clipB, в которой участвует конкатенация 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:

```
// Остановить clip1, clip2, clip3 и clip4
for (var i = 1; i <= 4; i++) {
  _root["clip" + i].stop();
}
```

Сила!

## Хранение ссылок на клипы в контейнерах данных

В начале главы было сказано, что в ActionScript клипы, в сущности, являются объектами данных. Записать ссылку на экземпляр клипа можно 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 13.1, показывающем код для помещения в кадр главной временной диаграммы, для хранения экземпляров и управления ими используются контейнеры данных.

*Пример 13.1. 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 myClips = [clipA, clipA.clipB, clipA.clipB.clipC];
```

```
myClips[0].play();     // Воспроизвести clipA
myClips[1]._x = 200;   // Поместить clipB в 200 пикселах от левого края
                       // рабочего стола

// Остановить все клипы в массиве, использовав цикл
for (var i = 0; i < myClips.length; i++) {
  myClips[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.

## Доступ к клипам с помощью предложения 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]);
}
```

Пример 13.2 показывает, как с помощью цикла *for-in* перечислить все клипы, находящиеся на данной временной диаграмме.

*Пример 13.2. 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);

    // Сделать что-то с клипом
    myClip[property]._x = 300;
    myClip[property].play();
  }
}
```

С помощью цикла *for-in* чрезвычайно удобно обращаться к клипам, которые содержатся в конкретном экземпляре клипа или временной диаграмме. Цикл *for-in* позволяет управлять клипами на любых временных диаграммах, независимо от того, известно ли имя клипа и был ли он создан вручную или программным образом.

В примере 13.3 показана рекурсивная версия предыдущего примера. Она находит все экземпляры клипов на временной диаграмме, а также экземпляры клипов на всех вложенных временных диаграммах.

*Пример 13.3. Рекурсивный поиск всех клипов на временной диаграмме*

```
function findClips (myClip, 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 myClip) {
    // Проверить, является ли текущее свойство myClip клипом
    if (typeof myClip[property] == "movieclip") {
      trace(indent + myClip[property]._name);
     // Проверить, является ли данный клип родительским для каких-либо других клипов
      findClips(myClip[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 «Функции».

## Свойство _name

Как указывалось выше в разделе «Имена экземпляров», имя каждого экземпляра хранится в виде строки во встроенном свойстве `_name`. Как было показано в примере 13.2, это свойство можно использовать для определения имени текущего клипа или какого-либо другого клипа в иерархии экземпляров:

```
_name;              // Имя текущего экземпляра
_parent._name       // Имя клипа, содержащего текущий клип
```

Свойство `_name` удобно использовать, если нужно выполнить условные операции, определенные для конкретных клипов. Например, следующий код создает копию клипа `seedClip` при его загрузке:

```
onClipEvent (load) {
  if (_name == "seedClip") {
    this.duplicateMovieClip("clipCopy", 0);
  }
}
```

Проверяя, является ли `seedClip` именем клипа, явным образом, 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`, являющееся строкой, задающей абсолютный путь к клипу в устаревших

обозначениях Flash 4 с применением косой черты. 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 записи. Функция *targetPath( )* служит синтаксическим эквивалентом _target во Flash 5. Она имеет формат:

```
targetPath(movieClip)
```

где *movieClip* является идентификатором клипа, абсолютную ссылку на который требуется получить. Вот несколько примеров все 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( )* дает полный путь к клипу, в то время как свойство _name дает только имя этого клипа. (Это аналогично указанию полного пути к файлу вместо его имени.) 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, которая на основании данных *targetPath( )* задает свой цвет, соответствующий тому разделу содержимого, в котором она находится. Демонстрация практического применения *targetPath( )* есть в примере для «Selection.setSelection() метод» в части III.

## Использовать ли Tell Target?

Во Flash 4 основным инструментом для ссылки на клипы был *Tell Target*, который, упокой его душу, был довольно громоздким и устарел благодаря появлению во Flash 5 значительно более элегантной объектной модели. Функция *Tell Target* является устаревшей (т. е. использовать ее больше не рекомендуется). Хотя функцию *tellTarget( )* по-прежнему можно использовать для программирования в стиле Flash 4, в будущем она, вероятно, исчезнет.

Рассмотрим такой код, в котором *Tell Target* используется для воспроизведения экземпляра с именем closingSequence:

```
Begin Tell Target ("closingSequence")
  Play
End Tell Target
```

Во Flash 5 просто вызывается 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 предпочтительнее использовать предложение *with( )*, описанное в главе 6 «Предложения»:

```
with (ball) {
  _xscale = 5;
  play();
}
```

Подробнее об устаревших элементах ActionScript из Flash 4 и их эквивалентах во Flash 5 читайте в приложении C «Обратная совместимость».

# Удаление экземпляров клипов и главных фильмов

Мы научились создавать клипы и пользоваться ссылками на них; теперь посмотрим, как превращать их обратно в электроны (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 изгонять клип неявно – путем загрузки, прикрепления или дублирования на его место нового клипа. 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:

```
unloadMovie(clipOrLevel);   // Глобальная функция
```

```
clipOrLevel.unloadMovie();  // Метод
```

В формате глобальной функции *clipOrLevel* является строкой, указывающей путь к клипу или уровню, которые должны быть выгружены. Благодаря автоматическому преобразованию значений *clipOrLevel* может также быть ссылкой на клип (при использовании в качестве строк клипы преобразуются в пути). В формате 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 (т. е. _level0, _level1, _level2), *unloadMovie( )* полностью удаляет указанный уровень и содержащийся на нем фильм. Последующие ссылки на этот 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 рабочем столе. На эту оболочку можно ссылаться, пока экземпляр не будет удален окончательно посредством применения *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, вполне допускается вызывать функции с экземпляром по имени clipA в следующей последовательности (в реальном приложении к этим предложениям нужно добавить соответствующий код предварительного загрузчика):

```
clipA.loadMovie("section1.swf");  // Загрузить документ в clipA
clipA.unloadMovie();              // Выгрузить документ, сохранив действующим
clipA.loadMovie("section2.swf");  // Загрузить в clipA другой документ
```

Сделаем одно предупреждение, связанное с такими действиями. При использовании 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, то тот клип, из которого вызвана функция *unloadMovie( ), будет выгружен*.

Например, если _level1 не определен и из главной временной диаграммы _level0 выполняется представленный ниже код, то будет выгружен _level0:

```
unloadMovie(_level1);
```

Такое поведение имеет свою логику, но об этом будет сказано ниже в разделе «Проблемы, связанные с перекрытием методов и глобальных функций». Избежать этой проблемы можно 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 прикреплением или дублированием, можно использовать функцию *removeMovieClip( )*. 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 использовать в формате метода или глобальной функции (несмотря на различие синтаксиса, результат одинаков):

```
removeMovieClip(clip)    // Глобальная функция
clip.removeMovieClip( )  // Метод
```

В формате глобальной функции *clip* является строкой, указывающей путь к клипу, который нужно удалить. Благодаря автоматическому преобразованию значений *clip* может также быть ссылкой на клип (при использовании в качестве строк клипы преобразуются в пути). В формате метода *clip* 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 *clip*.removeMovieClip() все ссылки на *clip* возвращают значение undefined.

## Ручное удаление экземпляров, созданных вручную

Экземпляры клипов, созданных вручную в среде разработки Flash, также имеют ограниченную продолжительность жизни: они удаляются, когда воспроизводящая головка 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, следует 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, нужно прикрепить постоянно требующийся код к кадру на главной временной диаграмме фильма.) Чтобы спрятать клип, все еще находящийся на временной диаграмме, нужно просто поместить его вне видимой области рабочего стола и присвоить его свойству `_visible` значение `false`. Присваивание свойству `_x` клипа очень большого положительного числа или очень маленького отрицательного также обеспечивает невидимость клипа для пользователя при сохранении его в памяти.

# Встроенные свойства клипов

В отличие от общих объектов класса *Object*, у которых мало встроенных свойств, у каждого клипа встроенных свойств масса. Эти свойства описывают физические характеристики клипа и могут использоваться для их изменения. Это базовые инструменты в наборе программиста ActionScript.

Имена всех встроенных свойств клипа начинаются с символа подчеркивания, что отличает их от пользовательских или специальных свойств. Встроенные свойства имеют формат:

```
_property
```

Имена встроенных свойств записываются строчными буквами. Однако, поскольку в ActionScript идентификаторы не чувствительны к регистру, допускается писать их с заглавных букв, хотя это не считается хорошим стилем.

Мы не будем сейчас заниматься старательным описанием встроенных свойств: эти данные есть в части III. Однако представление о том, что такое свойства и какие возможности они предоставляют, можно получить из табл. 13.1, в которой приведен список встроенных свойств клипов и базовое описание их назначения.

*Таблица 13.1. Встроенные свойства клипов*

| Имя свойства | Описание свойства |
|---|---|
| _alpha | Степень прозрачности |
| _currentframe | Позиция воспроизводящей головки |
| _droptarget | Путь к клипу или фильму, куда был сброшен перетаскиваемый клип |
| _framesloaded | Количество загруженных кадров |
| _height | Физическая высота в пикселах (экземпляра, а не исходного символа) |
| _name | Идентификатор клипа, возвращаемый в виде строки |
| _parent | Ссылка на объект временной диаграммы, содержащей данный клип |
| _rotation | Угол поворота (в градусах) |
| _target | Полный путь к клипу в обозначениях с косой чертой |
| _totalframes | Число кадров во временной диаграмме |
| _url | Сетевой адрес *.swf* |
| _visible | Булево значение, указывающее, отображается ли клип |
| _width | Физическая ширина в пикселах (экземпляра, а не исходного символа) |
| _x | Расстояние по горизонтали от левого края рабочего стола в пикселах |
| _xmouse | Положение указателя мыши по горизонтали в системе координат клипа |
| _xscale | Горизонтальный размер в процентном отношении от исходного символа (или главной временной диаграммы для фильмов) |
| _y | Расстояние по вертикали от верхнего края рабочего стола в пикселах |
| _ymouse | Положение указателя мыши по вертикали в системе координат клипа |
| _yscale | Горизонтальный размер в процентном отношении от размера исходного символа (или главной временной диаграммы для фильмов) |

К экземплярам или главным фильмам непосредственно свойство цвета не прикрепляется. Управление цветом происходит не через свойство, а с помощью класса *Color*, с помощью которого создается объект, используемый для управления цветом клипа. Методы объекта *Color* позволяют задавать или получать значения RGB и трансформации для конкретного клипа. Конкретные детали см. в разделе «Класс Color» части III.

# Методы клипов

В главе 12 рассказывалось об особом типе свойств, которые называются *методами* и являются функциями, прикрепляемыми к объекту. Методы чаще всего используются для обработки объектов, к которым прикреплены, взаимодействия с ними и управления ими. Для управления клипами различными программными способами можно использовать встроенные методы клипов. Можно определять и собственные методы клипов для отдельных экземпляров или символов клипов в библиотеке.

## Создание методов клипов

Чтобы добавить в клип новый метод, нужно определить функцию на временной диаграмме клипа (или в одном из обрабочиков событий клипа) или присвоить функцию свойству клипа. Например:

```
// Создать метод, определив функцию на временной диаграмме клипа
function halfSpin() {
  _rotation += 180;
}
// Создать метод, присвоив литерал функции свойству клипа
myClip.coords = function() { return [_x, _y]; };
// Этот метод осуществляет пользовательское преобразование клипа
myClip.myTransform = function () {
  _rotation += 10;
  _xscale -= 25;
  _yscale -= 25;
  _alpha -= 25;
}
```

## Вызов методов клипов

Вызов метода с клипом не отличается от вызова метода с любым объектом. Для этого нужно указать имя клипа и имя метода, например:

```
myClip.methodName();
```

Если методу требуются аргументы, они передаются при вызове:

```
_root.square(5);   // Передать 5 в качестве аргумента метода square()
```

Как отмечалось выше, если код расположен на временной диаграмме клипа или в обработчике событий клипа, большинство методов текущего клипа можно вызывать непосредственно, не указывая идентификатора экземпляра:

```
square(10);   // Вызвать пользовательский метод square() текущего клипа
play();       // Вызвать встроенный метод play() текущего клипа
```

Однако для некоторых встроенных методов должен быть задан идентификатор экземпляра (см. раздел «Проблемы, связанные с перекрытием методов и глобальных функций»).

## Встроенные методы клипов

Напомним, что общий класс *Object* предоставляет объектам, являющимся его членами, встроенные методы *toString( )* и *valueOf( )*. Напомним также, что и другие классы определяют встроенные методы, которые могут использоваться их членами-объектами: у объектов *Date* есть метод *getHours( )*, у объектов *Color* есть *setRGB( )*, у объектов *Array* есть *push( )* и *pop( )* и т. д. Таковы и клипы. Они снабжены целым рядом встроенных методов, используемых для управления появлением и поведением клипов, получения их характеристик и даже создания новых клипов. Методы клипов находятся в центре ActionScript. В табл. 13.2 представлен обзор методов клипов, о которых детально рассказывается в части III.

*Таблица 13.2. Встроенные методы клипов*

| Имя метода | Описание метода |
|---|---|
| *attachMovie( )* | Создает новый экземпляр |
| *duplicateMovieClip( )* | Создает копию экземпляра |
| *getBounds( )* | Описывает видимую область, занимаемую клипом |
| *getBytesLoaded( )* | Возвращает количество загруженных байтов экземпляра или фильма |
| *getBytesTotal( )* | Возвращает физический размер экземпляра или фильма в байтах |
| *getURL( )* | Загружает в броузер внешний документ (обычно файл *.html*) |
| *globalToLocal( )* | Преобразует координаты рабочего стола в координаты клипа |
| *gotoAndPlay( )* | Перемещает головку воспроизведения в новый кадр и воспроизводит фильм |
| *gotoAndStop( )* | Перемещает головку воспроизведения в новый кадр и останавливает ее там |
| *hitTest( )* | Определяет, находится ли точка внутри клипа |
| *loadMovie( )* | Загружает в проигрыватель внешний файл *.swf* |
| *loadVariables( )* | Загружает в клип или фильм внешние переменные |
| *localToGlobal( )* | Преобразует координаты клипа в координаты рабочего стола |

*Таблица 13.2 (продолжение)*

| Имя метода | Описание метода |
|---|---|
| *nextFrame( )* | Перемещает головку воспроизведения вперед на один кадр |
| *play( )* | Воспроизводит клип |
| *prevFrame( )* | Перемещает головку воспроизведения назад на один кадр |
| *removeMovieClip( )* | Уничтожает дублированный или прикрепленный экземпляр |
| *startDrag( )* | Вызывает физическое перемещение экземпляра или фильма вслед за указателем мыши по рабочему столу |
| *stop( )* | Останавливает воспроизведение экземпляра или фильма |
| *stopDrag( )* | Прекращает операцию перетаскивания, происходящую в данный момент |
| *swapDepths( )* | Изменяет слой, занимаемый экземпляром в стеке экземпляров |
| *unloadMovie( )* | Удаляет экземпляр или главный фильм из уровня документа или содержащего его клипа |
| *valueOf( )* | Строка, представляющая путь к экземпляру в абсолютном виде с использованием точечной |

## Проблемы, связанные с перекрытием методов и глобальных функций

Как неоднократно говорилось на протяжении данной главы, имена некоторых методов клипов совпадают с именами глобальных функций. Вы сами можете убедиться в этом с помощью среды разработки Flash. Откройте панель Actions, перейдите в режим Expert Mode и посмотрите в папку Actions. Вы увидите длинный список действий, в том числе *gotoAndPlay( ), gotoAndStop( ), nextFrame( )* и *unloadMovie( )*. Эти действия могут выполняться и как методы клипов. Такое дублирование касается не просто разделения на категории: действия являются глобальными функциями, совершенно отличными от методов клипов с теми же именами.

Поэтому при выполнении

```
myClip.gotoAndPlay(5);
```

происходит вызов *метода* с именем *gotoAndPlay( )*. Однако при выполнении

```
gotoAndPlay(5);
```

происходит обращение к *глобальной функции* с именем *gotoAndPlay( )*. У этих двух команд одинаковые имена, но они разные. Глобальная функция *gotoAndPlay( )* действует над текущим экземпляром или фильмом. Метод *gotoAndPlay( )* действует над объектом клипа, вызвавшим его. 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 с именем *target*, указывающего клип, над которым должна действовать функция. Этот параметр не требуется соответствующим этим функциям вариантам в виде методов, потому что методы автоматически действуют над клипами, через которые они вызываются. Например, *unloadMovie( )* в форме метода действует так:

```
myClip.unloadMovie();
```

В форме метода *unloadMovie( )* вызывается без параметров и автоматически выполняет действия с *myClip*. Но в форме глобальной функции *unloadMovie( )* действует так:

```
unloadMovie(target);
```

Глобальной функции необходимо передать параметр *target*, который укажет фильм, подлежащий выгрузке. Почему возникает такая проблема? Во-первых, потому что можно ошибочно полагать, будто текущий документ будет выгружен при вызове глобальной версии *unloadMovie( )* без параметров, подобно тому как без параметров используется *gotoAndPlay( )*:

```
unloadMovie();
```

Такой формат *не* выгружает текущий клип. Он вызывает ошибку «Wrong number of parameters» (неверное количество параметров). Вторая причина того, что параметры *target* в глобальных функциях могут вызывать проблемы, несколько более сложна, и ее будет достаточно трудно проследить, если вы к этому не готовы. При передаче клипа *target* в глобальную функцию, которая требует параметра *target*, можно использовать строку, описывающую путь к клипу, которым нужно управлять, или ссылку на клип. Например:

```
unloadMovie(_level1);    // Целевой клип задан как ссылка
unloadMovie("_level1");  // Целевой клип задан как строка
```

Ссылкой можно пользоваться просто потому, что ссылки на объекты клипов при использовании в строковом контексте преобразуются в пути к клипам. Достаточно просто, но если параметр *target* разрешается в пустую строку или значение undefined, то *функция действует над текущей временной диаграммой*! Например:

```
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 могут возникать совершенно неожиданные результаты. Посмотрим, что получится, если сослаться на несуществующий уровень:

```
unloadMovie(_level1);
```

Если `_level1` пуст, то интерпретатор разрешает ссылку так, как если бы она была необъявленной переменной. В результате получается значение `undefined`, поэтому функция действует над текущей временной диаграммой, а не над `_level1`! Как же с этим справиться? Есть несколько вариантов выбора. Можно проверить, существует ли цель, прежде чем выполнять над ней функцию:

```
if (_level1) {
  unloadMovie(_level1);
}
```

Можно взять за правило всегда использовать строку для указания цели. Если путь, указанный в строке, не разрешается в реальный клип, функция просто не работает:

```
unloadMovie("_level1");
```

В некоторых случаях можно использовать для той же операции эквивалентную функцию с числовым аргументом:

```
unloadMovieNum(1);
```

Либо можно вообще избавиться от проблем, если всегда пользоваться только методами:

```
_level1.unloadMovie();
```

Вот, для справки, перечень функций, которые мутят воду (глобальных функций Flash 5 ActionScript, принимающих параметр *target*):

*duplicateMovieClip( )*
*loadMovie( )*
*loadVariables( )*
*print( )*
*printAsBitmap( )*
*removeMovieClip( )*
*startDrag( )*
*unloadMovie( )*

Если в фильме возникают необъяснимые проблемы, то, возможно, нужно взять этот список и проверить, не вызваны ли они неправильным использованием глобальной функции. При передаче ссылки на клип в качестве параметра *target* лишний раз проверьте правильность синтаксиса.

# Практические примеры клипов

Основы программирования клипов нами изучены. Применим полученные знания для разработки двух очень разных приложений, каждое из которых показывает пример типичного использования клипов как основных контейнеров содержимого.

## Создание часов с помощью клипов

В этой главе было показано, как создавать клипы с помощью *attachMovie( )* и как задавать свойства клипов с помощью оператора точки. Эти сравнительно простые средства и некоторая помощь со стороны классов *Date* и *Color* достаточны для изготовления часов с действующими часовой, минутной и секундной стрелками.

Во-первых, сделаем циферблат и стрелки часов, для чего нужно пройти следующие шаги (обратите внимание, что мы не станем размещать детали часов на главном рабочем столе, а целиком создадим часы с помощью ActionScript):

1. Начать новый фильм Flash.
2. Создать символ клипа с именем `clockFace` (циферблат), содержащий черный круг шириной 100 пикселов.
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 → Panels → Info.
5. Поместить нижний конец линии в центр клипа, присвоив ее координатам значения x = 0, y = −50.

Теперь нужно экспортировать символы `clockFace` и `hand`, чтобы их экземпляры можно было динамически прикреплять к фильму:

1. Выбрать клип `clockFace` в библиотеке, затем выбрать меню Options → Linkage. Появится окно диалога Symbol Linkage Properties.
2. Выбрать Export This Symbol.
3. В поле Identifier ввести `clockFace` и щелкнуть OK.
4. Экспортировать клип `hand`, повторив шаги с 1 по 3 и дав клипу идентификатор `hand`.

Циферблат и стрелки готовы и могут присоединяться к фильму. Теперь напишем сценарий, который помещает детали часов на рабочий стол и раз в секунду задает их положение:

1. Добавить сценарий, приведенный в примере 13.4, в кадр 1 слоя 1 главной временной диаграммы.
2. Переименовать слой 1 в *scripts*.

Сначала просмотрите пример 13.4 в целом, затем мы разберем его по частям.

*Пример 13.4. Аналоговые часы*

```
// Создать циферблат и стрелки часов
attachMovie("clockFace", "clockFace", 0);
attachMovie("hand", "secondHand", 3);
attachMovie("hand", "minuteHand", 2);
attachMovie("hand", "hourHand", 1);

// Задать положение и размеры циферблата
clockFace._x = 275;
clockFace._y = 200;
clockFace._height = 150;
clockFace._width = 150;

// Задать положение, размеры и цвет стрелок
secondHand._x = clockFace._x;
secondHand._y = clockFace._y;
secondHand._height = clockFace._height / 2.2;
secondHandColor = new Color(secondHand);
secondHandColor.setRGB(0xFFFFFF);
minuteHand._x = clockFace._x;
minuteHand._y = clockFace._y;
minuteHand._height = clockFace._height / 2.5;
hourHand._x = clockFace._x;
hourHand._y = clockFace._y;
hourHand._height = clockFace._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._rotation = 360 * dayPercent + hourPercent * (360 / 24);
  minuteHand._rotation = 360 * hourPercent;
  secondHand._rotation = 360 * minutePercent;
}
```

Здесь довольно много кода. Рассмотрим его.

Сначала прикрепляется клип clockFace, которому присваивается глубина 0 (нужно, чтобы он находился за стрелками часов):

```
attachMovie("clockFace", "clockFace", 0);
```

Затем прикрепляются три экземпляра символа hand, которым присваиваются имена secondHand, minuteHand, hourHand. Каждая стрелка располагается в собственном слое в программно генерируемом стеке клипов над главной временной диаграммой. secondHand (глубина 3) находится поверх minuteHand (глубина 2), находящейся поверх hourHand (глубина 1):

```
attachMovie("hand", "secondHand", 3);
attachMovie("hand", "minuteHand", 2);
attachMovie("hand", "hourHand", 1);
```

В этом месте код помещает часы в левый верхний угол рабочего стола. Затем клип clockFace перемещается в центр рабочего стола и увеличивается в размерах при помощи свойств _height и _width:

```
clockFace._x = 275;           // Установить положение по горизонтали
clockFace._y = 200;           // Установить положение по вертикали
clockFace._height = 150;      // Установить высоту
clockFace._width = 150;       // Установить ширину
```

После этого помещаем на часы клип secondHand и делаем его почти таким же длинным, как радиус клипа clockFace:

```
// Поместить secondHand поверх clockFace
secondHand._X = clockFace._x;
secondHand._y = clockFace._y;
// Установить размер secondHand
secondHand._height = clockFace._height / 2.2;
```

Вспомните, что линия в символе hand красная, поэтому пока все наши экземпляры hand красные. Чтобы выделить клип secondHand, окрасим его в белый цвет, воспользовавшись классом *Color*. Обратите внимание на использование шестнадцатеричного значения цвета 0xFFFFFF (дополнительные сведения о работе с цветом есть в разделе «Color класс» части III):

```
// Создать новый объект Color для управления secondHand
secondHandColor = new Color(secondHand);
// Назначить secondHand белый цвет
secondHandColor.setRGB(0xFFFFFF);
```

Затем задаются положение и размер minuteHand и hourHand точно так же, как для secondHand:

```
// Поместить minuteHand поверх clockFace
minuteHand._x = clockFace._x;
minuteHand._y = clockFace._y;
// Сделать minuteHand короче, чем secondHand
```

```
minuteHand._height = clockFace._height / 2.5;
// Поместить hourHand поверх clockFace
hourHand._x = clockFace._x;
hourHand._y = clockFace._y;
// Сделать hourHand самой короткой
hourHand._height = clockFace._height / 3.5;
```

Теперь нужно задать угол поворота стрелок часов согласно текущему времени. Однако угол поворота недостаточно задать один раз. Необходимо делать это многократно, чтобы часы двигались с течением времени. Поэтому код для поворота помещен в функцию с именем *updateClock( )*, которая будет вызываться многократно:

```
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 в виде процентов
  var hourPercent = now.getMinutes()/60;
  // Определим число прошедших секунд минуты в виде процентов
  var minutePercent = now.getSeconds()/60;
  // Повернуть стрелки на циферблате на соответствующую величину
  hourHand._rotation = 360 * dayPercent + hourPercent * (360 / 24);
  minuteHand._rotation = 360 * hourPercent;
  secondHand._rotation = 360 * minutePercent;
}
```

Первая задача *updateClock( )* – получить и запомнить текущее время. Это осуществляется путем создания экземпляра класса *Date* и помещения его в локальную переменную `now`. Затем определим в виде процента величину перемещения стрелки по циферблату – примеримся, где разрезать пирог. Текущий час всегда представляет собой некоторую долю 12, в то время как текущая минута и секунда всегда являются некоторой долей 60. На основании этих процентов присваивается значение свойству `_rotation` для каждой стрелки. Для `hourHand` мы отражаем не только процент дня, но также процент текущего часа.

Наши часы, в сущности, готовы. Все, что осталось сделать, это вызывать функцию *updateClock( )* в каждом проходимом кадре. Вот как это делается:

1. Добавить два ключевых кадра в слой *scripts*.
2. В кадр 2 добавить код: `updateClock();`
3. В кадр 3 добавить код: `gotoAndPlay(2);`

Опробуйте фильм и проверьте, работают ли ваши часы. Если не работают, сравните ваш код с файлом, содержащим пример часов *.fla*, имеющимся в сетевом Хранилище кода, или сравните код с примером 13.4.

Подумайте, как можно развить приложение, показывающее часы. Сможете ли вы преобразовать цикл главной временной диаграммы (между кадрами 2 и 3) в цикл события клипа? Сможете ли вы улучшить переносимость часов, преобразовав их в интеллектуальный клип? А как насчет того, чтобы динамически добавлять отметки часов и минут на циферблате the `clockFace`?

# Последний вопросник

Вот последняя версия вопросника с вариантами ответов, начатого нами еще в главе 1 «Легкое введение для непрограммистов». Эта обновленная версия динамически генерирует все вопросы и ответы с помощью клипов, поэтому вопросник становится наращиваемым до бесконечности и хорошо конфигурируемым. На самом деле мы уже близки к тому, чтобы целиком превратить вопросник в интеллектуальный клип, который смогут настраивать непрограммисты.

Код вопросника показан в примере 13.5 и может быть загружен из сетевого Хранилища кода. Поскольку код теперь полностью генерируется динамически, 99% его помещается в один кадр: больше не нужно заполнять временную диаграмму вопросами. (Нам не хватает только предварительного загрузчика для плавного воспроизведения через сеть.) Обратите внимание на использование `#include` для импорта блока кода из внешнего текстового файла. Дополнительные сведения об `#include` см. в части III и в разделе «Размещение кода ActionScript во внешних файлах» главы 16. В качестве упражнения попробуйте добавить в вопросник новые вопросы, создав новые объекты и поместив в массив с вопросами.

Хотя код окончательного вопросника относительно краток, он насыщен важными техническими приемами. Со всеми из них по отдельности, за исключением `#include`, мы уже сталкивались, но этот расширенный практический пример показывает, как все они могут использоваться совместно. Внимательно изучите комментарии: если вы полностью разберетесь в этой версии вопросника, то будете вполне готовы к созданию развитых приложений на ActionScript.

Более пространное разъяснение кода этого вопросника можно найти по адресу:

*http://www.moock.org/webdesign/lectures/ff2001sfWorkshop*

*Пример 13.5. Опросник с вариантами ответов, последний раз*

```
// КОД В ФРЕЙМЕ 1 ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЫ
//  Остановить фильм
stop();

//  Init main timeline variables
```

```
var displayTotal;            // Текстовое поле для итогового результата
var totalCorrect = 0;           // Количество правильных ответов
var userAnswers = new Array();  // Массив с ответами пользователя
var currentQuestion = 0;        // Номер текущего вопроса

// Импортировать исходный файл с объектами вопросов
#include "questionsArray.as"
 // Объяснение будет дано ниже

// Начать опрос
makeQuestion(currentQuestion);

// Конструктор Question()
function Question (correctAnswer, questionText, answers) {
  this.correctAnswer = correctAnswer;
  this.questionText = questionText;
  this.answers = answers;
}

// Функция для вывода вопроса на экран
function makeQuestion (currentQuestion) {
  // Убрать с рабочего стола предыдущий вопрос
  questionClip.removeMovieClip();

  // Создать и разместить главный клип вопроса
  attachMovie("questionTemplate", "questionClip", 0);
  questionClip._x = 277;
  questionClip._y = 205;
  questionClip.qNum = "question\n  " + (currentQuestion + 1);
  questionClip.qText = questionsArray[currentQuestion].questionText;

  // Создать отдельные клипы ответов в клипе вопроса
  for (var i = 0; i < questionsArray[currentQuestion].answers.length; i++) {
    // Прикрепить связанный клип answerTemplate из библиотеки;
    // Он содержит обобщенную кнопку и текстовое поле для вопроса
    questionClip.attachMovie("answerTemplate", "answer" + i, i);
    // Place this answer clip in line below the question
    questionClip["answer" + i]._y += 70 + (i * 15);
    questionClip["answer" + i]._x -= 100;
    // Записать в текстовое поле клипа ответа соответствующий элемент
    // из массива ответов на данный вопрос
    questionClip["answer" + i].answerText =
        questionsArray[currentQuestion].answers[i];
  }
}

// Функция для регистрации ответов пользователя
function answer (choice) {
  userAnswers.push(choice);
  if (currentQuestion + 1 == questionsArray.length) {
    questionClip.removeMovieClip();
    gotoAndStop ("quizEnd");
```

```
  } else {
    makeQuestion(++currentQuestion);
  }
}

// Функция для подсчета результата пользователя
function gradeUser() {
  // Подсчитать количество вопросов, на которые дан правильный ответ
  for (var i = 0; i < questionsArray.length; i++) {
    if (userAnswers[i] == questionsArray[i].correctAnswer) {
      totalCorrect++;
    }
  }
  // Показать результат пользователя в текстовом поле на экране
  displayTotal = totalCorrect + "/" + questionsArray.length;
}

// КОД НА ДИНАМИЧЕСКИ ГЕНЕРИРУЕМЫХ КНОПКАХ ОТВЕТА
// Клипы ответов генерируются динамически и именуются последовательно
// "answer0", "answer1",..."answern". В каждом клипе ответа есть кнопка,
// при щелчке по которой проверяется имя клипа ответа, в котором она
// находится, чтобы определить ответ пользователя.
on (release) {
  // Обрезать префикс "answer" имени клипа
  choice = _name.slice(6, _name.length);
  _root.answer(choice);
}

// КОД В КАДРЕ quizEnd
gradeUser();
```

Содержимое файла *questionsArray.as* приводится ниже:

```
// КОД В ФАЙЛЕ questionsarray.as
// -------------------------------------------------
// Содержит массив объектов вопросов, которые заполняют  вопросник с
// вариантами ответов.
// Новые объекты вопросов составляются согласно
// следующему образцу.

/************** ПРИМЕР ОБЪЕКТА ВОПРОСА ***************
  // Вызвать конструктор Question с тремя аргументами:
  // начинающийся с нуля номер правильного ответа,
  // строка с текстом вопроса и
  // массив с вариантами ответов
  new Question
  (
    1,
    "question goes here?",
    ["answer 1", "answer 2", "answer 3"]
  )
*****************************************************/
```

```
// Не забудьте поставить запятую после каждого объекта массива,
// кроме последнего
questionsArray = [new Question (2,
  "Which version of Flash first introduced movie clips?",
  ["version 1", "version 2", "version 3",
  "version 4", "version 5", "version 6"]),

  new Question (2,
    "When was ActionScript formally declared a scripting language?",
    ["version 3", "version 4", "version 5"]),

  new Question (1,
    "Are regular expressions supported by Flash 5 ActionScript?",
    ["yes", "no"]),

  new Question (0,
    "Which sound format offers the best compression?",
    ["mp3","aiff", "wav"]),

  new Question (1,
    "True or False: The post-increment operator (++) returns the
    value of its operand + 1.",
    ["true", "false"]),

  new Question (3,
    "Actionscript is based on...",
    ["Java", "JavaScript", "C++", "ECMA-262", "Perl"])];
```

# 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 следующей главе рассказывается о «лексической структуре» (мелких деталях синтаксиса ActionScript). В главе, следующей за ней, мы рассмотрим ряд более сложных тем. Наконец, останутся часть II «Применение ActionScript» и часть III «Справочник по языку».

# 14

# Лексическая структура

*Лексическая структура* (*lexical structure*) языка – это набор правил, управляющих его синтаксическим составом. Этим правилам необходимо следовать при написании исходного кода сценариев.

## Пробельные символы

Символы табуляции, пробела и возврата каретки (т. е. переноса строки) используются в ActionScript так же, как и в английском языке: чтобы отделять слова друг от друга и не позволять им сливаться. На жаргоне программистов такие символы называются пробельными (*whitespace*) 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  // Ай! Нет пробельного символа между ключевым словом var и переменной x.
var x  // Теперь лучше...благодаря пробельному символу интерпретатор
       // сможет прочесть код.
```

Пробельный символ не обязателен, если есть какой-то другой разделитель (*delimiter*), сообщающий ActionScript о конце одной лексемы и начале другой. Следующий код является совершенно законным, 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, такие как квадратные, круглые или фигурные скобки, запятые, символы «больше» или «меньше», действующие как разделители. Следующие предложения кода совершенно законны, несмотря на вызываемую клаустрофобию:

```
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);};
```

Лишние пробельные символы являются делом вкуса, поскольку они игнорируются интерпретатором ActionScript. Несмотря на это, существуют соглашения, которых нужно придерживаться, чтобы сделать код более читаемым. Например, предыдущее выражение присваивания можно переписать и так, но очевидно, что прочесть его становится труднее:

```
x =
  10
  + 5
  / y;
```

Обратите внимание, что предложение завершается точкой с запятой, а не переносом строки. Почти всегда переносы строк несущественны и не выступают в качестве завершающих предложение символов. Поэтому часто один или более переносов строки (вместе с пробелами или символами табуляции) используются для облегчения чтения 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"]];
```

Если нужно разбить предложение на несколько строк, никаких особых действий не требуется. Просто добавьте возврат каретки и продолжайте ввод. Полезные советы относительно того, как улучшить читаемость кода, можно найти в книге Стива Макконелла (Steve McConnell) «Code Complete» (Microsoft Press). Однако, как будет видно из следующего раздела, бывают случаи, когда перенос строки может ошибочно интерпретироваться как окончание предложения.

Обратите внимание на то, что пробел *внутри* кавычек, в которые заключена строка, является существенным, 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 предложение (точка с запятой)

Как было сказано в главе 6 «Предложения», точка с запятой завершает предложение ActionScript. Хотя принято всегда завершать предложение точкой с запятой, но, строго говоря, в ActionScript это не обязательно. Если точка с запятой опущена, интерпретатор пытается догадаться, где заканчивается предложение. Например:

```
// Такой формат предпочтительнее
var x = 4;
var y = 5;
// Но такой тоже допускается
var x = 4
var y = 5
```

Интерпретатор ActionScript предполагает, что в предшествующем коде перенос строки предназначен служить завершением предложения. (В более строгих языках типа C компилятор сообщит об ошибке.) Однако пропуск точки с запятой в предложениях кода подобен пропуску точек в предложениях при обычном письме: большинство предложений будет, вероятно, понятно читателю, но в некоторых случаях все равно будет путаница, не говоря уже о том, что чтение потребует больших усилий. Посмотрим, например, что произойдет, если опустить точку с запятой после предложения `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 отдельное ключевое слово `return` является допустимым предложением. Даже если `return` находится на строке в одиночестве,

но после value + 1 добавлена точка с запятой, предложение return рассматривается как законченное.

Чтобы избежать такого рода неоднозначности, полезно ставить точку с запятой. Более того, в данном конкретном случае предложения return не отделяйте ключевое слово return от его выражения разрывом строки, поскольку это изменяет значение предложения. Поэтому предшествующий пример должен быть записан так:

```
function addOne (value) {
  return value + 1;
}
```

Обратите внимание, что точка с запятой завершает отдельные предложения, но не требуется там, где есть разделители блока предложений (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 не разрешает использовать точку с запятой, и при ее наличии возникает ошибка. Смотрите описание #include в части III «Справочник по языку».

# Комментарии

Так называемые *комментарии* являются текстом, который игнорируется интерпретатором и вводится программистом исключительно для удобства людей, читающих код. Комментарии можно свободно использовать для объяснения действий, выполняемых кодом, предоставления пользователю информации для контроля за версиями или других важных сведений, 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, следующий комментарий бесполезен:

```
// Присвоить i значение 5
i = 5;
```

в то время как следующий комментарий объясняет, почему i присваивается 5, что помогает понять логику кода:

```
// Инициализировать счетчик, используемый для поиска
// строки пароля, начиная с индекса 5
var i = 5;
```

Обратите внимание, что можно значительно приблизиться к созданию «самодокументирующегося кода», если использовать содержательные имена переменных и обработчиков. 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 на протяжении этой книги, начинаются с двух символов косой черты (//):

```
// Человеческий контакт...его интерпретатор никогда не прочтет
```

Однострочные комментарии автоматически завершаются переносом строки. Чтобы поместить дополнительный текст комментария на следующей строке, нужно повторить символы //:

```
// Здесь начало комментария...
// ...а здесь его продолжение.
```

Однострочные комментарии могут следовать за действительным кодом на той же строчке:

```
var x;    // Это допустимый комментарий на одной строчке с кодом
```

ActionScript поддерживает и многострочные комментарии, используемые для размещения крупных блоков комментариев к коду и *не* завершающиеся символом переноса 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, 2000
   ------END VERSION CONTROL INFO-----
*/
```

Многострочными комментариями можно пользоваться только в режиме Expert Mode редактора ActionScript Flash 5 или во внешних файлах *.as*, подключаемых с помощью директивы `#include`. Многострочные комментарии преобразуются в однострочные при переключении из режима Expert Mode в режим Normal Mode. Обратите внимание, что разрешены вложенные комментарии:

```
/* Это
/* вложенный // комментарий */
```

Использовать комментарий следующего вида не принято, но он допустим:

```
/* За этим комментарием следует действительный код */ var x = 5;
```

Распространена практика включения в комментарий инициалов и даты изменения при внесении в код крупных (или даже мелких) исправлений, например:

```
// Исправлена ошибка в финансовых расчетах. BAE 12-04-00. V1.01
```

Комментарии позволяют временно отключить код (или отключить устаревший код навсегда), не удаляя его. Код, отключенный путем превращения его в комментарий, называют *закомментированным* (*commenting out*). Такой код легко восстанавливается путем удаления ограничителей комментариев:

```
/* Отключу этот блок кода, пока не передумаю
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* используются интерпретатором ActionScript для обозначения специфических встроенных элементов языка, например, предложений и операторов. Они зарезервированы для использования интерпретатором, и нужно избегать использования их в качестве идентификаторов в собственном коде. Использование зарезервированных слов в иных целях, чем внутренне зарезервированные, в большинстве случаев приводит к ошибкам. Зарезервированные слова ActionScript перечислены в табл. 14.1.

Следует также избегать использования ключевых слов, перечисленных в табл. 14.2. Они не входят в состав ActionScript во Flash 5, но мо-

гут стать частью языка в будущем, так как запланированы для возможного использования стандартом ECMA-262.

*Таблица 14.1. Зарезервированные слова ActionScript*

| | | | |
|---|---|---|---|
| add* | for | lt* | tellTarget* |
| and* | function | ne* | this |
| break | ge* | new | typeof |
| continue | gt* | not* | var |
| delete | if | on | void |
| do | ifFrameLoaded* | onClipEvent | while |
| else | in | or* | with |
| eq* | le* | return | |

* Зарезервированные слова Flash 4, устаревшие в Flash 5.

*Таблица 14.2. Возможные в будущем зарезервированные слова*

| | | |
|---|---|---|
| abstract | extends | private |
| boolean | final | protected |
| byte | finally | public |
| case | float | short |
| catch | goto | static |
| char | implements | super |
| class | import | switch |
| const | instanceof | synchronized |
| debugger | int | throws |
| default | interface | transient |
| double | long | try |
| enum | native | volatile |
| export | package | |

Помимо формально определенных ключевых слов, следует избегать использования в коде названий встроенных свойств, методов и объектов, чтобы не изменить режим их действия, установленный по умолчанию. Например:

```
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 содержать только буквы (A–Z или a–z), цифры, символ подчеркивания и знак доллара. Особенно следите за тем, чтобы в идентификаторах не использовались пробелы, точки, обратная косая черта и прочие знаки пунктуации.
- Идентификаторы должны начинаться с буквы, символа подчеркивания или знака доллара (но не с цифры).
- Идентификаторы не должны совпадать с зарезервированными словами.

Хотя это не обязательное требование, но хорошим стилем является следование предшествующим правилам и при составлении имен экземпляров клипов, меток кадров и имен слоев.

# Чувствительность к регистру

Если язык полностью чувствителен к регистру букв, то все лексемы этого языка, включая идентификаторы и ключевые слова, должны вводиться с правильным указанием заглавных букв. Например, в языках, чувствительных к регистру, предложение

```
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, два следующих предложения объявляют две различные переменные (одну с именем `firstName`, а другую с именем `firstname`):

```
var firstName = "doug";
var firstname = "terry";
```

Спецификация ECMA-262, на которой базируется ActionScript, требует полной зависимости от регистра. Однако ActionScript отходит от стандарта в области чувствительности к регистру, чтобы обеспечить обратную совместимость с фильмами Flash 4. В ActionScript ключевые слова чувствительны к регистру, а идентификаторы – нет. Например, в следующем коде ключевое слово `onClipEvent` записано с неправильным выделением заглавных букв в виде `onclipevent`, что вызывает ошибку:

```
onclipevent (enterFrame)  // Должно быть onClipEvent (enterFrame)
```

Однако, в отличие от ключевых слов, идентификаторы в ActionScript не чувствительны к регистру, поэтому следующие предложения 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 вызвала бы ошибку в JavaScript, но благополучно создаст массив в ActionScript:

```
myList = new array();   // Предпочтительнее запись new Array();
```

Это может стать источником проблем при переносе кода JavaScript в ActionScript или на первых порах при изучении программистами JavaScript языка ActionScript. Чувствительность к регистру часто приводит в JavaScript к использованию одного и того же имени с разными целями. Например, предложение

```
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 отключит встроенный класс *Date*. Поэтому необходимо следить за тем, чтобы не один только регистр отличал идентификаторы один от другого или от всех предопределенных идентификаторов типа *Date* или *Array*. Вот как можно переписать предыдущий код в ActionScript:

```
myDate = new Date(); // Так надо писать в ActionScript
```

Главное – во всех случаях стремиться к единообразию, даже если язык строго не требует этого. Единообразное выделение заглавными буквами переменных, функций, экземпляров и других элементов облегчает чтение кода и не даст возможным будущим изменениям в правилах ActionScript, касающихся использования регистров, разрушить наши труды.

# Вперед!

Что касается основ языка ActionScript, то мы добрались до конца пути. Теперь вы полностью экипированы для чтения и написания на ActionScript. В следующей главе освещаются некоторые более сложные и доступные лишь посвященным темы, на которые мы не стали отвлекаться в предшествующем изложении.

# 15

# Более сложные темы

В данной главе собран ряд более сложных приемов и вопросов программирования на ActionScript.

## Копирование, сравнение и передача данных

Существует три основных способа обращения с данными. Можно копировать (*copy*) данные (например, присваивать значение переменной `x` переменной `y`), можно сравнивать (*compare*) данные (например, проверять равенство переменных `x` и `y`) и можно передавать (*pass*) данные (например, передавать переменную в функцию в качестве аргумента). Значения данных элементарных типов копируются, сравниваются и передаются совсем иначе, чем значения данных составных типов. Когда элементарные данные копируются в переменную, эта переменная получает свой собственный уникальный и личный экземпляр данных, независимо хранимый в памяти. 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 для

переменной, сохраняется только *ссылка* на данные (а не фактические данные). Эта ссылка сообщает интерпретатору, где хранятся фактические данные (т. е. 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 было бы весьма неэффективно копировать большие массивы и другие составные типы данных. Но для кода программы это имеет важные последствия. Когда нескольким переменным в качестве значения присваивается один и тот же кусок составных данных, каждая из них *не* хранит уникальный экземпляр данных (как произошло бы в случае элементарных данных). Напротив, существует лишь один экземпляр данных, а все переменные указывают на него. Если изменяется значение данных, обновляются *все* переменные.

Посмотрим, какое влияние это оказывает на практическое приложение. 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 равен 12. Но у y собственная копия значения 12, отличная от копии в x. Если изменить значение x, на значении y это не отразится:

```
x = 15;
trace(x); // Выводит 15 в окно Output
trace(y); // Выводит 12 в окно Output
```

Значение y не изменилось в результате изменения x, потому что при присвоении x переменной y последняя получила собственный экземпляр числа 12 (т. е. *элементарных* данных, содержащихся в x).

Теперь попробуем то же самое осуществить 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, что, поскольку x ссылается на массив, являющийся составным элементом данных, значение x является не буквально мас-

сивом ["first element", 234, 18.5], а просто ссылкой на этот элемент данных. 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")
```

Расширим 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, потому что 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 одно и то же составное данное. Однако посмотрим, что происходит при копировании ссылки 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.

Элементарные и составные данные по-разному обрабатываются и при передаче в функции, о чем говорилось в разделе «Элементарные и составные значения параметров» главы 9 «Функции». Самое важное, что при передаче элементарной переменной в качестве аргумента в функцию любые изменения этого элемента данных, произведенные функцией, не отражаются на исходной переменной. Однако при передаче составной переменной изменения в функции *влияют* на исходную переменную. То есть, если передать в функцию целочисленную переменную x, то изменения, производимые с ней в функции, не влияют на ее первоначальное значение. Но если передать в функцию массив y, любые изменения этого массива внутри функции изменят исходное значение y вне функции (потому что изменения в массиве оказывают воздействие на данные, на которые указывает y).

# Поразрядное программирование

Переключимся и рассмотрим не связанную с предыдущим эзотерическую тему: поразрядное программирование и поразрядные операторы. Обсуждение этой темы было малопонятным для главы 5 «Операторы», но сейчас мы осветим этот вопрос для опытных программистов и храбрых новичков.

Чтобы оптимальным способом работать с рядом опций, можно пользоваться поразрядными операторами. Формально поразрядные операторы являются математическими, но обычно они используются в логическом, а не в математическом контексте. Поразрядные операторы могут обращаться к отдельным двоичным цифрам (*битам*) целого числа. Чтобы понять механизм их действия, необходимо знать о том, как числа представляются в двоичном формате.

Двоичное число хранится в виде последовательности нулей и единиц, представляющих число в системе исчисления по основанию 2 (т. е. в *двоичной системе*). Каждая цифра числа представляет собой основание числовой системы, возведенное в некоторую степень. Двоичные числа используют в качестве основания число 2, поэтому первые четыре цифры двоичного числа представляют, справа налево, 1 ($2^0$), 2 ($2^1$), 4 ($2^2$) и 8 ($2^3$). Вот несколько примеров двоичных чисел с разъяснением

того, как по значениям их цифр можно вычислить эквивалентное число в десятичной системе счисления (десятичное число):

```
1     // Число 1 в десятичной системе: (1 x 1) = 1
10    // Число 2 в десятичной системе: (1 x 2) + (0 x 1) = 2
11    // Число 3 в десятичной системе: (1 x 2) + (1 x 1) = 3
100   // Число 4 в десятичной системе: (1 x 4) + (0 x 2) + (0 x 1) = 4
1000  // Число 8 в десятичной системе: (1 x 8) + (0 x 4) + (0 x 2) + (0 x 1) = 8
1001  // Число 9 в десятичной системе: (1 x 8) + (0 x 4) + (0 x 2) + (1 x 1) = 9
```

В двоичных числах *цифры*, о которых мы говорим, называют битами (*bits*, от «binary digit»). Например, четырехбитовое число состоит из четырех цифр (каждая из которых может быть нулем или единицей). Самый правый бит считается битом 0, слева от него находится бит 1 и т. д. Ниже приведено 8-битовое число с единицей в битах 0, 6 и 7; номера бит проставлены над числом:

```
бит: 76543210
     11000001
```

Как и во всех системах исчисления, наибольшее значение для каждой цифры на единицу меньше основания. Например, в числах десятичной системы счисления (десятичных) наибольшей цифрой является 9. Поскольку мы используем в качестве основания 2, каждая двоичная цифра должна быть 0 или 1. Как известно, числа 1 и 0 эквивалентны булевым значениям `true` и `false`, поэтому очень удобно использовать двоичные числа как последовательность ключей, находящихся в положении «включено» или «выключено». Как раз это позволяют делать поразрядные операторы.

О бите, имеющем значение 1, говорят, что он установлен (*set*, т. е. включен или имеет значение `true`). О бите, имеющем значение 0, говорят, что он сброшен (*cleared*, т. е. выключен или имеет значение `false`). Биты иногда представляют как флажки (*flag*) или ключи (*switch*), имея в виду, что они указывают на нечто, способное находиться в одном из двух состояний (включено/выключено или true/false).

В поразрядном программировании почти всегда есть ситуации, в которых ряд свойств могут быть включены или выключены. С помощью поразрядных операторов можно в компактном виде представить несколько опций в одном числовом значении вместо использования нескольких переменных. В результате достигаются большая производительность и меньший расход памяти.

Представим себе, что с помощью Flash разрабатывается сайт для продажи автомобилей. Простоты ради примем, что на продажу выставлена только одна марка автомобилей, но пользователь может выбрать для своей машины любую комбинацию четырех опций: кондиционер, CD-плеер, крыша с окошком и кожаные сиденья. Задачей нашей программы Flash будет получить суммарную стоимость машины вместе со

всеми опциями, а задачей серверной программы – сохранить эту информацию как часть настроек пользователя.

Можно хранить опции оборудования автомобиля в четырех отдельных булевых переменных, например:

```
var hasAirCon = true;
var hasCD = true;
var hasSunRoof = true;
var hasLeather = true;
```

По существу, есть четыре ключа – по одному для каждого дополнительного компонента оборудования автомобиля, для каждого из которых нужна переменная. Все прекрасно, но это означает, что необходимы четыре переменные в памяти и четыре поля в базе данных с настройками пользователей на сервере. Если же записать выбор дополнительного оборудования в виде отдельных двоичных цифр, можно сохранить все четыре опции в одном 4-битовом числе: бит 0 (цифра 1) для кондиционера, бит 1 (цифра 2) для CD-плеера, бит 3 (цифра 4) для крыши с окном и бит 3 (цифра 8) для кожаных сидений. Вот несколько примеров конфигураций, показывающих, как одно число может представлять любую комбинацию из четырех опций:

```
var options;
options = 1   // двоичное 0001: бит 0 включен: только кондиционер
options = 2   // двоичное 0010: бит 1 включен: только CD-плеер
options = 4   // двоичное 0100: бит 2 включен: только крыша с окном
options = 8   // двоичное 1000: бит 3 включен: только кожаные сиденья

// А теперь комбинирование опций
options = 5   // двоичное 0101: кондиционер (1) и крыша с окном (4)
options = 10  // двоичное 1010: CD-плеер (2) и кожаные сиденья (8)
options = 15  // двоичное 1111: полный комплект!
```

Если нужно добавить или удалить опции, прибавляем или вычитаем значение соответствующего бита:

```
var options = 0;  // Сначала ничего дополнительного
options += 4;     // Добавить крышу с окном (options = 4, или 0100)
options += 1;     // Добавить кондиционер (options = 5, или 0101)
options += 2;     // Добавить CD-плеер (options = 7, или 0111)
options -= 4;     // Убрать крышу с окном (options = 3, или 0011)
options += 8;     // Добавить кожаные сиденья (options = 11, или 1011)
```

Теперь мы умеем хранить несколько опций в виде последовательности битов в одном числе. Как получить значения этих битов для расчета стоимости автомобиля? Нужно использовать поразрядные операторы. Сначала рассмотрим все операторы, а потом вернемся к этому примеру с машинами.

## Поразрядное «И»

Оператор поразрядного «И» (&) объединяет биты двух операндов, выполняя операцию поразрядного «И» над каждым разрядом чисел. Результат комбинирования битов возвращается как число.

Поразрядное выражение «И» имеет вид:

```
operand1 & operand2
```

Операндами поразрядного «И» могут быть любые числа, но перед операцией они преобразуются в 32-разрядные двоичные целые. Если у операнда есть дробная часть, она отбрасывается.

Обратите внимание на то, что *поразрядный* оператор «И» пользуется одним символом оператора `&` и действует над отдельными битами в операндах, в то время как *логический* оператор «И», обсуждавшийся в главе 5, обозначается двумя символами `&&` и рассматривает каждый операнд как единое целое.

Поразрядное «И» возвращает число, значение которого определяется в результате сравнения отдельных битов в числовых операндах *operand1* и *operand2*. Если некоторый бит в обоих операндах содержит единицу, то в результате соответствующему биту будет тоже присвоена единица, в противном случае этот бит будет установлен в 0.

Операции поразрядного «И» легче всего изобразить, расположив двоичные представления десятичных операндов вертикально друг под другом и выровняв колонки разрядов. В таком представлении легко найти биты операндов, в обоих из которых содержится 1.

В данном примере в обоих операндах бит 2 (третий бит справа) содержит 1 в обоих операндах, и поэтому ему в результате присваивается 1. Остальные биты результата устанавливаются в 0:

```
  1111
& 0100
------
  0100
```

В следующем примере биты 0 и 3 равны 1 в обоих операндах, и поэтому устанавливаются в 1 в результате. Биты 1 и 2 устанавливаются в результате в 0:

```
  1101
& 1011
------
  1001
```

В ActionScript используются десятичные числа (числа десятичной системы счисления), а не двоичные, что затрудняет зрительное представление поразрядных операций. Вот как предыдущие операции выглядят в реальном коде:

```
15 & 4   // Результат 4
13 & 11  // Результат 9
```

На практике оператор поразрядного «И» используется для проверки установки отдельного флага или группы флагов (т. е. битов) в `true` или `false`.

В следующем примере проверяется, установлен ли бит 2 (имеющий значение 4) в `true`:

```
if (x & 4) {
  // Сделать что-то
}
```

Можно проверить, установлен ли хотя бы один из битов 2 или 3 (имеющий значение 8) в `true`:

```
if (x & (4|8)) {
  // Сделать что-то
}
```

Обратите внимание, что для проверки установки бита 2 *или* бита 3 используется оператор, который будет обсуждаться ниже. Для проверки того, что оба бита – 2 *и* 3 заданы, можно воспользоваться следующим кодом:

```
if (x & (4|8) == (4|8)) {
  // Сделать что-то
}
```

Оператор поразрядного «И» используется также для присваивания отдельным битам числа значения `false`. Смотрите раздел о поразрядном операторе «НЕ» ниже в этой главе.

## Поразрядное «ИЛИ»

Оператор поразрядного «ИЛИ» (`|`) объединяет биты двух чисел, выполняя операцию логического «ИЛИ» над каждым битом операндов. Подобно поразрядному «И», поразрядное «ИЛИ» возвращает результат комбинирования в виде числа. Выражение поразрядного «ИЛИ» имеет вид:

```
operand1 | operand2
```

Операнды могут быть любыми числами, но перед операцией преобразуются в 32-разрядные двоичные числа. Если у операнда есть дробная часть, она отбрасывается.

Обратите внимание на то, что *поразрядный* оператор «ИЛИ» обозначается одним символом оператора | и действует над отдельными битами в операндах, в то время как *логический* оператор «ИЛИ», обсуждав-

шийся в главе 5, обозначается двумя символами || и рассматривает каждый операнд как единое целое.

Каждый бит результата получается в результате применения логического «ИЛИ» к битам двух операндов. Поэтому, если бит установлен в 1 в одном (или обоих) из *operand1* или *operand2*, то в результате этот бит будет установлен в 1. Сравните следующие псевдопримеры с приведенными выше для оператора поразрядного «И».

В приведенном ниже примере только бит 1 устанавливается в результате в 0 , потому что бит 1 равен 0 в обоих операндах. Остальные биты устанавливаются в 1:

```
  1101
| 0100
------
  1101
```

В следующем примере все биты результата устанавливаются в 1, потому что каждый бит содержит 1 хотя бы в одном из двух операндов:

```
  1101
| 1011
------
  1111
```

В реальном коде это записывается так:

```
13 | 4    // Результат равен 13
13 | 11   // Результат равен 15
```

На практике оператор поразрядного «ИЛИ» часто используется для объединения нескольких чисел, представляющих отдельные опции, в одно числовое значение, которое представляет все опции системы. Например, следующий код объединяет бит 2 (значение 4) с битом 3 (значение 8):

```
options = 4 | 8;
```

Оператор поразрядного «ИЛИ» используется также для установки в `true` некоторой опции в существующем значении. Следующий пример устанавливает опцию, представляемую битом 3 (значение 8) в `true`. Если значение в бите 3 уже `true`, оно остается в неприкосновенности:

```
options = options | 8;
```

Можно также одновременно задать несколько битов:

```
options = options | 4 | 8;
```

## Поразрядное XOR («исключающее ИЛИ»)

Начинаем официально изучать странные символы пунктуации, используемые в качестве операторов. Оператором поразрядного XOR (eXclusive OR – «исключающее ИЛИ») служит знак вставки, или «крышка», ^ (на большинстве клавиатур вводится как Shift-6). Выражение поразрядного «исключающего ИЛИ» имеет вид:

```
operand1 ^ operand2
```

Операнды могут быть любыми числами, но перед операцией преобразуются в 32-разрядные двоичные числа. Если у операнда есть дробная часть, она отбрасывается.

Оператор «исключающего ИЛИ» отличается от оператора поразрядного «ИЛИ» тем, что для каждого разряда, содержащего 1 в *обоих* операндах, в результате будет 0, а не 1. Иными словами, результат «исключающего ИЛИ» содержит 0 для всех битов, которые одинаковы в обоих операндах, и содержит 1 для всех битов, различных в двух операндах.

В следующем примере биты 0 и 3 совпадают в обоих операндах, поэтому в результате эти биты устанавливаются в 0. Биты 1 и 2 различны в операндах, поэтому в результате они устанавливаются в 1:

```
  1011
^ 1101
------
  0110
```

В следующем примере все биты обоих операндов совпадают, поэтому результат состоит из одних нулей:

```
  0010
^ 0010
------
  0000
```

В следующем примере биты 0, 2 и 3 различны в двух операндах, поэтому в результате они устанавливаются в 1. Бит 1 одинаков в обоих операндах, поэтому в результате он устанавливается в 0:

```
  0110
^ 1011
------
  1101
```

В переводе на десятичные числа предшествующие примеры становятся такими:

```
11 ^ 13    // Результат 6
2 ^ 2      // Результат 0
6 ^ 11     // Результат 13
```

Оператор «поразрядного ИЛИ» обычно используется для переключения опций между 1 и 0 (`true` и `false`). Чтобы переключить опцию, задаваемую битом 2 (значение которого 4), можно выполнить код:

```
options = options ^ 4;
```

## Поразрядное «НЕ»

В отличие от поразрядных «И», «ИЛИ» и «исключающего ИЛИ», получающих результирующее число из двух других, поразрядное «НЕ» изменяет биты одного числа. Он использует символ «тильда» (~), находящийся на большинстве клавиатур в левом верхнем углу, и имеет вид:

```
~operand
```

Операнд может быть любым числом, но перед операцией он преобразуется в 32-разрядное двоичное число. Если у операнда есть дробная часть, она отбрасывается.

Поразрядное «НЕ» просто инвертирует биты своего операнда. Например:

```
~00000000000000000000000000000010
  // Результат 11111111111111111111111111111101

~11111111111111111111111111111010
  // Результат  00000000000000000000000000000101
```

В десятичном виде это записывается так:

```
~2    // Результат -3. См. разъяснение ниже.
~-6   // Результат 5. См. разъяснение ниже.
```

Здесь не место вдаваться в системы представления отрицательных двоичных чисел, но опытные программисты заметят, что поразрядные операции представляют отрицательные двоичные числа в дополнительном коде. Тем, кто незнаком с этой системой, следует просто запомнить, что значение, возвращаемое поразрядной операцией «НЕ», на единицу меньше значения, получаемого изменением знака исходного операнда на противоположный, например:

```
~-10  // Поменять знак у -10 и вычесть 1. Результат 9.
```

Оператор поразрядного «НЕ» обычно используется вместе с оператором поразрядного «И» для сброса определенных битов (т. е. присваивания им 0). Например, для сброса бита 2 можно выполнить:

```
options = options & ~4;
```

Выражение `~4` возвращает 32-разрядное целое, состоящее из сплошных единиц, кроме 0 в разряде 2. Поразрядное «И» этого числа с пере-

менной options сбрасывает бит 2 и оставляет все остальные биты неизменными. Более кратко предыдущий пример можно записать так:

```
options &= ~4;
```

Таким же способом можно одновременно сбросить несколько битов. Вот, например, как сбросить биты 2 и 3:

```
options &= ~(4 | 8);
```

# Поразрядные операторы сдвига

Мы уже видели, что поразрядное программирование рассматривает двоичные числа как последовательность ключей (*switches*). Часто оказывается удобным перемещать эти ключи. Например, если включен бит 0 и нужно выключить его, а бит 2 включить, можно просто передвинуть бит 0 влево на две позиции. Либо, если требуется узнать, задан ли бит 5, можно сдвинуть его вправо на 5 позиций и проверить значение бита 0. Такие перемещения могут осуществляться поразрядными операторами сдвига.

Поразрядные операторы сдвига позволяют также быстро выполнять умножение и деление на степени двойки. Если нужно разделить десятичное число (число десятичной системы счисления) на 10, можно просто сдвинуть десятичную точку на одну позицию влево. Аналогично, для умножения на 10 десятичная точка сдвигается на одну позицию вправо, а для умножения на $10^3$ (т. е. 1000) следует переместить десятичную точку на три позиции вправо. Поразрядные операторы сдвига позволяют осуществлять аналогичные операции над двоичными числами. При сдвиге битов вправо число делится на 2 для каждого перемещения на одну позицию. При сдвиге битов влево число умножается на 2 для каждого перемещения на одну позицию.

## Сдвиг вправо со знаком

Оператор сдвига вправо со знаком можно использовать для деления числа на степень двойки. Он использует символ >> (два знака больше подряд) и имеет общий вид:

```
operand >> n
```

где *n* задает число позиций, на которое сдвигаются вправо биты *operand*. Результат эквивалентен делению operand на $2^n$. Если есть остаток, он отбрасывается. Рассмотрим, как действует этот оператор.

Все биты сдвигаются вправо на число позиций, заданное *n*. Биты, выдвигаемые за правый край числа, отбрасываются. Для заполнения пустого места, создаваемого операцией сдвига, с левого края добавляются новые биты. Если *operand* положительный, добавляются нули. Если *operand* отрицательный, добавляются единицы (потому что отри-

цательные числа представляются в дополнительном коде). Вот пример на псевдокоде:

```
// Самый правый бит (0) теряется, а слева вводятся биты 0
// Реузльтат 00000000000000000000000000000100
00000000000000000000000000001000 >> 1
```

Сдвиг числа вправо на один бит равносилен делению на $2^1$ (т. е. 2). В десятичном виде это записывается так:

```
8 >> 1  // Результат 4
```

Обратите внимание, что остаток отбрасывается:

```
9 >> 1 // Результат все равно 4
```

Если числа отрицательные, >> все равно делит на 2 при сдвиге на каждый бит:

```
-16 >> 2  // Результат -4 (-16 делится на 2 в квадрате)
```

## Сдвиг вправо без знака

Оператор беззнакового сдвига вправо записывается с помощью трех последовательных знаков «больше» (>>>) и имеет вид:

```
operand >>> n
```

Он работает так же, как оператор сдвига вправо со знаком, но освобождаемые при сдвиге разряды всегда заполняются нулями (независимо от того, положителен *operand* или отрицателен). Для положительных чисел он не отличается от оператора сдвига вправо со знаком.

## Сдвиг влево

Оператор сдвига влево может использоваться для умножения числа на некоторую степень 2. Он записывается с помощью символа << symbol (состоящего из двух символов «меньше» подряд) и имеет общий вид:

```
operand1 << n
```

где *n* задает число позиций, на которое сдвигаются влево биты *operand*. Результат эквивалентен умножению operand на $2^n$. Вот как он действует.

Все биты сдвигаются влево на число позиций, заданное *n*. Биты, выдвигаемые за левый край числа, отбрасываются. Пустые биты, образующиеся справа при сдвиге, заполняются нулями. Например:

```
01000000000000000000000000001001 << 4
  // Результат: 00000000000000000000000010010000
```

Сдвиг числа влево на 4 бита эквивалентен умножению его на $2^4$ (т. е. 16). В десятичном виде это записывается так:

```
9 << 4  // В результате получается 9 * 16, т. е. 144
```

Заметьте, что в предшествующих примерах значение, связанное с конкретным битом, вычислялось «вручную»: 1 для бита 0, 2 для бита 1, 4 для бита 2, 8 для бита 3 и т. д. Оператор левого сдвига очень удобен для вычисления значения битовой позиции:

```
(1 << 0)    // Бит 0 равен 1
(1 << 1)    // Бит 1 равен 2
(1 << 2)    // Бит 2 равен 4
(1 << 3)    // Бит 3 равен 8
(1 << 15)   // Куда проще, чем помнить, что бит 15 равен 32768!
```

Оператор сдвига влево также удобен для динамического выбора бита по числовому индексу вместо значения бита. В примере 15.1 подсчитывается количество единиц в числе.

*Пример 15.1. Использование сдвига влево для подсчета установленных битов*

```
myNumber = 27583;  // Число, количество 1 в котором будет посчитано
count = 0;
for (var i=0; i < 32; i++) {
  if (myNumber & (1 << i)) {
    count++;
  }
}
```

Пример 15.2 является разновидностью примера 15.1, использующей оператор сдвига вправо. Вместо левого сдвига, применяемого для получения значения, ассоциируемого с каждым битом, можно повторять правый сдвиг и проверять самый правый бит (бит 0).

*Пример 15.2. Подсчет числа заданных бит с помощью сдвига вправо*

```
myNumber = 27583;
count = 0;
temp = myNumber;    // Создать копию для временного использования
for (var i = 0; i < 32; i++) {
  if (temp & 1) {
    count++;
  }
  temp = temp >> 1;
}
```

Переменная `myNumber` копируется во временную переменную `temp`, потому что иначе она была бы разрушена сдвигом вправо. Переменная `temp` в конечном итоге становится равной 0.

## Применение поразрядных операций

Рассмотрение поразрядных операторов мы начали с Flash-сайта, на котором продаются автомобили. Изучив работу поразрядных операторов, можно теперь применить их в расчете стоимости автомобиля, как показано в примере 15.3. Файл *.fla* для этого примера можно загрузить из сетевого Хранилища кода.

*Пример 15.3. Практическое применение поразрядных операций*

```
// Сначала устанавливаем опции (обычно путем сложения и вычитания чисел,
// исходя из выбора на заполняемой форме, но здесь мы "прошиваем" их в код)
var hasAirCon   = (1<<0)    // Бит 0: 0 означает "нет", 1 означает "да"
var hasCDplayer = (0<<1)    // Бит 1: 0 означает "нет", 2 означает "да"
var hasSunRoof  = (1<<2)    // Бит 2: 0 означает "нет", 4 означает "да"
var hasLeather  = (1<<3)    // Бит 3: 0 означает "нет", 8 означает "да"

// Теперь объединим опции в одно число с помощью поразрядного "ИЛИ"
var carOptions = hasAirCon | hasCDplayer | hasSunRoof | hasLeather;

// Вот функция расчета цены
function totalPrice(carOptions) {
  var price = 0;
  if (carOptions & 1) {  // Если задан первый бит,
    price += 1000;       // прибавить $1000
  }
  if (carOptions & 2) {  // Если задан второй бит,
    price += 500;        // прибавить $500
  }
  if (carOptions & 4) {  // Если задан третий бит,
    price += 1200;       // прибавить $1200
  }
  if (carOptions & 8) {  // Если задан четвертый бит,
    price += 800;        // прибавить $800
  }

  return price;
}

// Все готово: вызовем функцию и проверим, работает ли она!
trace(totalPrice(carOptions));  // Возвращает 3000. Круто...
```

Чтобы избавиться от фиксации в коде значений битов, полезно запомнить битовые значения, соответствующие конкретным опциям, в переменных, например:

```
var airConFLAG   = 1 << 0;  // Бит 0, значение которого 1
var cdPlayerFLAG = 1 << 1;  // Бит 1, значение которого 2
var sunroofFLAG  = 1 << 2;  // Бит 2, значение которого 4
var leatherFLAG  = 1 << 3;  // Бит 3, значение которого 8
```

Упражнение для читателя: Переработайте пример 15.3, использовав переменные и оператор сдвига влево вместо жестко закодированных значений, представляющих опции.

### Зачем нужны поразрядные операции?

Хотя пример 15.3 было бы легче понять как последовательность булевых операций, поразрядные операции компактны и выполняются очень быстро. В каждом случае, когда удается говорить с компьютером на его родном двоичном языке, сберегается память и увеличивается скорость.

Для сравнения рассмотрим ситуацию, в которой запоминается конфигурационный файл пользователя, и у каждого пользователя есть 32 настройки, которые могут находиться во включенном или выключенном состоянии. В обычной базе данных для каждого пользователя потребовалось бы 32 поля. Для миллиона пользователей это составит миллион записей из 32 полей. Но при поразрядном программировании можно сохранить эти 32 настройки в одном числе, для чего требуется только одно поле в базе данных для каждого пользователя! В результате не только экономится дисковое пространство: при каждом обращении к файлу пользователя нужно передать только одно целое, а не 32 булевых значения. При обработке миллионов транзакций экономия нескольких миллисекунд на каждой транзакции может существенно повысить производительность системы.

Для более глубокого изучения вопроса ознакомьтесь с превосходной статьей Джина Майера (Gene Myers) для С-программистов «Becoming Bit Wise», расположенной по адресу: *http://www.cscene.org/CS9/CS9-02.html*.

## Более сложные вопросы областей видимости функций

В данном разделе излагается еще одна тема, слишком сложная для включения ее в начальное изложение областей видимости функций. Вернемся же к ней сейчас, будучи вооружены знаниями о клипах, областях видимости функций и объектах.

В главе 9 отмечалось, что цепочка областей видимости функций обычно определяется относительно предложения, объявляющего функцию. Однако есть расширение этого правила. Если функция из одной временной диаграммы присваивается переменной из временной диаграммы другого клипа, это *присваивание* оказывает воздействие и на цепочку областей видимости функции. Если исходная функция вызывается непосредственно, в ее цепочку областей видимости включается исходная временная диаграмма, но если функция вызывается через переменную, ее цепочка областей видимости включает в себя временную диаграмму *переменной*.

Допустим, например, что создана функция *transformClip( )*, которая поворачивает текущий клип и изменяет его масштаб. Угол поворота и коэффициент масштабирования устанавливаются в переменных `rotateAmount` и `widthAmount`:

```
var rotateAmount = 45;
var widthAmount = 50;

function transformClip () {
  _rotation = rotateAmount;
  _xscale = widthAmount;
}

// Вызвать функцию
transformClip();
```

После этого функция `transformClip` присваивается переменной `tc` в клипе с именем `rect`:

```
rect.tc = transformClip;
```

При вызове функции `transformClip` через `rect.tc`, как показано ниже, ничего не происходит:

```
rect.tc();
```

Почему? Цепочка областей видимости хранящейся в переменной `tc` функции содержит `rect`, а не временную диаграмму исходной функции, поэтому переменные `rotateAmount` и `widthAmount` не будут найдены. Но если добавить в `rect` переменные `rotateAmount` и `widthAmount`, функция сможет найти эти переменные, и сработает код:

```
rect.widthAmount = 10;
rect.rotateAmount = 15;
rect.tc();  // Устанавливает для rect ширину 10% и угол поворота 15%
```

Напротив, когда в присваивании функции участвуют обычные объекты данных на той же временной диаграмме, цепочка областей видимости присвоенной функции не изменяется и определена постоянно относительно объявления функции. Это демонстрирует пример 15.4.

*Пример 15.4. Фиксированная область видимости методов объектов*

```
// Установка значений переменных
var rotateAmount = 45;
var widthAmount = 50;

// Создаем функцию transformClip(), которая
// выводит значения rotateAmount и widthAmount
function transformClip () {
  trace(rotateAmount);
  trace(widthAmount);
}
```

```
// Создаем объект, соответствующий клипу rect из предыдущего примера
var rectObj = new Object();

// Копируем transformClip в свойство rectObj
rectObj.tc = transformClip;

// Задаем свойства rotateAmount и widthAmount объекта rectObj
rectObj.rotateAmount = 15;
rectObj.widthAmount = 10;

// Теперь вызываем rectObj.tc
rectObj.tc();  // Displays 45 and 50, not 15 and 10. The scope of
               // rectObj.tc is the same as transformClip().
```

Если функция присваивается свойству объекта, то ее областью видимости становится временная диаграмма, на которой находится объявление функции. Однако при присвоении функции удаленному клипу областью видимости этой функции становится временная диаграмма удаленного клипа.

Заметьте, что такое поведение фактически является отходом от JavaScript, в котором области видимости функции постоянны и определяются объектами, исходя из объявлений функций. Например, если предположить, что фреймы в HTML примерно аналогичны клипам фильма Flash, можно установить различие между ними: в JavaScript, функция, присвоенная удаленному фрейму, сохраняет в качестве области видимости фрейм с объявлением функции, а не удаленный фрейм, как в примере 15.5.

*Пример 15.5. Статическая область видимости функций JavaScript*

```
// КОД В ФРЕЙМЕ 0 НАБОРА ФРЕЙМОВ
// Присвоение переменной в фрейме 0
var myVar = "frame 0";

// Создать функцию в фрейме 1 и скопировать ее обратно в фрейм 0
parent.frames[1].myMeth = function () { alert(myVar); };
myMeth = parent.frames[1].myMeth;

// Вызвать myMeth() из фрейма 0
myMeth(); // Выводит "frame 0"

// КОД В КНОПКЕ ФРЕЙМА 1 ТОГО ЖЕ НАБОРА ФРЕЙМОВ
<FORM>
  <INPUT type="button" value="click me" onClick="myMeth();">
</FORM>

// Щелкнуть по кнопке для вызова myMeth(_ ) из фрейма 1.
// Выводится "frame 0".
// область видимости myMeth(_ ) не фрейм 1, а фрейм 0, где
// находится предложение объявления.
```

# Тип данных movieclip

Овладев основами ActionScript, вернемся к еще одной таинственной теме. В главе 13 «Клипы» объяснялось, что клипы ведут себя, по большей части, как объекты. Однако клипы – это не просто еще один класс, это самостоятельный тип данных. Гари Гроссман, создатель ActionScript, так разъясняет различие между внутренними реализациями типов данных *movieclip* и *object*:

> Клипы внутри проигрывателя реализованы отдельно от объектов, хотя и те и другие проявляют себя в ActionScript почти одинаковым способом. Главное различие между ними состоит в способе выделения памяти и ее освобождения. Для обычных объектов существуют *счетчики ссылок* и *сборка мусора*, а срок существования клипов определяется временной диаграммой или явно функциями *duplicateMovieClip( )* и *removeMovieClip( )*.
>
> Если объявить массив с помощью `x = new Array()`, а затем присвоить `x = null`, то ActionScript сразу определит, что ссылок на объект *Array* не осталось (нет переменных, которые на него ссылаются), и уберет его как мусор (т. е. освободит используемую им память). Периодическая процедура уборки мусора по принципу «найти и подмести» удаляет объекты, содержащие циклические ссылки. (То есть специальная технология позволяет освобождать память, занимаемую неиспользуемыми объектами, ссылающимися друг на друга.)
>
> Клипы ведут себя не так. Их создание и уничтожение зависят от расположения объектов на временной диаграмме. Если они создаются динамически (например, с помощью функции *duplicateMovieClip( )*), то уничтожаются только посредством функции *removeMovieClip( )*.
>
> Ссылки на объекты являются *указателями* (указывают адреса памяти); отслеживание ссылок и сборка мусора оберегают пользователя от висячих указателей и утечки памяти. Ссылки же на клипы являются *мягкими*: фактически ссылка содержит абсолютный путь к цели. Пусть есть клип с именем `foo`; если задать `x = foo` (что сделает `x` ссылкой на клип `foo`), затем удалить `foo` с помощью *removeMovieClip( )* и создать *другой* клип с именем `foo`, ссылка `x` снова окажется действующей (она будет указывать на новый клип `foo`).
>
> Обычные объекты ведут себя не так. Прежде всего, существование ссылки на объект не позволяет его удалить. Поэтому, если бы клипы были объектами, нельзя было бы удалить объект из памяти с помощью *removeMovieClip( )*, пока переменная `x` содержит ссылку на него. Кроме того, если создать второй клип с именем `foo`, то прежний `foo` и новый `foo` могут существовать одновременно, хотя старый `foo` больше не будет выводиться.
>
> Итак, отдельный тип *movieclip* оправдан, поскольку имеет большие отличия от типа *object*. По сходным причинам оператор *typeof* возвращает `"function"` для функций, несмотря на то, что функции во многих отношениях близки объектам.

# Вперед!

И на этом завершается часть I «Основы ActionScript». В части II «Применение ActionScript» будут изложены некоторые важные приемы практического создания документов, а также будет рассказано о создании форм Flash и об отладке кода. Чтобы свободно изъясняться на ActionScript, нужно расширять свой запас слов, просматривая часть III «Справочник по языку», где описаны встроенные функции, свойства, объекты и классы ActionScript.

# II

# Применение ActionScript

В этой части описываются некоторые более практические стороны программирования на Flash, такие как использование среды разработки и отладчика. В части II также рассматриваются две особые области программирования на Flash: создание форм Flash и использование экранных текстовых полей.

- Глава 16 «Среда разработки ActionScript»
- Глава 17 «Формы Flash»
- Глава 18 «Экранные текстовые поля»
- Глава 19 «Отладка»

# 16

# Среда разработки ActionScript

В этой главе изучаются практические стороны редактирования кода ActionScript. Будут рассмотрены следующие темы:

- Размещение кода на кнопках, клипах и кадрах с помощью панели Actions
- Загрузка кода из внешнего файла
- Оформление кода в виде компоненты многократного использования с помощью Smart Clips

## Панель Actions

Панель *Actions* – это среда редактирования ActionScript во Flash. Все сценарии фильма создаются в панели Actions, открываемой через пункт меню Window → Actions.

Панель Actions разделена на две части: окно инструментов (слева) и окно сценариев (справа), как показано на рис. 16.1.

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 (Frame Actions) или на кнопке или клипе (Object Actions). При выборе кадра заголовок панели Actions автоматически изменяется на

Frame Actions; при выборе клипа или кнопки заголовок панели Actions изменяется на Object Actions.

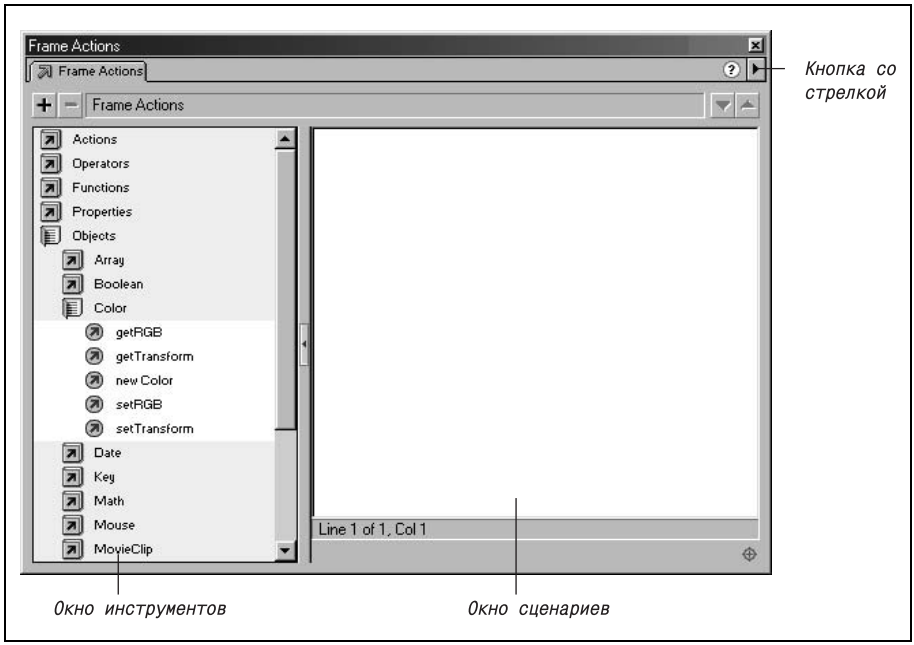


***Рис. 16.1.*** *Панель Actions*

Структура элементов в окне инструментов несколько отличается от категорий, использовавшихся при описании возможностей ActionScript в данной книге. Прежде всего, *предложения* не выделены в отдельную папку, а классы и объекты объединены вместе в папке *Objects*. В этой книге проводится различие между предложениями, классами и объектами, а также соблюдается более формальная терминология программирования.

## 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 предложения создаются с помощью окна инструментов, а окно сценариев используется для просмотра этих предложений. Чтобы поместить в окно сценариев новое предложение, находясь в обычном режиме, нужно выполнить двойной щелчок по требуемому элементу Action или перетащить его из окна инструментов в окно сценариев. Чтобы добавить предложение с произвольным ко-

дом, нужно выбрать действие *evaluate* и ввести предложение в поле Expression окна Parameters (на рис. не показано) в нижней части панели Actions.

После добавления предложения в окно сценариев, его можно настроить через окно Parameters в нижней части панели Actions. Структура и содержимое окна Parameters меняются в зависимости от предложения, выбранного в окне сценариев. Обратите внимание, что при работе панели Actions в обычном режиме нельзя вводить код в окно сценариев с клавиатуры. В обычном режиме окно сценариев служит не для редактирования текста, а предоставляет доступный только для чтения список предложений. Изменения вносятся в эти предложения с помощью окна Parameters.

Обычный режим Flash 5 похож на панель Actions во Flash 4, но не следует рассматривать его как режим, обеспечивающий обратную совместимость. Напротив, в обоих режимах редактирования можно создать код, который не будет совместим с Flash 4 (см. приложение C «Обратная совместимость»). Обычный режим удобен для начинающих программистов, но может создавать ограничения при разработке реальных проектов. Поэтому в данной книге для создания кода используется только режим эксперта. Однако, за исключением многострочных комментариев (которые можно использовать 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 осуществляют ввод 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 непосредственно вводятся в окно сценариев.

## Установка режима редактирования

Режим редактирования в панели Actions устанавливается индивидуально для каждого кадра и для каждого объекта. Это означает, что Flash запоминает режим редактирования панели Actions, выбираемый 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.

Выбор режима редактирования для каждого кадра или объекта позволяет осуществить кнопка со стрелкой в правом верхнем углу панели Actions (см. рис. 16.1). Чтобы выбрать режим по умолчанию для всех кадров и объектов фильма, нужно воспользоваться пунктом Expert

Mode или Normal Mode в меню Edit → Preferences → General → Actions Panel → Mode. Установка режима по умолчанию коснется только тех кадров и объектов, режим редактирования которых еще не задан. Поэтому режим, действующий по умолчанию, следует задавать в самом начале создания фильма. Возможность глобально переопределить индивидуальные режимы, явно установленные ранее для существующих кадров и объектов, отсутствует.

Переключение из режима эксперта в обычный разрушает имевшееся форматирование исходного кода. Новый формат кода будет соответствовать стандартным правилам форматирования во Flash: без лишних пробельных символов, с комментариями на отдельной строке, а для блоков текста устанавливается стиль отступов, который можно видеть в данной книге.

# Помещение сценариев в кадры

Структура документов Flash строится вокруг идеи анимации. Каждый документ Flash содержит линейную последовательность *кадров*, или кусочков видео- и аудиосодержимого. Большая часть кода ActionScript привязывается к этой кадровой структуре. Помещение блока кода в кадр обусловливает связь между временем запуска этого кода и воспроизведением 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.

Пусть, например, нужно написать код, который помещает на рабочий стол новый клип, когда выводится 20-й кадр фильма:

```
_root.attachMovie("myClip", "clip1", 0);
```

Чтобы этот код выполнился при выводе 20-го кадра фильма, он должен быть прикреплен к ключевому кадру 20-го кадра фильма. Чтобы прикрепить код к ключевому кадру, нужно выделить ключевой кадр на временной диаграмме, открыть панель Actions и ввести нужный код в окно сценариев. Во время воспроизведения фильма перед выводом содержимого кадра выполняется код, находящийся в ключевом кадре.

Код в ключевых кадрах используется 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 воспроизводящая головка доходит до кадра 15, выполняется этот код, и воспроизведение фильма продолжается опять 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 и 15 до бесконечности.

Однако код, находящийся в ключевых кадрах, не всегда управляет воспроизводящейголовкой или синхронизируется с ней. Ключевые кадры можно использовать просто как устройства хранения функций, переменных, объектов и других программных сущностей. Например, можно прикрепить к кадру 1 клипа функцию *moveTo( )*, и позднее вызывать *moveTo( )* из какой-либо кнопки:

```
function moveTo (x, y) {
  _x = x;
  _y = y;
}
```

Предостережение: поскольку выполнение кода, находящегося в кадрах, зависит от воспроизведения фильма, нужно следить, чтобы переменные, функции и другие элементы программ были доступны к моменту обращения к ним. Нельзя, например, вызвать 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.

Необходимо также гарантировать, что код никогда не будет пытаться обращаться к участкам фильма, которые еще не загружены. Чтобы проверить, загружена ли некоторая часть кода, можно воспользоваться свойством клипа `_framesloaded` или методом *getBytesLoaded( )*. Пример кода можно найти в описании *Movieclip*.`_framesloaded` в части III «Справочник по языку».

# Добавление кода к кнопкам

Код добавляется к кнопке, чтобы связать его выполнение с пользовательским событием. Например, если нужно, чтобы щелчок по кнопке перемещал воспроизводящую головку в кадр с меткой `section1`, в эту кнопку помещается код:

```
on (release) {
  gotoAndStop("section1");
}
```

Для того чтобы добавить код к кнопке, нужно выделить ее на рабочем столе, после чего добавить код в окно сценариев панели Actions. Код в кнопке всегда нужно помещать в обработчик события, описывающий обстоятельства, в которых должен быть выполнен код. Например, событием, запускающим большинство действий кнопок, служит *release*

(освобождение). Можно пользоваться и событием *rollOver*, которое вызывает выполнение кода не тогда, когда кнопка нажата, а когда курсор мыши появляется над кнопкой:

```
on (rollOver) {
  gotoAndStop("section1");
}
```

Полное описание обработчиков событий кнопок см. в главе 10 «События и обработчики событий».

Хотя не запрещено поместить в кнопку хоть тысячу строк кода, перегружать кнопку кодом обычно не рекомендуется. 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) {
  title._xscale = 20;
  title._yscale = 20;
  title._alpha  = 50;
  title._gotoAndPlay("fadeout");
}
```

Но лучше поместить код в функцию и вызывать ее из кнопки:

```
// КОД В КАДРЕ 1 ВРЕМЕННОЙ ДИАГРАММЫ КНОПКИ
function transformClip(clip, scale, transparency, framelabel) {
  clip._xscale = scale;
  clip._yscale = scale;
  clip._alpha = transparency;
  clip.gotoAndPlay(framelabel);
}
// КОД В КНОПКЕ
on (release) {
  transformClip(title, 20, 50, "fadeout");
}
```

При таком подходе весь код хранится централизованно, что облегчает его сопровождение и позволяет быстро и с минимальными усилиями организовывать такое же поведение для других кнопок.

Код кнопки должен быть прикреплен к объекту кнопки на рабочем столе и содержать обработчик события. Код кнопки, не находящийся в обработчике событий, приводит к ошибке; нельзя также прикреплять код к внутренним кадрам `UP`, `OVER`, `DOWN` и `HIT` символа кнопки.

# Добавление сценариев в клипы

Как уже рассказывалось, можно прикреплять код к кадрам временной диаграммы клипа. К самому объекту клипа код тоже можно прикреплять. Для этого нужно выделить экземпляр клипа на рабочем столе, 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 при загрузке кода присваивается значение 10:

```
onClipEvent (load) {
  var x = 10;
}
```

Обработчики событий клипов могут реагировать на действия с мышью и клавиатурой, загрузку данных, вывод кадров, рождение и гибель клипов. Полное описание клипов и обработчиков событий можно найти в главе 10 «События и обработчики событий» и главе 13 «Клипы».

# Куда девался код?

Даже для опытных пользователей Flash размещение кода в фильме иногда может представлять проблему. Поскольку код можно прикрепить к любому кадру любой временной диаграммы, к любой кнопке или любому клипу на рабочем столе, он часто может потеряться в море содержимого. Очевидно, развитая структура и тщательное документирование кода могут сберечь десятки, а то и сотни часов работы на протяжении всего проекта. Но если вы столкнулись с проблемой, когда важный код в фильме таинственным образом исчез, откройте панель Actions и попробуйте разобраться с помощью следующих приемов:

- Щелкните по любому кадру, на временной диаграмме которого расположен маленький круглый ярлычок. Кружок означает наличие кода ActionScript.
- Найдите на рабочем столе и выделите все белые кружки с черной каемкой. Они обозначают пустые клипы, в которых часто не содержится ничего, кроме кода. Если на самом пустом клипе кода не видно, сделайте двойной щелчок по нему, чтобы начать редактирование, и просмотрите его кадры.
- Выделите поочередно каждую кнопку фильма. Некоторые программисты не склонны централизованно размещать код и располагают длинные важные сценарии прямо в кнопках.
- Поищите на временной диаграмме скрытые или замаскированные 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 скрытых слоях могут содержаться закрытые объекты с кодом. Разблокируйте скрытые слои, чтобы увидеть их содержимое.

- Разблокируйте все слои. Пустые клипы (маленькие кружки с черной каемкой) скрываются, если заблокирован слой, на котором они находятся.

Иногда и эти приемы не помогают. Если некто решил спрятать свой код, то Flash предоставляет много мест, в которых можно это сделать. Например, пустой клип можно поместить далеко за пределами рабочего стола, что делает его обнаружение почти невозможным. К счастью, когда кажется, что все потеряно, можно с помощью Movie Explorer найти любой сценарий фильма. Выберите меню Window → Movie Explorer, чтобы с высоты птичьего полета рассмотреть все принадлежности фильма, в том числе сценарии, прикрепленные к кадрам, кнопкам и клипам. Сценарии помечаются ярлыком Actions с голубой стрелкой, совпадающей с ярлыком для элементов в окне инструментов панели Actions. Можно даже задать фильтр, чтобы выводить только сценарии: выбрать ярлык Actions в меню Show в верхней части панели Explorer и снять выделение со всех других ярлыков.

# Производительность

Вот некоторые советы по оптимизации редактирования исходного кода ActionScript:

- Все сценарии временной диаграммы храните в независимом слое под названием *scripts*. Не помещайте в слой *scripts* какое-либо содержимое: используйте его только для кода. Сохраните этот слой поверх структуры слоев, чтобы его код выполнялся после загрузки всех других слоев (или в самом низу, если порядок загрузки определен как «сверху вниз»). Если всегда хранить слой *scripts* в одном и том же месте, легко найти код. (Предыдущий раздел показывает, сколь ценно это может быть!)
- Все метки кадров храните в отдельном слое *labels*. Не помещайте в слой *labels* какое-либо содержимое: используйте его только для меток.
- Взгляните на кнопку со стрелкой в правом верхнем углу панели Actions. Вы найдете там удобные инструменты, такие как поиск и замена, печать исходного текста и управление шрифтами окна сценариев.
- При работе с библиотеками кода, участвующими в нескольких проектах, сохраняйте код во внешних файлах. Подробности смотрите в следующем разделе «Сохранение кода ActionScript во внешних файлах».

- Для экономии ввода используйте клавиши сокращенного набора, имеющиеся в Actions (например, Esc-G-P для *gotoAndPlay( )*). Клавиши сокращенного набора перечислены в кнопке (+) панели Actions.
- Настройте панель Actions, соединив ее с другими панелями для работы с кодом: Instance, Frame и Text Options. Просто откройте любую панель с помощью меню Window → Panels и перетащите на нее панель 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.

# 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 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 редактирования

Во время редактирования файла *.fla* можно импортировать код в окно сценариев при помощи пункта Import From File, который доступен при нажатии на кнопку со стрелкой, находящуюся в правом верхнем углу панели Actions (см. рис. 16.1). Import From File – единовременная 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, нужно выполнить копирование и вставку вручную, используя внешний текстовый редактор.

## #include (импорт на этапе компиляции)

При экспорте (*компиляции*) файла *.swf* из файла *.fla* может быть импортирован код из внешнего текстового файла с помощью директивы `#include`. Об использовании `#include` см. часть III.

## Библиотеки совместного доступа (импорт на этапе выполнения)

Для импорта кода из внешнего источника во время фактического воспроизведения фильма нужно создать файл *.swf* библиотеки совместного доступа, в котором находится клип, содержащий импортируемый код. Импорт на этапе выполнения дает самый гибкий подход к совместному использованию кода в разных фильмах, потому что не требует перекомпиляции фильмов, импортирующих совместный код: когда обновляется библиотечный файл *.swf*, это обновление автоматически отражается на связанных с ним фильмах.

Следующая процедура описывает, как простую проверочную функцию из фильма *codeLibrary.swf* сделать доступной в фильме *myMovie.swf*. Сначала создадим фильм *codeLibrary.swf*:

1. Создайте новый документ Flash.
2. Создайте новый символ клипа с именем `sharedFunctions`.
3. В кадр 1 клипа `sharedFunctions` поместите следующий код:

```
function test () {
  trace("The shared function, test, was called.");
}
```

4. Выберите в библиотеке клип `sharedFunctions`.
5. В панели библиотеки выберите Options → Linkage.
6. Выберите Export This Symbol.
7. В окне Identifier введите `sharedFunctions`.
8. Сохраните документ как *codeLibrary.fla*.
9. При помощи меню File → Export Movie создайте *codeLibrary.swf* из *codeLibrary.fla*.
10. Закройте файлы *codeLibrary.swf* и *codeLibrary.fla*.

Чтобы создать файл *myMovie.swf*, который будет выполнять код, импортированный из *codeLibrary.swf*, выполните следующее:

1. Создайте новый документ Flash.
2. Сохраните новый документ как *myMovie.fla* в одной папке с *codeLibrary.fla*.
3. Переименуйте *Layer 1* в *sharedCode*.
4. При помощи меню File → Open As Shared Library выберите *codeLibrary.fla*. Появится библиотека *codeLibrary. fla*.
5. Перетащите экземпляр клипа `sharedFunctions` из библиотеки *codeLibrary.fla* на рабочий стол кадра 1 *myMovie.fla*.
6. Выделите экземпляр на рабочем столе и выберите Modify → Instance.
7. Дайте экземпляру имя `sharedFunctions`.

8. На главной временной диаграмме *myMovie.fla* создайте новый слой с именем *scripts*.
9. Добавьте ключевой кадр в кадр 2 слоя *scripts*.
10. Добавьте новый кадр в слой *sharedCode*.
11. К кадру 2 слоя *scripts* прикрепите следующий код:

```
stop();
sharedFunctions.test();
```

12. Экспортируйте *myMovie.swf*.
13. В окне Output появляется текст: «The shared function, test, was called».

Обратите внимание, что библиотека совместного доступа является присоединенным ресурсом, вроде графического файла *.gif*, присоединенного к файлу *.html*. Поэтому библиотечные файлы *.swf* должны быть всегда правильно установлены на сервере вместе с файлами, которые их используют. Чтобы изменить значение ссылки с импортированного символа на библиотеку, нужно выполнить следующие шаги:

1. Выбрать символ в библиотеке.
2. В панели библиотеки выбрать меню Options → Linkage.
3. В пункте Import This Symbol from URL указать новый адрес.

# Создание интеллектуальных клипов

Интеллектуальный клип (*Smart Clip*) – это клип, позволяющий присваивать значения некоторым своим переменным с помощью особого графического интерфейса пользователя в редакторе Flash. Интеллектуальные клипы дают возможность пользователям, не являющимся программистами, настраивать программно управляемые клипы. В них переменные, определяющие поведение, отделены от кода клипа, что позволяет рассматривать их как «черные ящики»: их устройство неизвестно, но известны данные на входе и выходе и поведение.

Обычно инициализация переменных происходит в исходном коде клипа. Вот, например, настройка переменных, управляющих фейерверком:

```
// Переменные, определяемые пользователем
var numSparks = 10;           // Число клипов вспышек во взрыве
var randomDispersion = true;  // Тип взрыва (true для случайного,
                              // false для единообразного)
var duration = 1300;          // Длительность взрыва в миллисекундах
```

Модификация такого кода может испугать непрограммиста. Но если построить систему в виде интеллектуального клипа, непрограммист сможет настроить действие фейерверка с помощью знакомого ему интерфейса в стиле приложения. На рис. 16.2 показан интерфейс Smart Clip, эквивалентный коду инициализации переменных.

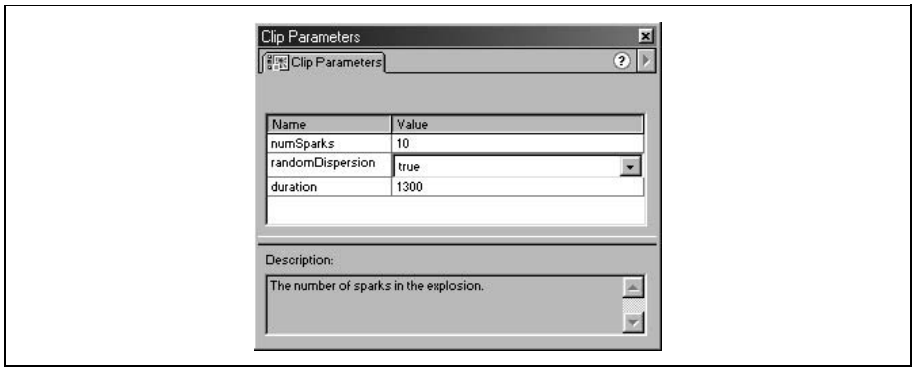


***Рис. 16.2.*** *Пример интерфейса настройки в виде интеллектуального клипа*

В интерфейсе Smart Clip выводятся имена и значения переменных, разделенные по строкам и столбцам. Имена переменных редактировать нельзя, чтобы случайная опечатка не привела к краху системы. Для каждой переменной выводится также словесное описание того, что она делает и как ее задать. 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 близок непрограммистам, чем исходный код. Но интеллектуальные клипы можно делать еще более дружественными пользователю. Интерфейс интеллектуального клипа, устанавливаемый по умолчанию, можно заменять собственным заказным (как на рис. 16.3). Заметьте, что в этом интерфейсе имена переменных вообще скрыты, а непрограммист может настраивать каждый экземпляр фейерверка с помощью текстовых полей и выпадающих меню. Интерфейс позволяет даже выполнить предварительный просмотр настроек в действии!

Рассмотрим, как это работает.

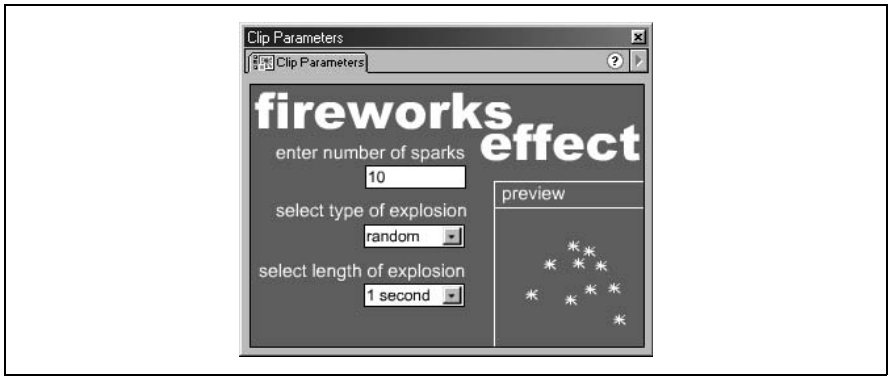


***Рис. 16.3.*** *Специализированный интерфейс настройки в виде интеллектуального клипа*

# Создание интеллектуального клипа со стандартным интерфейсом

Как только что говорилось, интеллектуальные клипы могут иметь системный интерфейс, установленный по умолчанию, или специализированный интерфейс. Сначала научимся строить интерфейс стандартного вида.

При построении любого интеллектуального клипа сначала нужно создать обычный клип, управляемый значениями одной или нескольких переменных. Например, в следующем коде переменные `xPos` и `yPos` определяют положение клипа на рабочем столе:

```
_x = xPos;
_y = yPos;
```

При создании интеллектуального клипа для собственных нужд или для использования другими лицами предполагается, что некоторые определенные переменные будут заданы через интерфейс Smart Clip при помещении клипа на рабочий стол. Эти переменные называют *параметрами клипа*. Когда клип, поведение которого определяется одним или более параметрами клипа, создан, нужно дать ему интерфейс интеллектуального клипа, посредством которого можно задавать эти параметры.

## Добавление стандартного интерфейса в интеллектуальный клип

Чтобы добавить в клип интерфейс интеллектуального клипа по умолчанию, нужно выполнить следующие шаги:

1. Выбрать клип в библиотеке.
2. Выбрать меню Options → Define Clip Parameters. (Появится окно диалога Define Clip Parameters.)
3. В окне Parameters щелкнуть по кнопке плюс (+), чтобы добавить параметр клипа.
4. Повторить шаг 3 для каждого параметра интеллектуального клипа.
5. Настроить параметры клипа, как описано в следующем разделе.

## Настройка стандартных параметров клипа

После добавления параметра в интеллектуальный клип нужно присвоить ему имя и необязательное значение по умолчанию. Как и переменные, параметры клипов могут содержать данные различных типов. Однако параметры клипов поддерживают не совсем такие же типы, которые поддерживаются переменными. Параметры клипов могут содержать *строки*, *числа*, *массивы*, *объекты* и *списки*. От типов данных, поддерживаемых переменными, они отличаются следующим:

- Параметры клипов поддерживают тип значений, называемый списком (*list*) и используемый только в интерфейсах. Список используется для того, чтобы ограничить список значений, которые могут быть присвоены параметру, некоторым заранее определенным множеством значений. Например, для параметра `difficulty` (сложность) может быть установлен список значений: `"высокая"`, `"обычная"`, `"низкая"`. Список предотвращает ввод пользователем недопустимого значения параметра.
- Элементарные типы `Boolean`, `null` и `undefined` нельзя присваивать непосредственно в качестве значений параметра клипа. Это ограничение накладывается интерфейсом интеллектуального клипа, а не самим параметром клипа: код, находящийся внутри клипа, может присваивать значения `Boolean`, `null` или `undefined` переменным, инициализированным как параметры клипа. Для моделирования булевых значений `true` и `false` в параметрах клипа используются числа 1 и 0, а не строки «true» и «false». Числа 1 и 0 преобразуются в булевы значения `true` и `false`, соответственно, при использовании в булевом контексте.

Чтобы дать параметру клипа имя и необязательное значение по умолчанию, нужно выполнить следующие шаги:

1. Дважды щелкнуть по имени параметра и ввести допустимый идентификатор.
2. Дважды щелкнуть по типу параметра и выбрать один из следующих:
   a) Default (значение по умолчанию) для параметров со строковыми или числовыми значениями
   b) Array для параметров, принимающих значение массива
   c) Object для параметров, имеющих значением объект
   d) List для параметров с предустановленным набором возможных строковых или числовых значений
3. Дважды щелкнуть по значению параметра и ввести значение по умолчанию, если оно требуется. Это значение будет выведено в интерфейсе интеллектуального клипа в качестве начального значения параметра. Способ ввода значений по умолчанию зависит от типа параметра:
   a) Для параметров Default дважды щелкнуть по значению параметра и ввести строку или число.
   b) Для параметров Array, Object и List дважды щелкнуть по значению параметра. В диалоговом окне Values можно добавлять, удалять и переупорядочивать с помощью кнопок «плюс», «минус» и стрелок. Щелкнуть OK для принятия настроек.
4. Чтобы добавить сведения о назначении параметров, ввести пояснения в окно Description.

5. Чтобы запретить пользователю интеллектуального клипа изменять имя параметра, выбрать Lock in Instance.
6. В окне диалога Define Clip Parameters щелкнуть OK для завершения настройки параметров.

### Удаление и изменение порядка параметров стандартного клипа

Иногда требуется удалить параметры интеллектуального клипа или изменить их порядок.

Чтобы удалить параметр клипа, нужно выполнить следующие шаги:

1. В библиотеке выбрать клип, который нужно изменить.
2. Выбрать меню Options → Define Clip Parameters.
3. Выбрать удаляемый параметр.
4. Щелкнуть по кнопке «минус» (−).
5. Щелкнуть OK.

Для изменения порядка параметров клипа нужно выполнить следующие шаги:

1. В библиотеке выбрать клип, который нужно изменить.
2. Выбрать Options → Define Clip Parameters.
3. Выбрать перемещаемый параметр.
4. Щелкать по кнопкам со стрелками, пока параметр не окажется в нужном положении.
5. Щелкнуть OK.

## Создание интеллектуального клипа с заказным интерфейсом

Чтобы создать интеллектуальный клип с заказным интерфейсом, нужно сначала создать обычный клип, поведение которого управляется рядом параметров клипа, как это было описано выше. После этого создается независимый файл *.swf* (так называемый *интерфейсный .swf*), который будет использоваться в качестве интерфейса панели параметров клипа. Обычно создается файл *.swf* с графическим интерфейсом, позволяющим пользователю вводить значения параметров (через текстовые поля, меню, кнопки и т. д.). Эти значения автоматически считываются и пересылаются в интеллектуальный клип как параметры.

Интеллектуальный клип обменивается данными с интерфейсом *.swf* через экземпляр `xch` (от *exchange*), специально поименованный экземпляр в интерфейсном *.swf*. (Как создавать экземпляр `xch`, будет по-

казано ниже.) На рис. 16.4 показано, как имена и значения параметров пересылаются из интерфейсного *.swf* в интеллектуальный клип.

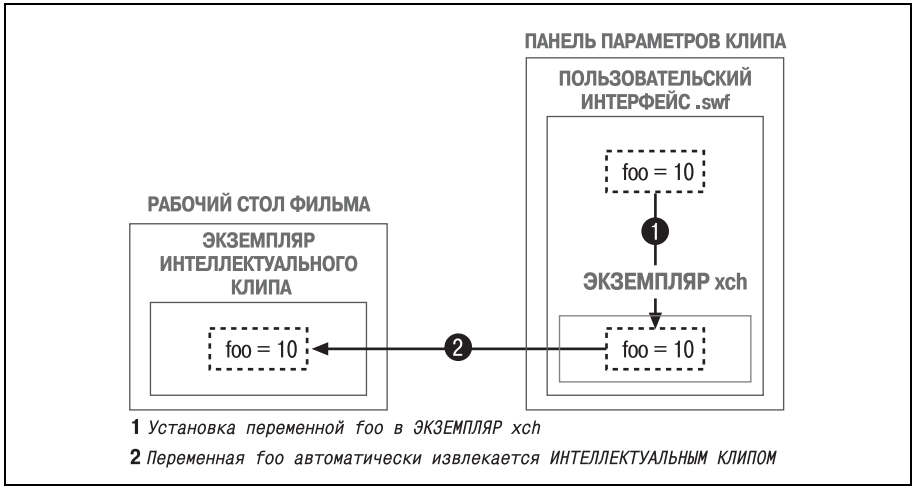


***Рис. 16.4.*** *Обмен данными в заказном интеллектуальном клипе*

Связь между интерфейсным *.swf* и интеллектуальным клипом осуществляется в цикле. Когда экземпляр интеллектуального клипа выбирается на рабочем столе, соответствующий интерфейсный *.swf* загружается в панель Clip Parameters. После этого текущие параметры экземпляра интеллектуального клипа передаются экземпляру `xch` файла *.swf*. Файл *.swf* должен извлечь эти параметры и настроить соответствующим образом состояние интерфейса. Переменные, которые будут потом заданы в `xch` файлом *.swf*, автоматически передаются интеллектуальному клипу в качестве параметров. Когда с экземпляра интеллектуального клипа снимается выделение, интерфейсный *.swf* удаляется из панели Clip Parameters. Однако значения параметров не теряются, они сохраняются интеллектуальным клипом. Если экземпляр интеллектуального клипа выделяется вновь, он передает свои параметры клипу `xch` файла *.swf*. Этот цикл позволяет сохранять синхронизацию интерфейсного файла *.swf* с параметрами интеллектуального клипа.

В следующих разделах описывается, как создать пользовательский интерфейсный *.swf* и связать его с интеллектуальным клипом. Пример интеллектуального клипа с заказным интерфейсом пользователя можно получить под названием «Playhead Control» из сетевого Хранилища кода.

## Создание заказного интерфейсного файла .swf

Чтобы создать файл *.swf*, используемый в качестве заказного интерфейса интеллектуального клипа, нужно выполнить следующие шаги:

1. Открыть новый документ Flash.

2. Создать новый слой с именем *xchLayer*.
3. Выбрать меню 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. Разместить кнопки, текстовые поля и прочие элементы интерфейса, необходимые для задания значений переменных.
8. Переменные, установленные в экземпляре `xch`, будут автоматически добавлены в интеллектуальный клип в качестве параметров. Вот, например, как задать значения двух пользовательских параметров:

```
xch.param1 = value1;  // Значения могут иметь любой тип данных
xch.param2 = value2;
```

9. Чтобы прочесть значение параметра интеллектуального клипа в заказном интерфейсном файле *.swf*, нужно обратиться к этому параметру как свойству `xch`. Например, следующий код инициализирует текстовое поле `param1Input` значением `param1` из интеллектуального клипа:

```
param1Input = xch.param1;
```

10. Экспортировать файл *.swf*.

### Добавление заказного интерфейса в интеллектуальный клип

Теперь, когда ясно, как создавать интерфейсный *.swf*, давайте добавим его к интеллектуальному клипу:

1. Закройте заказной интерфейсный *.swf* и вернитесь к исходному файлу *.fla*, содержащему интеллектуальный клип.
2. Выберите интеллектуальный клип в библиотеке.
3. Выберите меню Options → Define Clip Parameters. Появится диалоговое окно Define Clip Parameters.
4. В окне Link to Custom UI введите местоположение файла *.swf*, который будет использоваться в качестве заказного интерфейса, относительно текущего файла *.fla*. (Выбрать файл *.swf* можно и с помощью кнопки папки.)

## Использование интеллектуальных клипов

Когда клипу присвоены параметры, он официально становится «интеллектуальным клипом». Интеллектуальные клипы помечаются в библиотеке специальным значком: .

Чтобы использовать экземпляр интеллектуального клипа в фильме, нужно выполнить следующие шаги:

1. Перетащить интеллектуальный клип из библиотеки на рабочий стол.
2. Выбрать меню Window → Panels → Clip Parameters.
3. Если у клипа стандартный интерфейс, задать значение каждого параметра следующим образом:
   a) Для параметров Default дважды щелкнуть по значению параметра и ввести строку или число.
   b) Для параметров Array дважды щелкнуть по значению параметра. В диалоговом окне Values дважды щелкнуть по каждому значению элемента массива и ввести строку или число. Щелкнуть OK, чтобы принять значения элементов массива.
   c) Для параметров Object дважды щелкнуть по значению параметра. В диалоговом окне Values дважды щелкнуть по каждому значению свойства объекта и ввести строку или число. Щелкнуть OK, чтобы принять значения свойств вашего объекта.
   d) Для параметров List дважды щелкнуть по значению параметра, после чего выбрать вариант значения.
4. Если у клипа заказной интерфейс, используйте для настройки параметров клипа те средства, которые предоставляет заказной интерфейс.

# Вперед!

В следующей главе мы соединим ActionScript с простым приложением сервера для получения формы Flash.

# 17

# Формы Flash

Интерактивность в Сети достигает, видимо, своего высшего выражения в облике заполняемых форм. Это может показаться преувеличением, но на самом деле им не является. Формы могут, на первый взгляд, казаться тривиальными, но они являются сердцевиной электронных сообществ (чатов и досок объявлений), сохранения данных (персонализации) и, конечно, электронной коммерции (торговли через Интернет).

Первой версией Flash с поддержкой форм был Flash 4. С тех пор Flash приобрел много мощных средств для связи с сервером. В данной главе исследуются основы использования форм во Flash, от перехвата данных, вводимых пользователем, и вывода результатов во Flash, до отправки данных серверу и получения с сервера ответа. Более сложные темы, например, работа с XML, освещены в части III «Справочник по языку».

## Цикл данных форм Flash

Прежде чем погружаться в частности, бросим общий взгляд на типичные этапы процесса передачи формы:

1. Flash получает данные, введенные пользователем.
2. Flash готовит данные для отправки веб-серверу (собирает и проверяет значения переменных).
3. Flash отправляет данные веб-серверу по HTTP (или HTTPS).
4. Веб-сервер получает данные и передает их приложению, обрабатывающему данные на стороне сервера (например, сценарию Perl, сценарию PHP, Cold Fusion или ASP).

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, а, возможно, и 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

Чтобы осуществить передачу своих данных, пользователи обычно вводят текст в текстовые поля и щелкают по кнопке формы Submit. В качестве готовых компонент формы даются только текстовые поля, все остальное, включая кнопку Submit, нужно делать вручную (как будет показано ниже). Сведения о текстовых полях можно найти в разделе «Текстовые поля для ввода данных пользователем» в главе 18 «Экранные текстовые поля».

Так как Flash обладает полным языком для написания сценариев, можно создавать интеллектуальные формы, способные производить предварительную обработку данных до отправки их на сервер. Данные, введенные пользователем, проверяются перед их отправкой веб-серверу, чтобы приложение, обрабатывающее данные, получало пригодные для работы данные. Проверка данных обычно выясняет, 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 для обработки на стороне сервера

Если правильность данных подтверждена, можно безбоязненно передать их веб-серверу. ActionScript предоставляет несколько средств для пересылки данных формы на веб-сервер:

- Функция *loadVariables( )*, описываемая ниже в этой главе
- Методы *send( )* и *sendAndLoad( )* класса *XML*, описываемые в части III
- Глобальная функция *getURL( )*, описываемая в части III

Веб-сервер пересылает данные Flash выполняемому на стороне сервера приложению, которое должно обработать данные и обычно является

базой данных промежуточного уровня (например, Allaire ColdFusion или Microsoft ASP) или сценарием CGI (например, сценарием Perl или PHP, или серверным приложением (сервлетом) Java).

Описывая цикл данных клиент/сервер сети, мы обращаем особое внимание на различие между веб-сервером и приложением, обрабатывающим данные. Часто это различие скрыто: клиент всегда должен сделать запрос HTTP, чтобы отправить данные приложению, которое должно их обработать, поэтому естественно, что в процессе участвует веб-сервер. Однако при разработке форм Flash нужно всегда помнить о невидимых передачах между веб-сервером и приложением, обрабатывающим данные. Данные перемещаются из Flash на сервер либо в конце URL (если используется метод GET), либо в потоке имен переменных и их значений (если используется метод POST). Если возникает ошибка на веб-сервере, Flash не выводит сообщения 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, мы не увидим сообщений об ошибках выполнения. Чтобы выяснить, какие проблемы клиента/сервера возникают при работе с Flash, полезно посмотреть, что записал веб-сервер в журнал ошибок HTTP при попытках выполнения сценариев. Может оказаться, что веб-сервер пытается сообщить о чем-то, чего Flash не может передать.

## Приложение, обрабатывающее данные

Получив порцию данных, приложение, обрабатывающее данные, должно произвести их анализ (т. е. интерпретировать разумным образом и при необходимости разбить на более мелкие части). После анализа данных приложение, работающее на сервере, может распорядиться ими массой способов. Обычно в обработку данных входит сохранение содержимого в базе данных или плоском текстовом файле для возможного извлечения в будущем.

Если обработка данных завершена, то приложение, обрабатывающее данные, выводит результат, который должен быть передан обратно Flash. Этот результат может быть любым – от простого подтверждающего сообщения («Спасибо за переданные вами сведения») до списка записей из базы данных или цены 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 задавать следующий MIME-тип результата: «application/x-www-urlform-encoded». Если такого типа MIME нет, может оказаться, что Flash не работает с полученным им результатом.

## Flash получает и интерпретирует результаты

Мы приблизились к концу цикла форм Flash, но еще не достигли его. Во-первых, нужно гарантировать, чтобы Flash спокойно ждал, пока приложение на стороне сервера обработает данные и перешлет результат. 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 и возвратить ее. Пока фильм находится в ожидании, выводится сообщение «Loading» (идет загрузка). Когда цена получена, фильм снова активизируется.

Данные, полученные Flash в ответ на вызов *loadVariables( )*, хранятся в заданном клипе или уровне. Когда данные получены, цикл формы Flash завершается, и мы вправе как угодно обрабатывать проделавшие долгий путь драгоценные байты содержимого. Теперь применим полученные знания на практике, создав простую заполняемую форму.

# Создание заполняемой формы Flash

В нашем примере есть все необходимые компоненты формы Flash, перечисленные выше, но в максимально упрощенном виде. Будет продемонстрирована отправка единственной переменной текстового поля из Flash в сценарий Perl с именем *echo.pl* и получение Flash ответа от *echo.pl*. Действующие версии файлов примера можно загрузить из сетевого Хранилища кода.

## Создание интерфейса клиента

В отличие от HTML, во Flash не интегрирован механизм создания форм. В HTML выпадающие меню и переключатели создаются просто – с помощью тегов `<SELECT>`, `<OPTION>` и `<INPUT TYPE="RADIO">`. Во Flash эти элементы нужно делать самому. Единственным встроенным элементом форм во Flash является текстовое поле для данных, вводимых пользователем (эквивалент имеющихся в HTML `<INPUT TYPE="TEXT">` или `<INPUT TYPE="TEXTAREA">`).

Хотя графические элементы форм не встроены непосредственно в редактор Flash, но переключатели, флажки и выпадающие меню присутствуют в виде поставляемых вместе с продуктом интеллектуальных клипов. Для доступа к ним нужно выбрать пункт меню Window → Common Libraries → Smart Clips.

В нашей форме будут текстовое поле для ввода данных пользователем и кнопка Submit. Поместим эти два элемента в клип, чтобы можно было легко определить переменные, посылаемые серверу. Во-первых, со-

здадим новый документ и клип с именем `formClip`, для чего нужно проделать следующее:

1. Создать новый документ Flash.
2. Выбрать меню Insert → New Symbol. Появляется окно диалога Symbol Properties.
3. В поле Name ввести `formClip`.
4. Щелкнуть OK.
5. Перетащить экземпляр `formClip` из библиотеки на главный рабочий стол.

Теперь нужно добавить в `formClip` текстовое поле для ввода пользователем данных:

1. В библиотеке выполнить двойной щелчок по символу `formClip`, чтобы начать редактирование клипа.
2. Выбрать инструмент Text.
3. На холсте клипа растянуть прямоугольник, достаточный по размеру для ввода пользователем одной строки текста.
4. Выбрать меню Window → Panels → Text Options.
5. В Text Type выбрать Input Text.
6. В Line Display выбрать Single Line.
7. В поле Variable ввести `input`.
8. Выбрать Border/Bg.

Теперь назначим текстовому полю `input` значение по умолчанию, которое будет посылаться серверу, если пользователь не введет никаких данных:

1. Выберите кадр 1 на временной диаграмме formClip.
2. В панели Actions введите код: `input = "";`.

Наконец, чтобы добавить кнопку Submit в клип `formClip`, нужно выполнить следующие шаги:

1. Выбрать меню Window → Common Libraries → Buttons.
2. Перетащить из библиотеки *Buttons.fla* экземпляр кнопки *Push Bar* на рабочий стол `formClip`.

## Отправка данных серверу

Существует много способов отправить данные из Flash приложению на сервере, в том числе с помощью методов *loadVariables( )*, *getURL( )*, *loadMovie( )*, *XML.load( )*, *XML.sendAndLoad( )* и *XMLSocket.send( )*. В данном примере будет использован метод *loadVariables( )*. Сведения о других методах можно найти в части III.

В предыдущем разделе мы расположили в клипе `formClip` поле для текстового ввода и кнопку Submit. Чтобы заставить кнопку Submit от-

правлять по щелчку переменные клипа formClips сценарию *echo. pl*, нужно выполнить следующие действия:

1. Выполнить двойной щелчок по символу formClip в библиотеке (в результате можно редактировать клип).
2. Выделить экземпляр кнопки *Push Bar*.
3. Выбрать меню Window → Actions.
4. Ввести в панель Actions следующий код:

```
on (release, keyPress "<Enter>") {
  loadVariables ("http://www.yourserver.com/cgi-bin/echo.pl",
                 "_root.response",
                 "GET");

  _root.response.gotoAndStop("loading");
}
```

Код в кнопке Submit вызывает *loadVariables( )* для отправки переменных клипа formClip сценарию *echo.pl* и заставляет клип response вывести сообщение о продолжающейся загрузке. Клип response мы создадим чуть позже. А сейчас рассмотрим, как работает вызов метода *loadVariables( )*.

В первом параметре вызова *loadVariables( )* должно быть указано местонахождение *echo. pl* на сервере (сценарии, выполняемые на сервере, обычно хранятся в папке с именем *cgi-bin*). Правильно укажите адрес соответственно доменному имени и структуре каталогов вашего сервера. При выполнении *loadVariables( )* по этому адресу будут отправлены все переменные клипа formClip.

Второй параметр в вызове *loadVariables( )* указывает путь к клипу, в котором нужно сохранить значение, возвращаемое *echo.pl*, а именно: "_root.response".

Третий параметр вызова *loadVariables( )* указывает метод HTTP, применяемый для передачи серверу переменных formClip; в данном случае это метод GET. ActionScript поддерживает обе операции: GET и POST, как сказано в описании *loadVariables( )* в части III.

## Сценарий Perl echo.pl

При щелчке пользователя по кнопке Submit на клипе formClip нашего фильма Flash начинает выполнение запроса HTTP GET. Этот запрос выполняет сценарий Perl *echo.pl*. Чтобы наша форма работала, *echo.pl* должен быть помещен в каталог веб-сервера, подключенный к CGI, и следующим образом настроен администратором сервера:

- Сценарий должен быть исполняемым модулем (обычно для этого нужно установить права доступа к файлу как 755).
- В сценарии должен быть указан путь к интерпретатору Perl, если он выполняется в Unix.

В примере 17.1 показан исходный код *echo.pl*. Учтите, что в Perl символ # обозначает комментарий.

*Пример 17.1. Исходный код echo.pl*

```
#! /usr/local/bin/perl
#-------------------------------------------------------------------------------
#  Название: Simple Flash Echo
#  Версия:   1.2.0
#  Автор:    Derek Clayton  derek_clayton@iceinc.com
#  Описание: Возвращает пары имя/значение, полученные от Flash GET или POST.
#-------------------------------------------------------------------------------
# MAIN
#-------------------------------------------------------------------------------
use CGI;                    # Использовать при анализе CGI.pm
$query = new CGI;           # Объект запроса
$echoString = "output=";    # Инициализировать выходную строку
&getInput;                  # Получить данные от Flash
&writeResponse;             # Вывести Flash ответ
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:$value\n";
  }
  # Удалить символ перевода строки (\n) в конце строки перед выводом
  chomp($echoString);
}

sub writeResponse {
  # Установить тип содержимого для Flash
  print "Content-type: application/x-www-urlform-encoded\n\n";
  # Вывести данные
  print $echoString;
}
```

Сценарий *echo.pl* выполняет три главные задачи:

- Принимает данные, отправленные Flash, и разбирает их на имена переменных и их значения.
- Собирает из этих имен переменных и их значений строку, которая должна быть возвращена Flash. Формат строки следующий:

  `output=`*`name1`*`:`*`value1`*`\n`*`name2`*`:`*`value2`*`\n...`*`namen`*`:`*`valuen`*

- Возвращает строку Flash.

При получении строки, возвращенной *echo.pl*, Flash автоматически интерпретирует ее как последовательность переменных в кодировке URL (как описывается для *loadVariables( )* в части III). Поэтому `output` становится переменной на временной диаграмме клипа `response`. Рас-

смотрев значение output в фильме Flash, можно узнать, какие переменные и с какими значениями были первоначально отправлены в *echo.pl*.

Ясно, что *echo.pl* – не самое интересное на свете сетевое приложение. Оно служит только для демонстрации приема. Однако применение этого приема может привести к интересным и эффективным результатам. Образец более тщательно проработанной системы на Perl можно найти в примере с базой данных в плоском файле, который есть в сетевом Хранилище кода в разделе «Server Communication».

## Получение результатов с сервера

Вспомните, что при отправке переменных клипа formClip сценарию *echo.pl* возвращаемое этим сценарием значение должно было быть сохранено в клипе response:

```
loadVariables ("http://www.yourserver.com/cgi-bin/echo.pl",
               "_root.response",
               "GET");
```

Сейчас мы создадим клип response, у которого будет три состояния: *idle* (бездействует), *loading* (идет загрузка) и *done loading* (загрузка завершена). В состоянии *idle* клип response не виден пользователю и ждет начала загрузки данных. В состоянии *loading* клип response показывает пользователю, что данные переданы серверу, и Flash ждет ответа. В состоянии *done loading* клип response уже получил ответ сервера и показывает его пользователю в текстовом поле. Три состояния response определяют структуру его временной диаграммы. Каждое состояние представлено ключевым кадром с меткой: idle, loading и doneLoading. Какой кадр должен быть выведен, определяется следующим образом:

- При загрузке фильма response выводит кадр idle.
- Когда переменные переданы, кнопка Submit переводит response в кадр loading.
- Когда переменные получены, обработчик события *data* клипа response (мы еще создадим его) посылает response в кадр doneLoading.

Для создания клипа response нужно выполнить следующие действия:

1. Выбрать меню Insert → New Symbol. Появляется окно диалога Symbol Properties.
2. В окне Name ввести responseClip.
3. Щелкнуть OK.
4. Перетащить из библиотеки экземпляр responseClip на главный рабочий стол.
5. Дать экземпляру responseClip имя response.
6. Выполнить двойной щелчок по символу responseClip в библиотеке (это приводит к редактированию клипа).

7. Создать четыре слоя временной диаграммы с именами сверху вниз: *scripts*, *labels*, *loading* и *outputField*.
8. Создать три ключевых кадра в каждом слое.
9. В слое *labels* установить для кадров 1, 2 и 3 метки `idle`, `loading` и `doneLoading`, соответственно.
10. В кадр 1 слоя *scripts* поместить код: `stop();`.
11. В кадр 2 слоя *loading* поместить статический текст `"loading, please wait"`.
12. Выбрать инструмент Text.
13. В кадре 3 слоя *outputField* нарисовать текстовое поле.
14. Выбрать меню Window → Panels → Text Options.
15. Для Text Type выбрать Dynamic Text.
16. Для Line Display выбрать Multiline.
17. В поле Variable ввести `outputField`.

Теперь создадим обработчик события *data*, который будет запускаться по окончании пересылки сервером данных во Flash. Для этого нужно выполнить следующие действия:

1. На главном рабочем столе выделить экземпляр `response`.
2. Выбрать меню Window → Actions.
3. Ввести в панель Actions код:

```
onClipEvent (data) {
  this.gotoAndStop("doneLoading");
  outputField = output;
}
```

Когда Flash получает содержимое, посланное сценарием *echo.pl*, автоматически выполняется обработчик события *data* клипа `response`. В обработчике *data* воспроизводящая головка клипа `response` перемещается в кадр `doneLoading`, а затем в поле `outputField` выводится значение `output` (полученное от *echo.pl*). Использование обработчика события *data* гарантирует, что переменная `output` будет загружена ранее попытки вывести ее в текстовом поле `outputField`.

Осталось только проверить, как работает форма! Экспортируйте фильм, введите какой-нибудь текст в поле `input` и щелкните по кнопке Submit. Если форма сразу не сработает, проверьте правильность конфигурации сценария на сервере. Если вы помните, можно получить действующую версию в сетевом Хранилище кода.

Упражнение для читателей: попробуйте поместить на форму кнопку Reset, которая очищает значение поля ввода. Создайте на `formClip` новую кнопку и прикрепите к ней код:

```
on (release) {
  input = "";
}
```

Попробуйте также поместить на форму несколько полей ввода – сценарий Perl услужливо вернет все переменные, которые вы ему отправите. Сможете ли вы разделить переменные и вывести значение каждой из них отдельно?

# Вперед!

Должен сказать, мы неплохо поработали в этой главе. Подумайте, какие мощные вещи 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 плоском файле из сетевого Хранилища кода позволит вам еще дальше продвинуться по пути создания такого рода приложений. В следующей главе мы изучим сложности текстовых полей и их применения в формах Flash.

# 18

# Экранные текстовые поля

Поскольку Flash в основе своей является визуальной средой, его фильмы часто выводят информацию для пользователей на экран. Аналогично, поскольку Flash является интерактивной средой, фильмы часто получают информацию от пользователей через GUI. Для вывода значения переменной на экран или ввода пользователем с клавиатуры данных в фильм используются *текстовые поля*.

Текстовые поля предоставляют средство задания и получения значений переменных, имеющих видимое представление. Есть два вида текстовых полей: динамические текстовые поля, используемые для вывода информации пользователю, и текстовые поля для ввода, с помощью которых получают данные от пользователя.

## Динамические текстовые поля

Динамическое текстовое поле похоже на окно просмотра переменной: оно выводит значение заданной переменной в виде текстовой строки. Динамические текстовые поля создаются во Flash с помощью инструмента Text. Однако, в отличие от обычного статического текста, содержимое динамического текстового поля связано с переменной и может изменяться или извлекаться с помощью ActionScript.

Извлекая значение текстового поля, можно получать информацию с экрана и использовать ее в сценарии. Задавая значение текстового поля, можно выводить некоторое значение на экран.

## Создание динамического текстового поля

Чтобы создать новое динамическое текстовое поле, нужно выполнить следующие действия:

1. Выбрать инструмент Text.
2. Щелчком и перемещением курсора мыши нарисовать прямоугольник на рабочем столе. Созданные очертания определяют размер нового текстового поля.
3. Выбрать меню Text → Options. Появляется панель Text Options.
4. В меню Text Type выбрать пункт Dynamic Text (другие возможные варианты – Static Text и Input Text).
5. В поле Variable ввести имя динамического текстового поля согласно правилам составления допустимых имен переменных, изложенным в главе 2 «Переменные».

После создания динамического текстового поля обычно задаются параметры нового поля, как описано ниже.

## Изменение содержимого динамического текстового поля

Если текстовое поле создано, можно с его помощью вывести на экран значение. Например, если созданное динамическое текстовое поле названо `myText`, задать его содержимое можно с помощью следующих команд:

```
myText = 10;                    // Вывести число в текстовом поле myText
myText = "Welcome to my web site"; // Вывести строку вместо числа

var msg = "Please make a selection";
myText = msg;                        // Вывести значение msg в myText
```

Когда изменяется значение переменной `myText`, содержимое динамического текстового поля обновляется, отражая изменение `myText`. Однако, прежде чем значение посылается для отображения в динамическое текстовое поле, оно преобразуется в строку. Поэтому фактическое содержимое определяется правилами преобразования строк, описанными в табл. 3.2.

Подобно обычным переменным текстовые поля привязываются к временной диаграмме клипа, в котором находятся. Для доступа к динамическому текстовому полю во временной диаграмме удаленного клипа используются приемы, описанные в разделе «Доступ к переменным других временных диаграмм» главы 2.

## Извлечение значения динамического текстового поля

Извлечь значение динамического текстового поля можно, просто воспользовавшись его именем. Например, если myTextField является динамическим текстовым полем в фильме, можно получить его значение и присвоить другой переменной так:

```
welcomeMessage = myTextField;
```

Присвоение значения текстовому полю и его извлечение часто соединяются в одном предложении. Для дописывания текста к текущему содержимому текстового поля можно использовать оператор +=:

```
// Задать значение текстового поля
myTextField = "Today's Headlines...";
// Создать новое сообщение
var newText = "Update! The Party Has Been Cancelled!"
// Добавить новое сообщение к имеющемуся содержимому текстового поля
myTextField += newText;
```

# Текстовые поля для ввода данных пользователем

Текстовые поля для ввода данных пользователем отличаются от динамических текстовых полей только тем, что пользователь может редактировать их во время воспроизведения фильма. Это означает, что пользователь, вводя данные с клавиатуры, может изменять значение такого поля. ActionScript может извлекать введенные пользователем данные и обрабатывать их. Текстовые поля для ввода данных пользователем полезно применять в гостевых книгах, бланках заказов, полях для ввода паролей и всюду, где требуются сведения от пользователя.

## Создание текстового поля для ввода данных пользователем

Чтобы создать текстовое поле для ввода данных пользователем, нужно выполнить те же действия, которые описаны выше в разделе «Создание динамического текстового поля», но в меню Text Type выбрать Input Text вместо Dynamic Text.

## Изменение содержимого текстового поля для ввода

Так же как и для динамических текстовых полей, содержимое текстового поля для ввода данных пользователем можно в любой момент изменить, задав его новое значение с помощью оператора присваивания:

```
myInputText = "Type your name here";
```

Поскольку текстовые поля для ввода данных пользователем обычно используются для получения, а не отображения данных, их значение чаще всего задается, чтобы обеспечить значение по умолчанию для данных, вводимых пользователем.

## Извлечение значения текстового поля для ввода и его использование

Значение текстового поля можно извлечь, просто сославшись на него в сценарии по имени. Например, чтобы вывести значение текстового поля для ввода с именем `myInput`, можно воспользоваться следующим предложением:

```
trace(myInput);
```

Поскольку данные, введенные пользователем в текстовое поле, всегда имеют строковый тип, перед использованием их в ином, не строковом контексте следует выполнить явное преобразование типа данных, как в следующем простом примере калькулятора, который складывает значения двух текстовых полей ввода:

```
// Пусть пользователь ввел в myFirstInput 5 и в mySecondInput 10,
// тогда складываем поля
// ОШИБКА: "Сложение" полей присвоит myOutput значение "Total: 510",
// потому что оператор + интерпретируется как конкатенация строк
myOutput = "Total: " + (myFirstInput + mySecondInput);
// ПРАВИЛЬНО: Сначала преобразовать поля в числа
myOutput = "Total: " + (parseFloat(myFirstInput) + parseFloat(mySecondInput));
```

## Текстовые поля для ввода и формы

Текстовые поля для ввода данных пользователем часто применяются в заполняемых формах, пересылаемых приложениям, выполняемым на стороне сервера, например сценариям Perl. Когда переменные передаются серверу с помощью *loadVariables( )*, пересылаются только те переменные, которые определены в текущем клипе. Поэтому когда форма содержит текстовые поля для ввода, их следует хранить в одном отдельном клипе для облегчения передачи серверу группой. Дополнительные сведения о *loadVariables( )* см. в главе 17 «Формы Flash» и части III «Справочник по языку».

# Параметры текстовых полей

У динамических текстовых полей и текстовых полей для ввода данных пользователем многие параметры, используемые для настройки характеристик вывода и ввода, являются общими, но не все. На рис. 18.1

показана панель настройки Text Options для текстовых полей ввода и динамических текстовых полей.

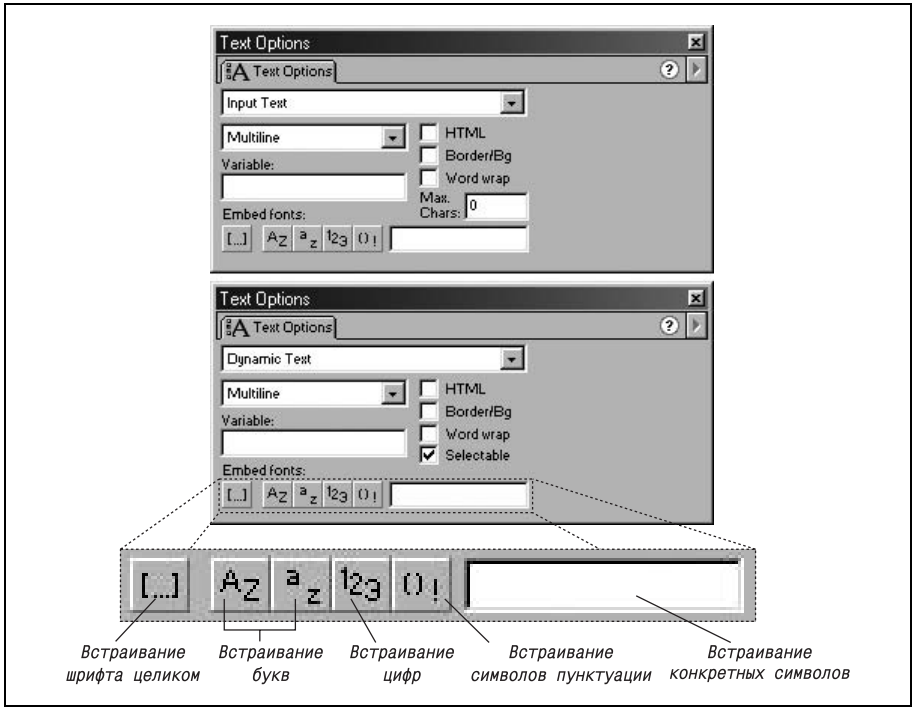


***Рис. 18.1.*** *Панель Text Options*

## Настройка стиля текстового поля

Для задания стиля ввода текстового поля или маскировки вводимых пользователем данных используется меню Line Display, в котором можно выбрать один из трех параметров:

*Single Line*

Вариант Single Line не позволяет пользователю ввести в поле более одной строки текста, отключая клавишу Enter на время ввода текста.

Установка Single Line оказывает также воздействие на текст, вводимый в редакторе Flash без символов переноса строк; текст, который автоматически «мягко переносится» во время редактирования, не будет переноситься в проигрывателе. Вместо этого текст будет выводиться в одну строку, даже если он целиком не умещается в поле по ширине. Однако на символы возврата каретки, введенные во время редактирования, установка Single Line не оказывает воздействия; текст с жесткими переносами строки выводится в проигрывателе так, как он выглядел в редакторе.

Параметр настройки Single Line в основном относится к текстовым полям для ввода данных пользователем. При использовании с динамическими текстовыми полями он ведет себя так же, как динамическое текстовое поле Multiline, если только не включен также и параметр Word Wrap.

Escape-последовательность `\n` и ключевое слово `newline` вставляют символ переноса строки, не обращая внимание на установку Single Line. Например, если присвоить переменной текстового поля значение `"this is\na test"`, то тексты `"this is"` и `"a test"` будут выведены на разных строках.

*Multiline*

Установка Multiline разрешает пользователю ввести в текстовое поле более одной строки текста. Если выбран режим Multiline, то пользователь может вводить символ возврата каретки.

Multiline не оказывает воздействия на вывод динамического текстового поля, если только не используется вместе с опцией Word Wrap. Если опция Word Wrap не включена, текстовые поля Multiline ведут себя так же, как текстовые поля Single Line.

*Password*

Параметр Password позволяет скрыть символы, вводимые в форму, и применим только к текстовым полям для ввода данных пользователем. Такое поле ведет себя, как поле Single Line, но все символы, включая пробелы, маскируются звездочками (*). Например, слова `"hi there"` будут выведены как `"********"`.

Благодаря некоторой странности интерфейса Flash оказывается возможным переносить текст в поле Password по словам. Если установить Line Display в Multiline и выбрать Word Wrap, а затем установить Line Display в Password, то установка Word Wrap будет сохранена. Однако использовать многострочные пароли не рекомендуется, т. к. большинство пользователей будет этим озадачено.

## Указание имени текстового поля

Параметр Variable панели Text Options используется для указания имени текстового поля – динамического или для ввода данных. Текстовым полям необходимо давать имена, чтобы иметь возможность работать с ними через ActionScript. Давая имена текстовым полям, соблюдайте правила построения допустимых имен переменных, изложенные в главе 2 «Переменные» и главе 14 «Лексическая структура».

## Задание рамки и фона

При задании в панели Text Options параметра Border/Bg текстовое поля обводится черной рамкой, а для просматриваемой области поля устанавливается белый фон. Выбрать другие цвета и стили нельзя. Что-

бы установить для текстового поля другой фон, нужно отключить параметр Border/Bg и вручную нарисовать форму под текстовым полем.

## Перенос по словам

При использовании вместе с установкой Multiline меню Line Display опция Word Wrap вызывает мягкий перенос строк текста, превосходящих ширину поля. Эта настройка распространяется как на текст, введенный пользователем, так и на текст, выводимый через ActionScript.

Если установить опцию Word Wrap при выбранной опции Multiline, а затем выбрать Single Line, то установка Word Wrap сохранит свое действие. Если не нужно, чтобы текст переносился в конце каждой строки, следует убрать Word Wrap.

## Выделение текста

Пользователь может выделить текст в динамическом текстовом поле, только если для этого поля установлена опция Selectable. Но даже в этом случае динамический текст можно только копировать, а удалять или редактировать его нельзя. В полях для ввода данных пользователем всегда можно производить выделение, копирование, удаление и редактирование текста.

Копирование, удаление и вставку текста во Flash под Windows необходимо выполнять при помощи контекстного меню, вызываемого щелчком правой кнопки мыши (или Ctrl+щелчок на Macintosh). Клавиши сокращенного набора, например Ctrl-C и Ctrl-V (в Windows) или Cmd-C и Cmd-V (на Macintosh), игнорируются.

## Максимальное количество символов

Параметр Max Characters используется только с текстовыми полями для ввода данных пользователем и ограничивает возможный объем вводимого текста. По умолчанию Max Characters устанавливается в 0, что разрешает вводить текст неограниченного объема. Другие установки задают максимальное количество вводимых символов.

Max Characters часто используется с формами, требующими определенного формата данных. Например, с помощью этого параметра можно ограничить поля для ввода даты двумя цифрами для числа, двумя цифрами для месяца и четырьмя цифрами для года.

## Встраивание шрифтов

По умолчанию во всех текстовых полях (динамических и полях ввода) применяются шрифты устройств (*device fonts*), т. е. шрифты, установ-

ленные на машине пользователя. При использовании шрифтов устройств текст выглядит на машине пользователя так, как он выглядел при создании документа (но без сглаживания), если у пользователя есть шрифт, указанный в панели Character для текстового окна. Если такого шрифта у пользователя нет, используется какой-то альтернативный шрифт, что не всегда желательно.

Чтобы гарантировать вывод текста определенным шрифтом, его встраивают в фильм с помощью параметров Embed Fonts, показанных в увеличенном виде на рис. 18.1.

Существуют следующие возможности:

- Встроить весь шрифт целиком с помощью кнопки [ . . . ].
- Встроить любую комбинацию букв, цифр и символов пунктуации с помощью кнопок AZ, az, 123 и ()!.
- Встроить конкретные символы, введя их в имеющееся поле.

Встраивание всего шрифта типа Roman обычно увеличивает размер фильма на 20–30 Кбайт (для азиатских шрифтов увеличение может оказаться гораздо более значительным). Используя лишь некоторое подмножество символов (встраивая только необходимые символы), размер файла можно значительно сократить. Те символы, которые не встраиваются, не могут быть введены пользователем или отображены с помощью ActionScript. Этим можно воспользоваться, чтобы ограничить ввод текста определенными символами.

Параметр Embed Fonts должен задаваться отдельно для *каждого* текстового поля, использующего какой-то особый шрифт, даже если несколько полей используют один и тот же шрифт. Однако если для нескольких полей встраивается один и тот же шрифт, это не влияет на размер файла, т. к. только один экземпляр шрифта загружается вместе с фильмом. Чтобы применить один и тот же параметр Embed Fonts одновременно к нескольким текстовым полям, выделите нужные поля и задайте параметр Embed Fonts обычным образом.

При использовании встроенных шрифтов текст всегда выводится в текстовых полях без сглаживания. Поэтому не рекомендуется применять встроенные шрифты размером менее 10 пунктов, т. к. при таком размере большинство шрифтов без сглаживания становятся нечитаемыми. Чтобы сохранить сглаживание шрифтов, отключите все параметры Embed Fonts и используйте шрифты устройств (т. е. системные). Для шрифтов устройств сглаживание всегда выполняется.

Содержимое текстового поля, которое повернуто или замаскировано, появляется на экране, только если его шрифт является встроенным. Это означает, что нельзя поворачивать или маскировать текстовые поля, использующие шрифты устройств.

О других важных особенностях использования шрифтов в текстовых полях читайте в разделе «Вывод в формате HTML» ниже в этой главе.

# Свойства текстовых полей

Если текст, подлежащий выводу в текстовом поле, занимает больше строк, чем помещается в видимой области текстового поля, лишние строки текста оказываются скрытыми. Однако эти лишние строки остаются частью текстового поля. Чтобы увидеть их, нужно щелкнуть по полю и нажимать клавишу «стрелка вниз», пока не появятся эти избыточные строки. Очевидно, нельзя полагаться на то, что пользователи будут прокручивать текст в текстовом поле с помощью клавиш со стрелками. Вместо этого следует разместить кнопки для прокрутки текста с помощью свойств `scroll` и `maxscroll`, использующих числовые индексы для ссылки на строки текстового поля. Номер верхней строки равен 1; номер возрастает для каждой следующей строки текстового поля, в том числе и для не поместившихся в видимую область поля. На рис. 18.2 приведен пример значений индексов строк текстового поля.

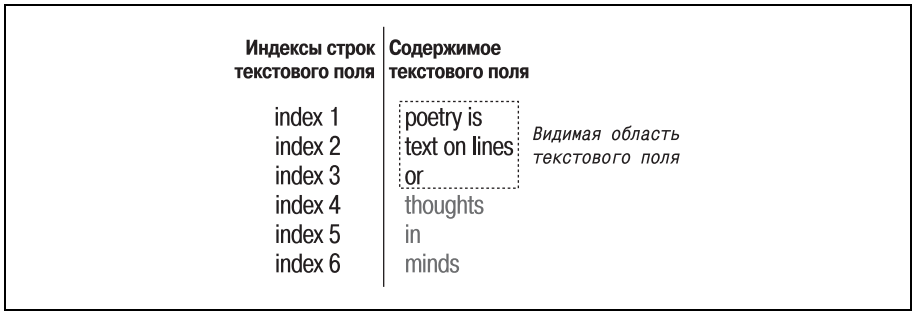


***Рис. 18.2.** Индексы строк текстового поля*

# Свойство scroll

Свойство `scroll` представляет номер самой верхней строки, выведенной в данный момент в текстовом поле. Обращение к нему выглядит так: *`textFieldName`*`.scroll`.

Если текстовое поле содержит больше строк, чем может одновременно быть выведено, можно управлять тем, какие строки отображаются в видимой области поля, путем задания свойства `scroll`. Например, если задать свойство `scroll` текстового поля, показанного на рис. 18.2, равным 3, то в текстовом поле будет выведено:

```
or
thoughts
in
```

## Свойство maxscroll

Свойство `maxscroll` сообщает о том, насколько далеко можно прокрутить текстовое поле (т. е. на сколько строк нужно его прокрутить, чтобы стала видна последняя строка). Оно всегда равно индексу последней строки поля минус количество строк, которое можно одновременно вывести в видимой области, плюс один. Например, свойство `maxscroll` текстового поля на рис. 18.2 должно быть равно 4 (номер последней строки 6, минус 3 строки в видимой области, плюс 1). Обратите внимание, что `maxscroll` *не* равно количеству строк текста.

Свойство `maxscroll` можно получить (но не задать) с помощью *`textFieldName`*`.maxscroll`.

## Типичный код для прокрутки текста

В совокупности свойства `scroll` и `maxscroll` можно использовать для прокрутки текстового поля. Следующий код прокручивает текст вниз на одну строку при каждом щелчке по кнопке:

```
on (press) {
  if (textField.scroll < textField.maxscroll) {
    textField.scroll++;
  }
}
```

А вот код, который прокручивает текст вверх на одну строку при каждом щелчке:

```
on (press) {
  if (textField.scroll > 1) {
    textField.scroll--;
  }
}
```

В качестве примера простых кнопок прокрутки для использования в фильме загрузите образцы, имеющиеся в сетевом Хранилище кода.

В build 30 проигрывателя Flash 5, поставлявшейся с инструментом редактирования Flash 5, была программная ошибка, касающаяся вывода текстового поля. При прокрутке текстовых полей без сглаживания шрифтов не всегда исчезали остатки прокрученного текста. Чтобы избавиться от этой проблемы, поместите вокруг текста рамку, закрывающую остатки текста. В сборке 41 проигрывателя Flash 5, выпущенной в декабре 2000 года, эта ошибка была исправлена. Выяснить версию имеющегося проигрывателя можно с помощью глобальной функции *getVersion( )*.

## Событие _changed

Во Flash 4 и Flash 5 изменение содержимого текстового поля для ввода данных пользователем можно обнаружить с помощью события *_changed*, которого нет в документации. Когда пользователь вводит или удаляет текст в поле ввода данных, событие *_changed* запускает подпрограмму в стиле Flash 4 со специальным именем. Чтобы создать событие *_changed* для текстового поля, нужно выполнить такие шаги:

1. На любой временной диаграмме создать текстовое поле для ввода данных пользователем.
2. Дать полю имя `myField`.
3. На той же временной диаграмме присвоить кадру метку `myField_changed`.
4. Прикрепить любой код к кадру `myField_changed`. Например:

```
trace("myField was changed");
```

5. Выполнить экспорт фильма с помощью меню Control → Test Movie.
6. Ввести символы в текстовое поле `myField`. Выполняется код кадра `myField_changed`, и в окне Output появляется строка «myField was changed».

Разумеется, имя `myField` выбрано произвольно: можно выбрать любое имя для текстового поля, но и метка кадра должна получить соответствующее имя. Обратите внимание, что присвоение значения текстового поля с помощью ActionScript не влечет генерации события *_changed* для этого поля: *_changed* вызывается только нажатием клавиш.

Событие *_changed* не документировано. В будущих версиях будет, вероятно, реализован новый более стандартный метод обработки событий для текстовых полей.

# Поддержка HTML

Панель Character позволяет устанавливать гарнитуру, размер и начертание шрифта, но только для текстового поля в целом. Для задания стиля отдельных символов и ввода гипертекстовых ссылок нужно использовать HTML (появившийся во Flash 5 как возможность текстовых полей).

Можно использовать HTML как в динамических текстовых полях, так и в полях для ввода данных пользователем, но обычно текстовые поля HTML используются только для вывода. Чтобы добавить в текстовое поле поддержку HTML, выберите параметр HTML в панели Text Options.

Текстовые поля поддерживают только следующие теги HTML: `<B>`, `<I>`, `<U>`, `<FONT>`, `<P>`, `<BR>` и `<A>`.

## <B> (полужирный)

Тег <B> выводит текст полужирным шрифтом, при условии, что для рассматриваемого шрифта есть такая версия:

```
<B>This is bold text</B>
```

## <I> (курсив)

Тег <I> выводит текст курсивом, при условии, что для рассматриваемого шрифта есть такая версия:

```
<I>This is italic text</I>
```

## <U> (подчеркивание)

Тег <U> выводит помеченный им текст, подчеркнув его. Например:

```
<U>This is underlined text</U>
```

Поскольку текст ссылок не подчеркивается во Flash, следует использовать тег <U> для обозначения гиперссылок:

```
<A HREF="http://www.thesquarerootof-1.com"><U>Click here</U>
</A> to visit a neat site.
```

## <FONT> (настройка шрифта)

Тег <FONT> поддерживает следующие три атрибута:

*FACE*

Атрибут FACE задает имя используемого шрифта. Обратите внимание, что во Flash не поддерживается список из нескольких шрифтов, как в HTML. Flash пытается выводить только первый шрифт из перечисленных в атрибуте FACE. Например, в коде <FONT FACE="Arial, Helvetica">my text</FONT> Flash не станет выводить «my text» шрифтом Helvetica при отсутствии Arial, а воспользуется шрифтом по умолчанию.

*SIZE*

Атрибут SIZE задает размер текста, заключенного в теги, как фиксированный кегль (например, <FONT SIZE="18">) или в относительном виде. Относительному размеру предшествует знак + или –, и он определяется относительно кегля, указанного в панели Character. Например, если в панели Character указан кегль 14, то <FONT SIZE="-2"> выводит помеченный тегами текст кеглем 12 пунктов.

*COLOR*

Атрибут COLOR задает цвет помеченного тегами текста в виде шестнадцатеричного числа, которому предшествует символ (#). Напри-

мер: `<FONT COLOR="#FF0000">this is red text</FONT>`. Шестнадцатеричное число задается как значение RGB из трех двузначных чисел в диапазоне от `00` до `FF`. Обратите внимание, что во Flash атрибут `COLOR` реализован строже, чем в HTML: знак (#) обязателен, а названия цветов типа `"green"` или `"blue"` нельзя использовать в качестве значений `COLOR`.

Вот несколько примеров использования `<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>
```

Дополнительные важные сведения об использовании шрифтов во Flash см. в разделе «Вывод в формате HTML» далее в этой главе.

## <P> (абзац)

Тег `<P>` размечает абзацы, но во Flash он ведет себя совсем иначе, чем в HTML. Во-первых, теги `<P>` без закрывающего тега не вызывают перевод строки во Flash, как это происходит в обычном HTML. Обратите внимание на различие между выводом во Flash и веб-броузере:

```
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.
```

Во Flash перевод строки добавляется только тогда, когда есть закрывающий тег `</P>`, например:

```
<P> I hate filling out forms.</P> So sometimes I don't.
```

Более того, во Flash тег `<P>` вызывает перевод одной строки, так же как и `<BR>`, в то время как в веб-броузерах `<P>` обычно вызывает перевод двух строк. Рассмотрим следующий код:

```
<P>This is line one.</P><P>This is line two.</P>
```

Во Flash такой код будет выведен без промежутка между строками:

```
This is line one.
This is line two.
```

В веб-броузере строки будет выведены с разрывом:

```
This is line one.

This is line two.
```

Поскольку поведение тега <P> во Flash отличается от поведения, характерного для веб-броузеров, вместо него часто используется тег <BR>. Однако атрибутом ALIGN тега <P> можно по-прежнему пользоваться для выравнивания текста по центру, левому и правому краям, например:

```
<P ALIGN="CENTER">Centered text</P>
<P ALIGN="RIGHT">Right-justified text</P>
<P ALIGN="LEFT">Left-justified text</P>
```

## <BR> (перенос строки)

Тег <BR> вызывает перенос строки в середине текста и функционально равносилен escape-последовательности \n или ключевому слову newline. Рассмотрим две строки:

```
This is line one. <BR>This is line two.
This is line one. \nThis is line two.
```

Каждая из них выводится во Flash одинаково:

```
This is line one.
This is line two.
```

## <A> (якорь или гипертекстовая ссылка)

С помощью тега <A> создается гипертекстовая ссылка. При щелчке пользователем по тексту, размеченному тегами <A>, в броузер загружается документ, указанный в атрибуте HREF тега. Если проигрыватель Flash работает в автономном режиме, запускается веб-броузер, установленный на машине в качестве броузера по умолчанию, и документ загружается в этот броузер.

Общий синтаксис тега <A> такой:

```
<A HREF="documentToLoad.html">linked text</A>
```

Например, для ссылки на хорошую видеоигру можно воспользоваться кодом:

```
<A HREF="http://www.quake3arena.com/">nice game</A>
```

Как и в HTML, можно задать URL в абсолютном виде или относительно текущей страницы. Обычно ссылки, переход по которым происходит по тегу якоря, вызывают замену текущего фильма документом, указанным в HREF тега якоря. Однако этот тег может вызвать и запуск второго окна броузера. С помощью атрибута TARGET можно задать имя окна, в которое должен быть загружен документ, указанный в ссылке, например:

```
<A HREF="documentName" TARGET="windowName">linked text</A>
```

Если окна с именем *windowName* еще не существует, броузер открывает новое окно и присваивает ему имя *windowName*. Запускать каждый документ в собственном безымянном окне можно с помощью ключевого слова _blank, например:

```
<A HREF="mypage.html" TARGET="_blank">linked text</A>
```

Обратите внимание, что при открытии новых окон при помощи атрибута TARGET нельзя управлять размером или панелью инструментов нового окна. Для запуска через ссылку окон со специфическими размерами нужно применять JavaScript. Технология открытия вторичных окон с особыми настройками с помощью JavaScript описана по адресу:

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

Дополнительные сведения об организации связи с помощью JavaScript из ActionScript можно найти в описании глобальных функций *fscommand( )* и *getURL( )* в части III и в разделе «Выполнение JavaScript из ссылок HTML» далее в этой главе.

Атрибут TARGET можно использовать и для загрузки документов в кадры, например:

```
<A HREF="documentName" TARGET="frameName">linked text</A>
```

Теги якорей во Flash не всегда ведут себя в точности, как теги якорей HTML. Например, во Flash нельзя использовать атрибут NAME тега якоря, поэтому внутренние ссылки в тексте невозможны. Кроме того, ссылки Flash не подчеркиваются и не выделяются каким-либо другим способом. Подчеркивание и цвет должны задаваться вручную с помощью тегов <U> и <FONT>, описанных выше.

## Порядок перемещения по тегам якорей с помощью табуляции

Во Flash 5 теги якорей не добавляются к последовательности элементов фильма, обход которых осуществляется с помощью клавиши табуляции, и поэтому не доступны с клавиатуры. Если требуется доступ к содержимому с помощью клавиатуры или других устройств ввода, следует использовать для ссылок кнопки, а не теги якорей.

## Заключение значений атрибутов в кавычки

Вне Flash значения атрибутов могут заключаться в одинарные или двойные кавычки или задаваться вообще без кавычек. В большинстве веб-броузеров допустимы все из приведенных ниже тегов:

```
<P ALIGN=RIGHT>
<P ALIGN='RIGHT'>
<P ALIGN="RIGHT">
```

Однако во Flash указывать значения атрибутов без кавычек не разрешается. Например, синтаксис <P ALIGN=RIGHT> во Flash не допустим. В качестве границ значений атрибутов можно использовать как одинарные, так и двойные кавычки. Составляя значения текстовых полей, содержащие атрибуты HTML, нужно следить за правильным заключением атрибутов в кавычки и использовать один тип кавычек для ограничения самой строки, а другой – для ограничения значений атрибутов. Например:

```
// Оба следующие примера правильные
myText = "<P ALIGN='RIGHT'>hi there</P>";
myText = '<P ALIGN="RIGHT">hi there</P>';
// Этот пример вызовет ошибку, потому что двойные кавычки используются
// для ограничения как строки, так и атрибутов
myText = "<P ALIGN="RIGHT">hi there</P>";
```

Дополнительные сведения об использовании кавычек при формировании строк см. в разделе «Строковые литералы» главы 4 «Элементарные типы данных».

## Нераспознаваемые теги и атрибуты

Как и веб-броузеры, Flash игнорирует теги и атрибуты, которые ему не знакомы. Например, если присвоить во Flash текстовому полю HTML значение:

```
<P>Please fill in and print this form</P>
<FORM><INPUT TYPE="TEXT"></FORM>
<P>Thank you!</P>
```

то результатом будет:

```
Please fill in and print this form
Thank you!
```

Элементы FORM и INPUT не поддерживаются Flash и потому игнорируются. Аналогично при использовании элементов контейнеров, таких как <TD>, содержимое сохраняется, а разметка игнорируется. Например,

```
myTextField = "<TABLE><TR><TD>table cell text</TD></TR></TABLE>";
```

выводит следующую строку без форматирования таблицы:

```
table cell text
```

## Вывод в формате HTML

Текст HTML, введенный вручную в текстовое поле с помощью инструмента Text, не будет выведен как HTML. Для вывода на экран текста в формате HTML нужно с помощью ActionScript присвоить текст HTML динамическому текстовому полю, например:

```
myTextField = "<P><B>Error!</B> You <I>must</I> supply an email address!</P>";
```

Встраивание шрифта для текстового поля HTML задает только один стиль или один шрифт. Например, текстовое поле, для которого в панели Character задан шрифт полужирный Arial, будет поддерживать только символы полужирной гарнитуры Arial. Если с помощью HTML присвоить другой стиль Arial (например, курсив) или вообще другую гарнитуру (например, Garamond), то текст, заключенный в теги, вообще окажется невидимым, пока соответствующие шрифты не будут встроены в фильм!

Допустим, например, что создается текстовое поле с именем `output`. В панели Character для текстового поля `output` выберем Arial и установим Italic. В панели Text Options встроим весь шрифт Arial italic. Затем укажем, что `output` выводит HTML. Наконец, присвоим этому текстовому полю значение:

```
output = '<P><I>My</I>, what <B>lovely</B>'
         + '<FONT SIZE="24">eyes</FONT> you have!</P>';
```

При воспроизведении фильма в текстовом поле появится текст:

*My*

Все остальное, что было присвоено `output`, отсутствует! В HTML можно вывести текст только курсивом. Оставшемуся тексту нужны другие разновидности шрифта Arial, которые мы не встраивали: «what», «eyes» и «you have» не помечены курсивом, а «lovely» имеет полужирный шрифт.

Для каждой гарнитуры и разновидности шрифта, используемой в текстовом поле HTML, необходимо встроить соответствующий шрифт. Это можно сделать двумя способами:

- Создать фиктивное невидимое текстовое поле, для которого нужный шрифт выбран в панели Character и встроен в панель Text Options.
- Добавить новый символ шрифта в библиотеку фильма и экспортировать шрифт вместе с фильмом.

Вот действия, которые нужно выполнить, чтобы встроить в фильм полужирный Arial для использования в текстовом поле:

1. Выбрать меню Window → Library.
2. Выбрать меню Options → New Font. Появится диалоговое окно Font Symbol Properties.
3. В Font выбрать Arial.
4. В Style выбрать Bold.
5. В Name ввести `ArialBold` (это косметическое имя будет использоваться только в библиотеке).
6. В библиотеке выделить символ шрифта `ArialBold`.

7. Выбрать меню Options → Linkage.
8. В диалоговом окне Symbol Linkage Properties выбрать Export This Symbol.
9. В поле Identifier ввести `ArialBold`. Для наших задач вводимое здесь имя не имеет значения. Идентификаторы экспортированных символов используются только для библиотек совместного доступа.

Обратите внимание, что каждую разновидность стиля шрифта нужно встраивать отдельно. Если в текстовом поле используются Arial полужирный, Arial курсив и Arial полужирный курсив, то встраиваться должны все три разновидности шрифта. Подчеркивание не считается разновидностью шрифта, так же как кегль или цвет.

Однако если не включать никакие параметры Embed Fonts панели Text Options, то Flash полагается исключительно на шрифты, имеющиеся на машине пользователя, и тогда обычный текст и текст, выделенный полужирным шрифтом или курсивом, будут выводиться только тогда, когда соответствующие разновидности шрифта установлены на машине пользователя.

Чтобы обеспечить единообразный вывод на всех платформах и во всех системах, следует встроить все шрифты, необходимые текстовому полю.

## Ввод HTML

Обычно HTML используется в текстовых полях с целью вывода, но его можно и вводить в фильм через текстовое поле для ввода данных пользователем – обычное (не-HTML) или поле с включенным HTML.

При вводе обычного текста в текстовое поле для ввода данных пользователем с включенным HTML теги разметки HTML добавляются автоматически. Например, текст «Hi there» будет преобразован в значение HTML:

```
'<P ALIGN="LEFT"><FONT FACE="Arial" SIZE="10" COLOR="#000000">Hi there</FONT></P>'
```

Когда теги HTML вводятся пользователем в текстовое поле для ввода данных с включенным HTML, символы < и > преобразуются в `>` и `<`. Например, текст «`<B>hi there</B>`» будет преобразован в значение:

```
'<P ALIGN="LEFT"><FONT FACE="Arial" SIZE="10"
 COLOR="#000000">&lt;B&gt;hi there&lt;/B&gt;</FONT></P>'
```

Текстовые поля для ввода данных пользователем с включенным HTML можно использовать для создания очень простых систем ввода данных HTML.

Когда обычный текст или текст HTML вводится в обычное (не-HTML) текстовое поле для ввода данных пользователем, вводимый текст не

модифицируется. Обычные текстовые поля для ввода позволяют вводить в фильм необработанный код HTML без каких-либо искажений.

Пример, демонстрирующий текстовые поля для ввода данных с включенным HTML и обычные поля ввода, можно получить из сетевого Хранилища кода.

## Выполнение JavaScript из ссылок HTML

В большинстве веб-броузеров, поддерживающих JavaScript, можно выполнять команды JavaScript в тегах якорей, указывая протокол `javascript:` в качестве значения атрибута `HREF`. Например:

```
<A HREF="javascript:square(5);">find the square of 5</A>
```

В ActionScript тоже можно выполнять команды JavaScript из тега `<A>`, например:

```
myTextField = "<A HREF='javascript:alert(5);'>display the number 5</A>";
```

Однако для включения строковых значений в предложения JavaScript необходимо в качестве кавычек использовать элемент HTML `"`:

```
myTextField = "<A HREF='javascript:alert(&quot;hello world&quot;);'>"
              + "display hello world</A>";
```

## Вызов функций ActionScript из ссылок HTML

Из тегов Flash `<A>` можно вызывать не все предложения кода ActionScript, но *функции* ActionScript вызывать можно. Для вызова функции ActionScript из тега якоря используется следующий синтаксис:

```
<A HREF="asfunction:myFunctionName">invoke the function</A>
```

Не разрешается использовать оператор вызова функции ( ) при вызове функций ActionScript из тега якоря. Помимо вызова функции ActionScript из тега якоря, можно передать этой функции один параметр при помощи следующего синтаксиса:

```
<A HREF="asfunction:myFunctionName,myParameter">invoke the function</A>
```

где *myParameter* является значением передаваемого параметра. Внутри вызываемой функции *myParameter* всегда является строкой. Чтобы передать функции из якоря не один, а несколько элементов информации, нужно использовать разделитель в значении *myParameter* и самостоятельно разделить строку на части в функции. Вот, например, вызов функции *roleCall( )*, которой передаются два значения, разделенные символом «|»:

```
<A HREF="asfunction:roleCall,megan|murray">invoke the function</A>
```

А вот функция *roleCall( )*. Обратите внимание, как она разделяет значения с помощью метода *split( )*:

```
function roleCall (name) {
  var bothNames = name.split("|");
  trace("first name: " + bothNames[0]);
  trace("last name: " + bothNames[1]);
}
```

# Работа с выделением текста в текстовых полях

Когда пользователь выделяет часть динамического текстового поля или поля для ввода, позиции выделенных символов хранятся в особом встроенном объекте с именем *Selection*. С помощью объекта *Selection* можно узнать, какую часть текстового поля выделил пользователь, или даже выделить часть текстового поля программным способом. Объект *Selection* может также сообщить, которое из ряда текстовых полей выбрано в данный момент пользователем. Наконец, с помощью объекта *Selection* можно направить ввод с клавиатуры в определенное текстовое поле, предлагая пользователю ввести данные в указанном месте.

Чтобы узнать, как работать с выделением в текстовых полях, обратитесь к описанию объекта *Selection* в части III.

# Пустые текстовые поля и предложение for–in

Чтобы проверить значения всех переменных на временной диаграмме, можно воспользоваться предложением *for-in*, как показано в главе 6 «Предложения». Однако неопределенные текстовые поля (присутствующие на экране, но содержащие значение `undefined`) не проходятся предложением *for-in*. (Поля, содержащие пустую строку (`""`) или только пробелы, не считаются пустыми и потому проходятся в цикле.)

Невидимость неопределенных текстовых полей в циклах *for-in* может вызвать проблемы в сценариях, проверяющих ошибки. Сценарии, использующие цикл *for-in* для обхода текстовых полей, нужно писать так, чтобы учитывались неопределенные текстовые поля. Попробуем, например, проверить, есть ли в клипе с именем `formClip` переменные, содержащие пустую строку:

```
for (i in formClip) {
  if (formClip[i] == "") {
    trace(i + " is empty! don't submit the form!");
    break;
  }
}
```

В таком виде код не будет работать должным образом, поскольку неопределенные текстовые поля не перечисляются в цикле и не будут проверены. Для того чтобы неопределенные текстовые поля были перечислены в цикле *for-in*, необходимо явно присвоить пустую строку соответствующим переменным временной диаграммы. Например, прикрепим этот сценарий к кадру `formClip`, чтобы исправить предыдущий пример:

```
// Присвоить текстовым полям пустую строку, чтобы они
// стали видны в цикле for-in
formField1 = "";
formField2 = "";
```

# Вперед!

Мы почти достигли того неизбежного момента, когда путешествие с руководителем закончено, и вы начинаете разрабатывать собственные проекты и идеи. В предыдущих двух главах мы научились создавать формы, выводить данные на экран и получать данные, введенные пользователем. Последняя остановка перед справочным разделом – отладка кода – обучает технике выживания в предстоящих путешествиях по неизведанному.

# 19

# Отладка

Мы уже исследовали множество технических приемов, а также синтаксис для выполнения многих задач. Однако, начав самостоятельную работу с ActionScript, вы неизбежно будете совершать множество ошибок (особенно на первых порах, когда все еще случаются синтаксические и концептуальные ошибки). Не отчаивайтесь! Даже опытные программисты много времени тратят на *отладку* (исправление неправильно работающего кода).

Важно досконально проверять свою продукцию и в первую очередь находить программные ошибки. Это означает, что тестирование нужно проводить с различными броузерами и версиями этих броузеров на всех платформах, которые вы намереваетесь поддерживать. Проводите тестирование под различными версиями Windows и, если это возможно, с более старыми версиями подключаемых модулей Flash, которые можно найти по адресу:

*http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm*

Обсуждение тестирования и контроля качества (QA) выходит за рамки данной книги. Скажем лишь, что необходимо осуществлять тестирование и контроль качества, при этом следует иметь бланк сообщений об ошибках, в котором указываются платформа, броузер и его версия, версия подключаемого модуля Flash, а также описание последовательности выполняемых действий. Все это позволит воспроизвести ошибку, что станет первым шагом на пути к ее исправлению.

Отладка является важной частью программирования и тем, что отличает выдающихся программистов от программистов средней руки. Новички часто счастливы, если ранее возникавшая ошибка вдруг поче-

му-то исчезает. Опытный программист знает, что такая ошибка все равно всплывет на поверхность, причем в самый неподходящий момент, и, хотя она возникает лишь периодически (а возможно, именно поэтому), она требует дальнейшего расследования. С другой стороны, неопытные программисты часто прячутся от ошибок или теряют присутствие духа при ошибках очевидного происхождения, в то время как опытные программисты очень внимательно относятся к сообщениям об ошибках и знают, что те ошибки, которые легко воспроизвести, легче всего исправить.

Для успешной отладки требуются логика и дисциплина при проведении исследований, а также достаточное владение необходимыми средствами. В этой главе будут вкратце рассмотрены основы инструментов отладки и некоторые общие приемы для решения проблем с кодом. Запомните, что для отладки характерно постоянное оспаривание предположений, которые мы делаем. Какая-то проблема часто обусловлена совсем другой более общей проблемой (болезнью). Средства отладки применяются для того, чтобы выяснить, действительно ли код выполняет то, что задумано, а это приводит к пониманию и исправлению проявившейся ошибки (т. е. симптома).

# Средства отладки

В ActionScript имеются следующие инструменты для отладки:

- Функция *trace( )*
- Команда List Variables
- Команда List Objects
- Профайлер (Bandwidth Profiler)
- Отладчик (Debugger)

Все эти средства используются в режиме Test Movie. Чтобы войти в режим Test Movie, нужно экспортировать фильм из редактора с помощью меню Control → Test Movie.

Кроме этих формальных средств отладки есть сообщения об ошибках, которые Flash отправляет в окно Output во время экспорта фильма или при выполнении Check Syntax (проверка синтаксиса). (Check Syntax – это команда, выводимая при нажатии кнопки со стрелкой в правом верхнем углу панели Actions.) Сообщения об ошибках часто указывают точную причину проблемы вплоть до номера строки исходного кода, которым она вызвана. Полное объяснение всех сообщений об ошибках есть в фирменном руководстве Macromedia по ActionScript.

Обратите внимание на то, что не все ошибки влекут вывод сообщений. Например, если при вычислении получается неверный результат, то это программная ошибка, хотя и не вызывающая аварийного завершения работы броузера. Кроме того, есть два типа сообщений об ошиб-

ках: так называемые сообщения об ошибках *этапа компиляции*, возникающие при попытке экспорта сценария, и так называемые ошибки *времени исполнения*, которые возникают только тогда, когда фильм Flash воспроизводится и достигает той точки, в которой генерируется ошибка. Ошибки времени компиляции указывают на какие-то синтаксические ошибки, например, отсутствие скобок или закрывающей кавычки. Обращайтесь к части III «Справочник по языку», чтобы узнать точный синтаксис каждой команды, или к главе 14 «Лексическая структура» за разъяснением правильного синтаксиса ActionScript.

Ошибки времени выполнения могут быть весьма разнообразными и свидетельствовать о проблеме, вызванной не текущим рассматриваемым кодом, а неверным результатом более ранних действий. Допустим, например, что делается попытка использования значений, возвращенных командой *loadVariables( )*, отправленной серверу. Если сценарий Perl, отвечающий на команду, не предоставил верных данных в правильном формате, то исправлять нужно сценарий Perl. Сценарий Flash может быть совершенно верным, но не выполниться из-за получения неверных входных данных.

В итоге мы приходим к важной технологии – оборонительному программированию. Множества реальных и потенциальных ошибок можно избежать, если всегда проверять ситуации, чреватые проблемами, что называют контролем ошибок (*error checking*) либо, иногда, проверкой данных (*data validation*), если это относится к вводу, производимому пользователем. Например, прежде чем выводить вопросы вопросника, можно убедиться в том, что эти вопросы правильно загрузились. Можно проверить и каждый вопрос, чтобы быть уверенным в правильности его формата для вывода. Если полученные данные были введены неверным образом, следует вывести соответствующее сообщение об ошибке, которое позволит программисту или пользователю предпринять корректирующие действия.

## Функция trace()

В ActionScript мощнейшее средство выявления ошибок является также и простейшим: это функция *trace( )*. Как демонстрировалось на протяжении всей книги, *trace( )* направляет значение выражения в окно Output в режиме Test Movie. Например, если добавить в фильм такой код:

```
trace("hello world");
```

то в окне Output появится текст «hello world». Аналогично трассируется значение переменной:

```
var x = 5;
trace(x);  // Выводит 5 в окно Output
```

С помощью *trace( )* можно проверять состояние переменных, свойств и объектов и тем самым отслеживать ход выполнения кода. Источник проблемы часто можно обнаружить при проверке результата каждой операции сценария. Допустим, например, что некоторая функция должна возвращать значение, но с помощью команды *trace( )* обнаруживается, что возвращается значение `undefined` (которое никак не выводится в окно Output). В результате становится ясно, что нужно внимательнее рассмотреть функцию и проверить, правильно ли в ней используется команда *return* для возврата осмысленного значения.

## Команда List Variables

При воспроизведении фильма в режиме Test Movie можно проверить значения текущих переменных, определенных в фильме, с помощью команды Debug → List Variables. List Variables сообщает имена и местонахождение всех переменных, активных в фильме в данный момент, а также их значения. Так как функции и клипы хранятся в переменных, команда List Variables показывает также функции и клипы, имеющиеся в фильме.

В примере 19.1 приведен образец вывода после выполнения команды List Variables. Обратите внимание, что переменная `rate` показана как объявленная, но не определенная (`undefined`). Такие детали иногда трудно обнаружить с помощью *trace( )*, поскольку *trace( )* преобразует значение `undefined` в пустую строку (`""`).

*Пример 19.1. Образец выдачи команды List Variables*

```
Level #0:
  Variable _level0.$version = "WIN 5,0,30,0"
  Variable _level0.calcDist = [function]
  Variable _level0.deltaX = 194
  Variable _level0.deltaY = 179
  Variable _level0.rate = undefined
  Variable _level0.dist = 264
Movie Clip:  Target="_level0.clip1"
Movie Clip:  Target="_level0.clip2"
```

Обратите внимание, что обе команды – *trace( )* и List Variables – дают только мгновенный снимок. Часто требуется следить за изменением переменной во времени или проверять его многократно. Следить за изменениями переменной позволяет отладчик (Debugger), который будет обсуждаться позднее.

## Команда List Objects

Команда List Objects создает каталог текста, фигур, графических изображений и клипов, определенных в фильме. Чтобы выполнить ее, выберите меню Debug → List Objects, находясь в режиме Test Movie. Об-

ратите внимание, что List Objects не показывает объекты данных (экземпляры класса): о них сообщает команда List Variables.

В примере 19.2 приведен образец выдачи команды List Objects. Обратите внимание на явную пометку текстовых полей, которые могут редактироваться, и показ экземпляров клипов с автоматическими именами (например, `_level0.instance1`).

*Пример 19.2. Образец выдачи List Object*

```
Level #0: Frame=1
  Shape:
  Text: Value = "variables functions clip events startDrag stopDrag Math"
  Text: Value = "this movie demonstrates a little math, variables, movie clip events"
  Text: Value = "draggable distance"
  Text: Value = "calculator"
  Movie Clip: Frame=1 Target="_level0.instance1"
    Shape:
  Text: Value = "distance between clipstotal:horizontal:vertical:"
  Edit Text: Variable=_level0.dist Text="222"
  Edit Text: Variable=_level0.deltaX Text="174"
  Edit Text: Variable=_level0.deltaY Text="138"
  Movie Clip: Frame=1 Target="_level0.obj1"
    Shape:
  Movie Clip: Frame=1 Target="_level0.obj2"
    Shape:
```

Опять-таки, List Objects предоставляет только мгновенный снимок в данный момент времени. Чтобы получить текущие значения объектов после их изменения, команду нужно выполнить снова.

## Профайлер Bandwidth Profiler

Профайлер Bandwidth Profiler используется для моделирования загрузки фильма при разных скоростях модема. С помощью Bandwidth Profiler можно измерить производительность фильма, проверить код предварительной загрузки и проследить за положением воспроизводящей головки главного фильма. Включить Bandwidth Profiler можно следующим образом:

1. Находясь в режиме Test Movie, выбрать меню View→Bandwidth Profiler.
2. В меню Debug выбрать нужную скорость загрузки.
3. Для моделирования загрузки фильма на этой скорости выбрать меню View → Show Streaming.

На производительность Flash могут влиять многие факторы, в том числе используемые объекты и требования к проигрывателю. Например, при использовании больших растровых изображений, выводе сложных фигур, состоящих из многих кривых, и чрезмерном использовании альфаканалов производительность может существенно упасть. Уменьшение

времени загрузки и отображения объектов может сократить требования к пропускной способности каналов и мощности процессора, необходимых для выполнения ActionScript. Что говорить, ActionScript обычно работает значительно медленнее, чем компилируемые языки, такие как C.

С точки зрения ActionScript, больше всего времени требуют операции, которые должны ждать окончания отправки или получения данных, либо повторяющиеся многократно (например, исследование большого массива).

Отображение элементов в окне Output происходит очень медленно по сравнению с «невидимыми» операциями ActionScript. Если простой фильм кажется чрезмерно прерывистым, попробуйте отключить все предложения *trace( )* или воспроизвести фильм в режиме, отличном от Test Movie.

Обсуждение того, как писать оптимальный код, находится за рамками данной книги, но все же мы дадим некоторые несложные советы:

- Не повторяйте операцию многократно в цикле, если того же результата можно добиться, выполнив ее один раз вне цикла.
- Не следует дожидаться в цикле совершения некоторого события. Ожидание может оказаться долгим или событие вообще никогда не произойдет, а производительность снизится или приложение окажется заблокированным. Вместо этого используйте обработчики событий, такие как *on (load)*, запускаемые при возникновении или завершении некоторого события.
- При всякой возможности обобщайте свой код (возможно, даже используя для этого интеллектуальные клипы). Это сокращает размер кода, который необходимо загрузить. Например, вместо двух почти идентичных процедур по 5 Кбайт можно написать одну более общую процедуру и вызывать ее дважды с разными параметрами, что позволит сэкономить 5 Кбайт. (Об обобщении кода рассказывается в главе 9 «Функции», а интеллектуальные клипы обсуждаются в главе 16 «Среда редактирования ActionScript»).
- Если используется проигрыватель Flash 5, а оптимальная работа ActionScript для всего проекта критична, попробуйте использовать вместо новейших технологий прежний синтаксис в стиле Flash 4. Во Flash 5 некоторые операции осуществляются быстрее, если сформулировать их в синтаксисе Flash 4. Например, функция *substring( )* Flash 4 быстрее, чем методы *substring( )* и *substr( )* Flash 5, а *Tell Target* из Flash 4 быстрее, чем точечная запись Flash 5.
- Экспортируйте фильмы, удаляя предложения *trace( )* путем задания опции Publish Settings → Flash → Options → Omit Trace Actions.

- Помните, что удаление клипа и прикрепление его заново обходится дороже, чем перемещение существующего клипа; при всякой возможности повторно используйте входящие в фильм элементы.
- Перечень общих приемов оптимизации Flash можно найти по адресу: *http://www.macromedia.com/support/flash/publishexport/stream_optimize/stream_optimize.html*.

## Отладчик

Отладчик (Debugger) служит весьма полезным инструментом, который предоставляет организованный доступ к значениям свойств, объектам и переменным фильма и даже позволяет изменять значения переменных на этапе исполнения.

Чтобы включить отладчик, выберите в среде разработки Flash меню Control → Debug Movie (*не* в режиме Test Movie). Отладчик можно использовать и в веб-броузере, при условии что:

- Просматриваемый фильм был первоначально экспортирован с разрешенной отладкой.
- Проигрыватель, используемый для просмотра фильма, поддерживает отладку.
- Во время отладки выполняется редактор Flash.

Чтобы экспортировать фильм с разрешением отладки в броузере, выберите меню File → Publish Settings → Flash → Debugging Permitted, после чего можно задать пароль для защиты своего кода от любопытствующих глаз. Чтобы установить в броузере проигрыватель с поддержкой отладки, воспользуйтесь программами установки из папки */Players/Debug/* в каталоге на жестком диске, куда был установлен Flash. Для включения отладки во время просмотра фильма щелкните по фильму правой кнопкой мыши под Windows (Ctrl-щелчок на Macintosh) и выберите Debugger.

Не для всех версий Flash Player есть соответствующий Player с поддержкой отладки. Поищите на сайте поддержки Macromedia *http://www.macromedia.com/support/flash* новейшую версию проигрывателя с поддержкой отладки.

В верхней части отладчика (Display List) показывается иерархия клипов фильма. Для исследования свойств и переменных конкретного клипа выделите его в Display List. В нижней части Debugger есть три закладки – Properties, Variables и Watch, которые динамически обновляются, показывая свойства и переменные выбранного клипа. Чтобы задать значение свойства или переменной, дважды щелкните по значению и введите новые данные. Чтобы выделить один или несколько элементов для удобного просмотра, выделите их на вкладке Properties или Variables, после чего выберите Add Watch при помощи кнопки

со стрелкой в правом верхнем углу Debugger. Все «контролируемые» переменные будут добавлены во вкладку Watch (она позволяет одновременно просматривать переменные из разных клипов).

Дополнительные данные относительно механики использования отладчика Flash ищите в подробной документации Macromedia в разделе «Troubleshooting ActionScript» Справочного руководства по ActionScript. Если вы потеряли свое справочное руководство, то его можно взять на сайте Macromedia по адресу *http://www.macromedia.com/support/flash* либо в меню Help редактора Flash.

# Методика отладки

Взглянем кратко на некоторые приемы, используемые при отладке кода. Отладка может быть разбита на три отдельных этапа:

- Обнаружение и воспроизведение проблемы
- Выяснение источника проблемы
- Устранение проблемы

## Обнаружение ошибок

Очень часто проблемы кода выясняются в процессе активного программирования. Это означает, что пишется некоторый код, тестируется фильм, и оказывается, что он работает не так, как нужно. Проблема обнаружена.

Чем раньше обнаружена проблема, тем лучше. Поэтому процесс написания кода должен быть постоянным чередованием приливов и отливов: написать несколько строк, экспортировать фильм, проверить, что эти строки работают так, как предполагалось, написать еще несколько строк, экспортировать фильм и т. д. Убедитесь в правильности работы каждого компонента программы, прежде чем тестировать программу в целом. Постарайтесь не увлечься написанием сложного тела программы без частой попутной проверки ее работоспособности.

Не считайте свой фильм совершенным лишь потому, что не можете обнаружить в нем ошибки сами. Всегда выделяйте время для внешней проверки пользователями, особенно если создаваемый вами код входит в продукт или сервис, предназначенный для некоего клиента. Как уже говорилось, реализуйте проверку ошибок, чтобы исключить возникновение проблем из-за некорректного ввода данных. Например, если вы пишете функцию, которой нужен целочисленный аргумент, можно использовать оператор *typeof* для проверки правильности типа входных параметров. Проверяйте также *граничные значения,* например, очень большие или очень маленькие, а также отрицательные значения и ноль.

Не стоит недооценивать поиск *минимума воспроизводимых действий*, с помощью которых можно воссоздать проблему. Они должны представлять собой наикратчайший путь к воспроизведению ошибки. Отчет об ошибке типа: «Я воспроизводил фильм в течение часа, а потом все умерло» принесет мало пользы. Полезные отчеты об ошибках содержат последовательность действий, например:

1. Ввести 0 в качестве количества лет.
2. Щелкнуть по кнопке Calculate.
3. В поле результата вместо суммы в долларах выводится «NaN».

## Выявление источника ошибки

Признание наличия ошибки является лишь началом пути к ее устранению. Первейшей задачей является выяснение источника ошибки, как бы высоко он ни находился. Ошибку можно представлять себе как сердечный приступ, причиной которого может быть нарушение режима питания, происходившее за годы до него. Сердечный приступ служит просто наиболее явственно проявившимся симптомом, а в исправлении нуждается нечто более раннее в процессе. Большинство ошибок вызывается ложными предположениями: мы предполагаем, что правильно ввели имя переменной, хотя это не так, или что в текстовом поле находятся числовые данные, а их там нет. Путем выполнения ряда предложений *trace( )* или команд Debugger, или List Variables можно сравнить наши предположения с тем, как интерпретатор воспринимает наш код.

Вот, например, код, содержащий ошибку. Он ошибочно присваивает переменной `status` строку «equal»:

```
var x = 11;
isTen(x);
function isTen(val) {
  if (val = 10) {
    status = "equal";
  }
}
```

Чтобы выяснить, что неверно в коде, сравним *предполагаемое* действие кода с тем, что он делает *в действительности*, двигаясь вперед по шагам:

```
// Переменная x должна быть установлена равной 11
var x = 11;

// Посмотрим, происходит ли это в действительности
trace(x);     //...выводит: 11

// Здесь должна быть вызвана функция isTen()
isTen(x);
```

```
// Теперь обратимся к нашей функции
function isTen(val) {
  // Проверим, действительно ли вызвана функция
  trace("была вызвана isTen ");     //...выводит: "была вызвана isTen"

  // Проверим, правильно ли передан параметр
  trace("val равна " + val);        //...выводит: "val равна 11"
```

Остановимся на мгновение. Обратите внимание: пройден почти весь код, и пока что все работает так, как предполагалось. Переменная задана правильно; *isTen( )* вызвана и получила правильный аргумент.

Многие ошибки происходят потому, что код, который, как нам кажется, выполнялся, вообще не был достигнут! С помощью предложений *trace( )* можно убедиться, что вход в некоторый участок кода был осуществлен.

Методом исключения выясняется, что проблема кода заключается либо в условном предложении `if(val = 10)`, либо в присваивании текстовому полю: `status = "equal"`. Следующим проверяем условное предложение, выводя с помощью *trace( )* значение условного выражения (предполагается `true` или `false`):

```
trace(val = 10);
```

Эврика! В окне Output выводится `10`, а не `true` или `false`, как ожидалось.

При ближайшем рассмотрении выясняется, что условное выражение является присваиванием, а не сравнением! Пропущен знак равенства в операторе проверки равенства. Выражение `if(val = 10)` на самом деле должно иметь вид `if(val == 10)`.

Очевидно, не все ошибки столь просты, как это условное предложение (которое является весьма распространенной ошибкой), но использованный подход применим в большинстве случаев поиска ошибки: выполнить ряд функций *trace( )*, чтобы создать текущий пошаговый отчет о фактическом выполнении кода фильма и использовать отладчик, как советуется в документации Macromedia.

## Часто встречающиеся причины программных ошибок

В табл. 19.1 перечислены некоторые часто встречающиеся причины программных ошибок в ActionScript.

*Таблица 19.1. Ошибки, встречающиеся в ActionScript*

| Проблема | Описание |
|---|---|
| Неверное расположение кода | Всякий код должен быть прикреплен к клипу, кадру или кнопке. Следите, чтобы код был прикреплен туда, куда нужно, по заголовку панели Actions: если код прикреплен к кадру, заголовок панели Actions – Frame Actions; если код прикреплен к клипу или кнопке, заголовком панели Actions будет Object Actions. Если нужно расположить код в некотором конкретном кадре, в начале написания кода следите за тем, чтобы этот кадр был выделен на временной диаграмме и чтобы был ключевой кадр, куда должен быть помещен код. Если нужно, чтобы код был в клипе или кнопке, обеспечьте выделение соответствующего объекта на рабочем столе перед началом ввода кода. Чтобы точно проследить, куда прикрепляется код, используйте Movie Explorer (Window → Movie Explorer). |
| Отсутствие обработчика события | Код, прикрепляемый к клипам и кнопкам, *обязан* находиться в обработчике события. Для клипов используйте:<br>`onClipEvent (event) {`<br>`  // statements`<br>`}`<br>Для кнопок используйте:<br>`on (event) {`<br>`  // statements`<br>`}`<br>где *event* является именем обрабатываемого события. Ошибка «Statement must appear within on handler» указывает на отсутствие обработчика события. См. главу 10. |
| Неправильная ссылка на клип | Ссылка на клип, который не существует или имеет неверный формат. Проверьте, всем ли экземплярам даны имена и соответствуют ли эти имена указанным ссылкам. О том, как правильно составлять ссылки на клипы, читайте в разделе «Ссылки на вложенные клипы» главы 13. |
| Непредусмотренное преобразование типа | Неожиданный результат преобразования данных. Например, `3 + "4"` дает строку «34», а не число 7. Аналогично, строка «true» преобразуется в булево значение `false`! См. правила преобразования типов в главе 3. Проверяйте типы данных с помощью оператора *typeof*. |
| Отсутствие точки с запятой | Предложение преждевременно завершено из-за отсутствия точки с запятой. О том, как правильно использовать точку с запятой, читайте в главе 14. |
| Неверный символ кавычки | Строка содержит непреобразованный в escape-последовательность символ, конфликтующий со строковым литералом. См. раздел «Строковые литералы» в главе 4. |

*Таблица 19.1 (продолжение)*

| Проблема | Описание |
|---|---|
| Неверное использование данных текстового поля | Текстовое поле используется как число или другой тип данных, отличный от строки. Значения, вводимые пользователем в текстовые поля, всегда являются строками и должны быть вручную преобразованы перед использованием в качестве другого типа данных. |
| Проблемы области видимости | Ссылка на переменную, свойство или функцию не учитывают область видимости. Например, предложение в обработчике клипа пытается вызвать функцию, областью видимости которой является временная диаграмма родителя этого клипа. См. раздел «Область видимости обработчика событий» в главе 10, «Область видимости переменных» в главе 2 и «Доступность и срок жизни функций» в главе 9. |
| Смешение глобальной функции с методом | Имена некоторых глобальных функций совпадают с именами методов клипов. Иногда это служит источником проблем. См. раздел «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13. |
| Не загружено содержимое | Ссылка на клип, свойство, функцию или переменную не может быть разрешена, поскольку содержимое еще не загружено. Убедитесь в завершении загрузки с помощью свойства *MovieClip*.`_framesloaded`, как показано в части III. |
| Неверное использование заглавных букв | В некоторых ключевых словах ActionScript учитывается регистр букв. Если ошибочно записать *onClipEvent* как *onclipevent*, ActionScript решит, что вызывается пользовательская функция *onclipevent*, а не используется ключевое слово встроенного обработчика события *onClipEvent*. В результате будет сообщено об ошибке при обнаружении { в начале блока предложений *onClipEvent* (интерпретатор ищет точку с запятой, обозначающую конец того, что он считает вызовом функции *onclipevent*). См. раздел «Чувствительность к регистру» в главе 14. |

## Исправление ошибок

В некоторых случаях способ исправления выявленной ошибки очевиден. Например, если обнаружена ошибка, вызванная отсутствием кавычки в строке, она исправляется добавлением пропущенной кавычки.

В сложных программах исправление ошибок может представлять серьезную проблему. Если исправление ошибки оказывается затруднительным делом, руководствуйтесь следующими советами:

- Не бойтесь переписать код заново. Во многих случаях лучшим способом исправить очень сложный код будет повторное проектирование системы и написание кода сначала. Повторное написание про-

граммы почти всегда происходит более быстро и гладко, чем ее первоначальное создание. С этим согласно большинство экспертов (например, в Quake III ядро Quake II было полностью переписано). При этом новый код все равно нуждается в отладке. Не выбрасывайте вполне пригодный код, когда срок завершения работы уже близок. Оставьте то, что хорошо работает, и перепишите только код, вызывающий проблемы.

- Выделите проблематичные компоненты в самостоятельные тестовые фильмы. Работайте отдельно над каждой составляющей системы, а затем объединяйте работающие части по одной.
- Пусть ваш код посмотрит коллега. Не бойтесь. Любой программист стыдится кода, написанного им годом раньше.
- Обратитесь за помощью к одному из ресурсов, приведенных в приложении A «Ресурсы». Например, почтовый список рассылки FlashCoders посвящен исключительно вопросам ActionScript.

Множество добрых советов по технике программирования можно найти в книгах «*Extreme Programming Explained*» Кента Бека (Kent Beck) издательства Addison Wesley и «*Code Complete*» Стива Макконнелла (Steve McConnell) издательства Microsoft Press.

# Вперед!

Наш разговор на ActionScript закончен, а ваш только начинается. Прочтя часть I «Основы ActionScript» и часть II «Приложения ActionScript», вы научились разговаривать на ActionScript; настало время применить эти знания в собственных проектах. Пока вы не начали, несколько мыслей на прощание:

- Изучение программирования, как любого вида искусства, – процесс, а не событие. Пока вы занимаетесь программированием, вы получаете новые знания о программировании. Посмотрите на пример вопросника с вариантами ответов из глав 1, 9, 11 и 13 – мы переделывали его четыре раза! Каждый раз мы уточняли свой подход, добавляли новые функции и узнавали нечто новое. Так же как каждая картина дает художнику какое-то новое понимание предмета, создание и переработка приложений открывают программисту новые подходы.
- Практическую помощь вам окажет часть III, в которой подробно описываются встроенные функции, свойства, классы и объекты ActionScript. Необязательно читать ее сначала и до конца, но следует просмотреть каждую тему, чтобы почувствовать возможности, предоставляемые ActionScript.
- Чаще перечитывайте эту книгу и часть III. С каждым разом вы будете глубже проникать в суть благодаря более высокой степени понимания и способности связывать понятия с практическим опы-

том. Рассматривайте часть III как словарь и всегда держите ее под рукой при работе.

- Существует быстро расширяющееся сообщество разработчиков Flash, которое предлагает идеи и решения и, что еще важнее, готово делиться исходным кодом! Как можно больше занимайтесь разбором кода. Выделяйте то, что вам непонятно, и ищите эти темы в данной книге. Macromedia поддерживает список сайтов и почтовых списков рассылки, посвященных Flash, по адресу: *http://www.macromedia.com/support/flash/ts/documents/flash_websites.htm*.
- Не ограничивайте свое изучение ActionScript только данной книгой. Постарайтесь взглянуть на вещи под разными углами, обращаясь к другим источникам знаний и вдохновения, например, к перечисленным в предисловии. Большой список ресурсов Flash вы найдете также по адресу *http://www.moock.org/moockmarks*, а рецензии на стоящие книги по Flash – по адресу *http://www.moock.org/webdesign/books*.
- Наконец, не забывайте заглядывать на сайт поддержки данной книги на *http:// www.moock.org/asdg*, где есть масса примеров программ, технических заметок и обсуждений новых тем.

И на этом я желаю вам удачного написания программ! Пожалуйста, сделайте как-нибудь лишний проход в цикле *while* для меня лично. :)

# III

# Справочник по языку

В этой части объясняется применение всех встроенных классов, объектов, функций, свойств и обработчиков событий, которые имеются в ActionScript. Вы будете регулярно обращаться к части III при решении конкретных задач ежедневной работы программиста.

- «Справочник по языку ActionScript»

# Справочник по языку ActionScript

В данном *справочнике по языку* документируются все поддерживаемые ActionScript классы и объекты, с описанием их общего назначения, применения, свойств, методов и обработчиков событий. Помимо этого, справочник охватывает все глобальные функции и глобальные свойства (т. е. те, которые не закреплены за каким-либо классом или объектом, а могут использоваться самостоятельно в любом месте фильма).

Все описания расположены в справочнике в алфавитном порядке. Например, глобальная функция *duplicateMovieClip( )* представлена после класса *Date* (а не, скажем, в отдельном разделе «Глобальные функции»). Однако в заголовке каждой статьи ясно указан тип описываемого элемента, поэтому элементы с одинаковыми названиями нельзя спутать между собой. Например, *duplicateMovieClip( )* может иметь формат глобальной функции или метода клипа, поэтому даются две статьи: *duplicateMovieClip( ) глобальная функция* и *MovieClip.duplicateMovieClip( ) метод*. Если неясно, где искать какой-либо элемент, обратитесь к предметному указателю в конце книги.

## Глобальные функции

Глобальные функции – это встроенные функции, которые доступны 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. 1 перечислены глобальные функции, имеющиеся во Flash 5 ActionScript.

*Таблица 1. Глобальные функции ActionScript*

| | | |
|---|---|---|
| *Boolean( )* | *call( )* | *Date( )* |
| *duplicateMovieClip( )* | *escape( )* | *eval( )* |
| *fscommand( )* | *getProperty( )* | *getTimer( )* |
| *getURL( )* | *getVersion( )* | *gotoAndPlay( )* |
| *gotoAndStop( )* | *#include* | *int( )** |
| *isFinite( )* | *isNaN( )* | *loadMovie( )* |

*Таблица 1 (продолжение)*

| | | |
|---|---|---|
| *loadMovieNum( )* | *loadVariables( )* | *loadVariablesNum( )* |
| *maxscroll* | *newline* | *nextFrame( )* |
| *nextScene( )* | *Number( )* | *parseFloat( )* |
| *parseInt( )* | *play( )* | *prevFrame( )* |
| *prevScene( )* | *print( )* | *printAsBitmap( )* |
| *printAsBitmapNum( )* | *printNum( )* | *random( )** |
| *removeMovieClip( )* | *scroll* | *setProperty( )* |
| *startDrag( )* | *stop( )* | *stopAllSounds( )* |
| *stopDrag( )* | *String( )* | *targetPath( )* |
| *tellTarget( )** | *toggleHighQuality( )** | *trace( )* |
| *unescape( )* | *unloadMovie( )* | *unloadMovieNum( )* |
| *updateAfterEvent( )* | | |

* Устаревшая во Flash 5.

# Глобальные свойства

К глобальным свойствам, как и к глобальным функциям, можно обращаться из любого сценария в фильме. В них хранится информация, используемая в любой точке кода. Многие из них оказывают влияние на Flash Player в целом, а не только на конкретный клип или фильм.

В табл. 2 перечислены глобальные свойства, имеющиеся во Flash 5.

*Таблица 2. Глобальные свойства ActionScript*

| Имя свойства | Описание |
|---|---|
| `_focusrect` | Выделенное состояние кнопок, активируемых через клавиатуру |
| `_highquality` | Качество вывода Player* |
| `Infinity` | Константа, представляющая значение Infinity (бесконечность) *числового* типа |
| `-Infinity` | Константа, представляющая значение –Infinity (отрицательная бесконечность) *числового* типа |
| `_level`*n* | Уровень документа Player |
| `NaN` | Not-A-Number; значение *числового* типа, представляющее неверные числовые данные |
| `_quality` | Качество вывода Player |
| `_root` | Ссылка на главную временную диаграмму текущего уровня |
| `_soundbuftime` | Количество секунд предварительной загрузки потокового звука |
| `$version`* | Версия Flash Player |

* Устарело во Flash 5.

# Встроенные классы и объекты

В данном справочнике предполагается, что вам понятны термины *класс*, *объект* и *экземпляр*, исчерпывающим образом обсуждавшиеся в главе 12 «Объекты и классы». Встроенные классы ActionScript используются для создания объектов, которые могут управлять фильмом и манипулировать данными. ActionScript имеет такие встроенные классы: *Array*, *Boolean*, *Color*, *Date*, *MovieClip*, *Number*, *Object*, *Sound*, *String*, *XML*, *XMLNode* и *XMLSocket*. Для создания экземпляра конкретного класса используется функция *конструктора* класса и оператор *new*. Например, для создания нового объекта класса *Color* следующим образом используется конструктор *Color*:

```
myColor = new Color(_root);
```

Для каждого класса в справочнике по языку статья *Конструктор* показывает, как создавать объекты данного конкретного класса (т. е. демонстрирует 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 также 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.

Особые встроенные объекты ActionScript – *Arguments*, *Key*, *Math*, *Mouse* и *Selection* – приводятся в этой книге в алфавитном порядке, перемежаясь 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 (т. е. для их образования не используется оператор *new*). Напротив, это предопределенные 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, используемые для документирования каждого элемента справочника по языку, описываются в табл. 3.

*Таблица 3. Заголовки в справочнике по языку*

| Заголовок | Описание |
|---|---|
| Доступность | Указывает время ввода элемента в ActionScript (или превращения в Action, если элемент предшествует формальному ActionScript). Также отмечает обратную совместимость и статус устаревшего элемента |

*Таблица 3 (продолжение)*

| Заголовок | Описание |
|---|---|
| Краткий обзор | Иллюстрирует абстрактный синтаксис, требуемый элементом. Код, выделенный курсивом, обозначает текст, который должен вводиться программистом |
| Аргументы | Относится только к методам и функциям. Описывает параметры, используемые методом или функцией и перечисленные в Synopsis |
| Возвращаемое значение | Относится только к методам и функциям. Описывает значение, возвращаемое методом, если таковое имеется. Если метод не возвращает никакого значения, этот заголовок опускается |
| Тип доступа | Относится только к свойствам. Описывает, можно ли значение свойства только извлекать («Только для чтения»), или как извлекать, так и присваивать («Чтение/запись») |
| Описание | Разъясняет действие элемента и возможности его использования в практических сценариях |
| Особенности применения | Описывает примечательные особенности элемента |
| Ошибки в реализации | Описывает известные проблемы, связанные с элементом |
| Пример | Предлагает пример кода, в котором используется элемент |
| См. также | Список перекрестных ссылок на родственные темы |

# Алфавитный справочник по языку

В последующих статьях документированы объекты и классы ActionScript. Чтобы найти операторы и предложения, о которых говорится в других местах книги, обратитесь к предметному указателю.

**Arguments объект** доступ к параметрам функции и текущей функции

***Доступность*** Flash 5

***Краткий обзор*** `arguments[`*`elem`*`]`
`arguments.`*`propertyName`*

***Свойства***

*callee* Ссылка на вызываемую функцию.

*length* Число параметров, 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 так далее. В качестве объекта *Arguments* хранит свойство `callee`, которое можно использовать для идентификации или вызова текущей функции.

***См. также***

раздел «Объект arguments» в главе 9 «Функции».

---

**arguments.callee свойство** — ссылка на выполняемую в данный момент функцию

***Доступность*** Flash 5

***Краткий обзор*** `arguments.callee`

***Тип доступа*** Чтение/запись

***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();
  }
};
```

---

**arguments.length свойство** — количество параметров, переданных в аргумент

***Доступность*** Flash 5

***Краткий обзор*** `arguments.length`

***Тип доступа*** Чтение/запись

***Описание***

Свойство `length` хранит целое число, представляющее количество элементов в массиве `arguments`, равное числу параметров, переданных функции, выполняющейся в данное время.

***Пример***

С помощью свойства `length` объекта `arguments` можно определить, была ли функция вызвана с правильным количеством параметров. В следующем

примере проверяется, было ли *someFunction( )* передано два параметра. Проверка правильности типа переданных аргументов оставляется читателю в качестве упражнения. (Совет: см. оператор *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** поддержка упорядоченных списков данных

***Доступность*** Flash 5

***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 значений, которые должны быть присвоены элементам массива.

***Свойства***

*length* Количество элементов в массиве (включая пустые).

***Методы***

*concat( )* Создать новый массив, дописав новые элементы к существующему массиву.

*join( )* Преобразовать массив в строку.

*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.

*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 объекте массива. Полное описание, что такое массивы, и как ими пользоваться, а также подробное определение используемой в данном разделе терминологии см. в главе 11 «Массивы». См. также операторы [] и ., использующиеся для доступа к элементам массивов и описанные в главе 5 «Операторы».

***Особенности применения***

Если конструктор *Array* вызывается с одним целочисленным аргументом, то этот аргумент используется для задания длины нового массива, а не как значение первого элемента. Если конструктору передаются два или более аргументов, или один аргумент, но не числовой, то аргументы используются в качестве начальных значений элементов массива, а длина массива определяется числом заданных аргументов.

---

**Array.concat( ) метод** создать новый массив путем расширения существующего

***Доступность*** Flash 5

***Краткий обзор*** *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, отдельно добавляется каждый элемент этого массива. 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 массив ["apples", "oranges", "bananas"]
```

```
myListB = myListA.concat("bananas");

// Создать еще один новый массив
myListC = new Array("grapes", "plums");

// Присвоить myListD массив ["apples", "oranges", "bananas", "grapes", "plums"]
myListD = myListB.concat(myListC);

// Присвоить myListA массив ["apples", "oranges", "bananas"]
myListA = myListA.concat("bananas");

// Создать массив
settings  = ["on", "off"];
// Приписать массив, содержащий вложенный массив
options = settings.concat(["brightness", ["high", "medium", "low"]]);
// options станет: ["on", "off", "brightness", ["high", "medium", "low"]]
// не: ["on", "off", "brightness", "high", "medium", "low"]
```

***См. также***

*Array.push( ), Array.shift( ), Array.splice( )*; раздел «Метод concat()» в главе 11.

---

**Array.join( ) метод** преобразовать массив в строку

***Доступность*** Flash 5

***Краткий обзор***

```
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 используется запятая.

***Возвращаемое значение***

Строка, образованная из всех элементов *array*, преобразованных в строки и разделяемых *delimiter*.

***Описание***

Метод *join( )* возвращает строку, созданную путем объединения всех элементов массива, для чего:

1. Преобразует каждый элемент массива в строку (пустые элементы преобразуются в пустую строку).
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 разделителем, использовавшимся при вызове *join( )*.

***Пример***

```
fruit = new Array("apples","oranges","bananas","grapes","plums");
// Присвоить fruitString значение "apples,oranges,bananas,grapes,plums"
fruitString = fruit.join();
// Присвоить fruitString значение "apples-oranges-bananas-grapes-plums"
fruitString = fruit.join("-");
```

***См. также***

*Array.toString( )*, *String.split( )*; раздел «Метод join()» в главе 11.

---

**Array.length свойство** количество элементов в массиве

***Доступность*** Flash 5

***Краткий обзор*** `array.length`

***Тип доступа*** Чтение/запись

***Описание***

Свойство `length` является неотрицательным целым числом, задающим количество элементов в массиве. Массив без элементов имеет длину 0, массив из двух элементов имеет дину 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 по 8 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.

Присвоение значения свойству `length` изменяет количество элементов в массиве. Если увеличить `length`, то в конец массива будут добавлены пустые элементы, а если уменьшить `length`, то существующие элементы удаляются из конца массива. Свойство `length` автоматически изменяется, если добавить или удалить элементы с помощью методов класса *Array*. Свойство `length` отражает только пронумерованные элементы: оно не включает именованные элементы массива, которые рассматриваются как свойства массива.

***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***

раздел «Определение размера массива» в главе 11.

**Array.pop( ) метод** удалить последний элемент массива

***Доступность*** Flash 5

***Краткий обзор*** `array.pop()`

***Возвращаемое значение***

Значение последнего элемента массива *array*. Сам 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());  // теперь myList: ["one", "two"]
```

***См. также***

*Array.push( )*, *Array.shift( )*, *Array.splice( )*;разделы «Удаление элементов из массива» в главе 11 и «Оператор delete» в главе 5.

---

**Array.push( ) метод** добавить один или более элементов в конец массива

***Доступность*** Flash 5

***Краткий обзор*** `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 к концу массива список значений 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 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( )*; раздел «Добавление элементов в массив» в главе 11.

**Array.reverse( ) метод** изменяет порядок элементов массива на противоположный

***Доступность*** Flash 5

***Краткий обзор*** *array*.reverse()

***Описание***

Метод *reverse* изменяет порядок элементов массива на противоположный, меняя местами последний элемент с первым, предпоследний со вторым и так далее.

***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** удалить первый элемент массива

***Доступность*** Flash 5

***Краткий обзор*** *array*.shift()

***Возвращаемое значение***

Значение первого элемента *array*, который при этом 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 ["b", "c"]
```

***См. также***

*Array.pop( )*, *Array.splice( )*, *Array.unshift( )*; раздел «Удаление элементов из массива» в главе 11.

---

**Array.slice( ) метод** создать новый массив из подмножества элементов существующего массива

***Доступность*** Flash 5

***Краткий обзор*** *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* ука-

зывает номер элемента, отсчитываемый в обратном порядке 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. Новый массив является подмножеством 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( )*; раздел «Метод slice()» в главе 11, «Оператор delete» в главе 5.

---

**Array.sort( ) метод** сортировать элементы массива

***Доступность*** Flash 5

***Краткий обзор***

```
array.sort()
array.sort(compareFunction)
```

***Аргументы***

*compareFunction* Функция, которая указывает, как нужно сортировать *array*.

***Описание***

При вызове метода *sort( )* без аргументов элементы массива временно преобразуются в строки и упорядочиваются в соответствии с кодовыми позициями этих строк (приблизительно в алфавитном порядке). 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*, являющейся 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 предшествовать второму, положительное число, если первый элемент должен следовать после второго, 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 Latin 1. В главе 11 разъясняется, как сортировать числа по их числовым значениям.

***Пример***

В следующем примере массив клипов сортируется в соответствии с расположением на экране по горизонтали:

```
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 одинаковое x-положение
  }
}

clips.sort(compareXposition);
```

***См. также***

*Array.reverse( )*; раздел «Метод sort()» в главе 11.

---

**Array.splice( ) метод** — удалить элементы из массива и/или добавить элементы

***Доступность*** Flash 5

***Краткий обзор***

```
array.splice(startIndex)
array.splice(startIndex, deleteCount)
array.splice(startIndex, deleteCount, value1,...valuen)
```

***Аргументы***

*startIndex* — Целое число с отсчетом от нуля, указывающее место начала удаления элементов и необязательной вставки новых. 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 с *startIndex* и до конца массива.

*value1,...valuen* Необязательный список из одного или более значений, добавляемых в *array* начиная с индекса *startIndex* после удаления заданных элементов.

***Возвращаемое значение***

Новый массив, содержащий *удаленные* элементы (исходный массив модифицируется отдельно, отражая запрашиваемые изменения).

***Описание***

Метод *splice( )* удаляет элементы 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 смещает элементы вверх или вниз, обеспечивая непрерывность расположения 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 [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** преобразовать массив в строку значений-элементов, разделенных запятыми

***Доступность*** Flash 5

***Краткий обзор*** *array*.toString()

***Возвращаемое значение***

Список с разделителем-запятой элементов *array*, преобразованных в строки.

***Описание***

Метод *toString( )* создает строковое представление массива *array*. Строка, возвращаемая *toString( )*, есть список элементов массива, преобразованных в строки и разделенных запятыми (то же, что возвращает метод *join( )* при вызове без параметров). Метод массива *toString( )* автоматически вызывается, если массив используется в строковом контексте. Поэтому необходимость в выполнении *toString( )* вручную возникает редко. Обычно, если требуется точное строковое представление массива, используется 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** — добавить один или более элементов в начало массива

***Доступность*** Flash 5

***Краткий обзор*** `array.unshift(value1, value2, ...valuen)`

***Аргументы***

*value1,...valuen* Список из одного или более значений элементов, которые нужно 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 указанном порядке. Чтобы добавить элементы в конец массива, используйте *push( )*.

***Пример***

```
myList = new Array (5, 6);
myList.unshift(4);         // myList становится [4, 5, 6]
myList.unshift(7, 1);      // myList становится [7, 1, 4, 5, 6]
```

***См. также***

*Array.push( )*, *Array.shift( )*; раздел «Добавление элементов в массив» в главе 11.

---

**Boolean( ) глобальная функция** — преобразовать значение к типу данных *boolean*

***Доступность*** Flash 5

***Краткий обзор*** `Boolean(value)`

***Аргументы***

*value* Выражение со значением, которое 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. Обычно необходимости в использовании функции *Boolean( )* нет: ActionScript при надобности автоматически преобразует значения в булев тип.

***Особенности применения***

Не путайте глобальную функцию *Boolean( )* с конструктором класса *Boolean*. Функция *Boolean( )* преобразует свой аргумент в булев тип данных, в то время как конструктор класса *Boolean* создает новый булев объект. Обратите внимание, что в ECMA-262 все непустые строки преобразуются в `true`. Во Flash 5 `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***

The *Boolean* class; раздел «Преобразование типов данных» в главе 3 «Данные и типы данных».

---

**Boolean класс** класс-оболочка для элементарных булевых данных

***Доступность*** Flash 5

***Конструктор*** `new Boolean(`*value*`)`

***Аргументы***

*value* Выражение, которое должно быть вычислено и, при необходимости, преобразовано в булево значение, после чего помещено в оболочку объекта *Boolean*.

***Методы***

*toString( )* Преобразует значение объекта *Boolean* в строку.

*valueOf( )* Извлекает элементарное значение объекта *Boolean*.

***Описание***

Класс *Boolean* создает объект *Boolean*, содержащий элементарное булево значение в виде недоступного внутреннего свойства. Объекты *Boolean* используются только для 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, давая ему некоторые напоминающие объект методы. Сравните класс *Boolean* с классами *String* и *Number*, которые то-

же служат просто оболочками элементарных значений *string* и *number*, но приносят больше пользы.

По большей части объекты *Boolean* используются внутренним образом. Они автоматически создаются интерпретатором при вызове метода с элементарным булевым значением и автоматически уничтожаются после каждого использования. Объекты *Boolean* можно создавать самостоятельно с помощью конструктора *Boolean*, но в этом редко возникает необходимость.

***Особенности применения***

Обратите внимание, что на практике гораздо чаще в качестве средства преобразования типа данных используется глобальная функция *Boolean( )*, а не класс *Boolean*.

***См. также***

*Boolean( )* глобальная функция; раздел «Булев тип данных» в главе 4.

---

**Boolean.toString( ) метод** значение объекта Boolean, преобразованное в строку

***Доступность*** Flash 5

***Краткий обзор*** *booleanObject*.toString()

***Возвращаемое значение***

Строка «true», если элементарным значением *booleanObject* является `true`; «false», если элементарным значением *booleanObject* является `false`. Значение булева объекта указывается при создании объекта, после чего хранится внутренним образом. Хотя внутреннее значение булева объекта недоступно, 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** элементарное значение объекта Boolean

***Доступность*** Flash 5

***Краткий обзор*** *booleanObject*.valueOf()

***Возвращаемое значение***

Булево значение `true`, если элементарным значением *booleanObject* является `true`; `false`, если элементарным значением *booleanObject* является `false`.

Значение объекта Boolean указывается при создании объекта, после чего хранится внутренним образом.

***Описание***

Метод *valueOf( )* возвращает элементарное булево данное, связанное с объектом Boolean. Хотя внутреннее значение объекта Boolean недоступно, с помощью *valueOf( )* можно преобразовать его в эквивалент элементарного типа.

***Пример***

```
x = new Boolean(0);
trace(x.valueOf());  // Выводит: false
```

***См. также***

*Object.valueOf( )*.

---

**Call( ) глобальная функция** выполнить сценарий удаленного кадра

***Доступность*** Flash 4; устарела во Flash 5

***Краткий обзор***

```
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*. Например, следующий код выполняет сценарий кадра 20 текущей временной диаграммы:

```
call(20);
```

Во Flash 4 *call( )* использовалась для создания грубого подобия многократно используемой подпрограммы (не способной принимать параметры и возвращать значение). Во Flash 5 предпочтительнее использовать предложение *function*.

Обратите внимание, что во Flash 5 при удаленном вызове сценария через *call( )* переменные, объявленные посредством ключевого слова *var*, считаются локальными для этого сеанса выполнения и прекращают свое существование по завершении сценария. Чтобы создать нелокальные переменные в удаленно выполняемом сценарии, опустите ключевое слово *var*:

```
var x = 10;  // Локальная переменная; умирает после выполнения сценария
x = 10;     // Переменная временной диаграммы; сохраняется по завершении
```

Для вызова *call( )* с кадрами вне текущей временной диаграммы воспользуйтесь функцией *tellTarget( )*. В следующем примере выполняется сценарий в кадре 10 клипа `box`:

```
tellTarget ("box") {
  call(10);
}
```

***См. также***

главу 9 «Функции»; приложение C «Обратная совместимость».

---

**Color класс** управление значениями цветов клипа

***Доступность*** Flash 5

***Конструктор*** `new Color(target)`

***Аргументы***

*target* Строка или ссылка, задающая путь к клипу или уровню документа, цвет которых будет управляться новым объектом (ссылки преобразуются в пути, если используются в строковом контексте).

***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 смещениям 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*. Допустим, например, что есть экземпляр клипа с именем `ball`, который нужно 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
```

Этот пример показывает, как управлять цветом в простых приложениях. Для осуществления более сложных идей 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 и Alpha в некотором цвете описывается в виде чис-

ла 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 более темным цветам. Если значения всех трех RGB-составляющих равны, результатом будет оттенок серого цвета. Если все они равны 0, получается черный цвет. Если все они равны 255, получается белый цвет. Чем больше значение Alpha, тем менее прозрачен результирующий цвет. (Цвет со значением Alpha 0 полностью прозрачен, а со значением Alpha 255 – полностью непрозрачен.)

Например, чистый красный цвет описывается значениями:

```
Red: 255, Green: 0, Blue: 0, Alpha: 255
```

в то время как частично прозрачный красный может иметь значения:

```
Red: 255, Green: 0, Blue: 0, Alpha: 130
```

В данном изложении в разговоре о цветах принята так называемая нотация *тройки RGB* (*RGB triplet* – *Red*, *Green*, *Blue*). Хотя ActionScript не поддерживает десятичные тройки RGB, такие как (255, 0, 255), он поддерживает эквивалентный шестнадцатеричный формат 0x*RRGGBB*, где *RR*, *GG* и *BB* являются двузначными шестнадцатеричными числами, представляющими красный, зеленый и голубой цвета. В последующем изложении только для удобства принято также обозначение *четверки RGBA* (*Red*, *Green*, *Blue*, *Alpha*).

Изначальные значения Red, Green, Blue и Alpha для каждого цвета в клипе устанавливаются в редакторе Flash с помощью панели Mixer. (В панели Mixer значение Alpha представлено как процент, а не число от 0 до 255.) Для изменения всех цветов клипа с помощью ActionScript осуществляются общие настройки (называемые *трансформациями*) составляющих цвета клипа Red, Green, Blue и Alpha.

Есть два способа задать трансформацию для каждой составляющей цвета:

- Можно задать *процентное отношение* первоначального значения составляющей числом от –100 до 100. Скажем, можно потребовать, чтобы красный цвет, используемый в клипе, равнялся 80% первоначального значения.
- Можно указать величину *смещения* первоначального значения составляющей. Смещение является числом в диапазоне от –255 до 255. Например, можно сказать: «добавить 20 ко всем значениям голубого в данном клипе», или, с помощью отрицательного числа, «вычесть 30 из всех значений голубого в данном клипе».

Окончательное значение цвета в трансформированном клипе определяется, исходя из исходных, установленных на этапе создания документа, значений составляющих цвета и процентных отношений и смещений, установленных с помощью объекта *Color*, например:

```
R = originalRedValue   * (redTransformPercentage/100)   + redTransformOffset
G = originalGreenValue * (greenTransformPercentage/100) + greenTransformOffset
```

```
B = originalBlueValue  * (blueTransformPercentage/100) + lueTransformOffset
A = originalAlphaValue * (alphaTransformPercentage/100)+ alphaTransformOffset
```

Если никаких трансформаций с помощью ActionScript не проводилось, то начальное *процентное отношение* для каждой составляющей по умолчанию принимается равным 100, а первоначальное *смещение* – 0.

Рассмотрим, как действуют преобразования цвета, на практическом примере. 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 отношение Green равным 50, процентное отношение 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 сокращается на 50%
B ==   0 * (100/100) + 100 == 100    // Голубой смещается на 100
A == 255 *  (80/100) +   0 == 204    // Прозрачность 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 сокращается на 50%
B ==   0 * (100/100) + 100 == 100    // Голубой смещается на 100
A == 255 *  (80/100) +   0 == 204    // Прозрачность уменьшается на 80%
```

Окончательно в результате трансформации зеленый круг имеет значение цвета (R:0, G:127.5, B:100, A:204).

Чтобы применить гипотетическую трансформацию цвета к реальному клипу, используется рассматриваемый объект *Color*. Для задания универсальных значений смещения и процентного отношения цветов применяются методы *setRGB( )* или *setTransform( )* (примеры кода находятся в статьях по этим методам). Обратно, чтобы узнать текущую трансформацию цветов клипа, используются методы *getRGB( )* и *getTransform( )*. С помощью методов цвета *Color* можно создавать эффекты анимации – например, постепенное проявление, постепенное исчезновение и окрашивание. Кроме того, так как изменять оттенки каждого экземпляра клипа можно независимо, 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 ниже под заголовком «Пример».

### *Особенности применения*

Вот некоторые особенности применения объектов *Color*:

- Изменение цвета клипа с помощью объекта *Color* прервет трансформацию и передаст клип под управление ActionScript, что может прекратить действие некоторых режимов клипа, заданных на этапе создания документа.
- Установка значения свойства `_alpha` клипа воздействует на процентное отношение Alpha, отражаемое в свойстве `aa` объекта, возвращаемом *getTransform( )*.
- Трансформации цвета не действуют на цвет фона фильма или клипа. Они воздействуют только на твердые тела, помещаемые на рабочий стол.
- К клипам можно применять вручную трансформации цветов в редакторе с помощью панели Effect (меню Window → Panels → Effect). Все такие трансформации отражаются в свойствах объекта, возвращаемых *getTransform( )*. Панель Effect дает отличный способ 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 рабочий стол
  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));
}
```

Присваивая смещениям цветов Red, Green и Blue одинаковые значения, можно добиться 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 клип осветляется и затемняется в зависимости от положения курсора мыши:

```
onClipEvent (load) {
  var brightness = new Color(this);
  var brightnessTransform = new Object();
  var stageWidth = 550;
```

```
}

onClipEvent (mouseMove) {
  brightnessAmount = -255 + (_root._xmouse / stageWidth) * 510;
  brightnessTransform.rb = brightnessAmount;
  brightnessTransform.bb = brightnessAmount;
  brightnessTransform.gb = brightnessAmount;
  brightness.setTransform(brightnessTransform);
  updateAfterEvent();
}
```

---

**Color.getRGB( ) метод** получить текущие значения смещений для Red, Green и Blue

***Доступность*** Flash 5

***Краткий обзор*** *colorObj*.getRGB()

***Возвращаемое значение***

Число, выражающее текущие смещения RGB для *colorObj*.

***Описание***

Метод *getRGB( )* возвращает число в диапазоне от –16777215 до 16777215, представляющее текущие смещения цветов клипа. Для получения процентных отношений цветов нужно использовать *getTransform( )*. Поскольку значения смещений цветов обычно лежат в диапазоне от 0 до 255, удобно работать со значением, возвращаемым *getRGB( )*, в шестнадцатеричном виде, где смещение каждого цвета представляется двузначным шестнадцатеричным числом. Для расшифровки числа, возвращаемого *getRGB( )*, нужно обрабатывать его как шестнадцатеричное число вида 0x*RRGGBB*, где *RR* является смещением красного, *GG* смещением зеленого и *BB* смещением голубого. Например, если *getRGB( )* возвращает число 10092339, преобразуем его в шестнадцатеричное число 0x99FF33 и получим смещения цветов R:153, G:255, B:51. Если же *getRGB( )* возвращает число 255, преобразуем его в шестнадцатеричное число 0x0000FF и получим 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 разработчиков сети, поскольку они часто используются в тегах HTML. Вот, например, как шестнадцатеричное число используется для задания цвета фона страницы (полные значения красного и голубого вместе дают розовый):

```
<BODY BGCOLOR="#FF00FF">
```

Шестнадцатеричный формат цвета, используемый в тегах HTML, является, в сущности, тем же форматом, который используется *getRGB( )* и *setRGB( )*. Однако необязательно использовать для интерпретации значения, возвращаемого *getRGB( )*, шестнадцатеричные числа: можно извлекать отдельные значения смещений красного, зеленого и голубого 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 голубого и присвоить blue
```

Когда значения смещений выделены в отдельные переменные, можно работать с ними отдельно как с десятичными числами. Однако если нужно применить изменения значений смещений к объекту *Color*, нужно снова преобразовать смещения в одно число, как показано в описании метода *setRGB( )*.

***Особенности применения***

Методами *getRGB( )* и *setRGB( )* удобно пользоваться при непосредственном присвоении клипу новых цветов без учета исходных цветов клипа. Методы *getTransform( )* и *setTransform( )* лучше подходят для изменения составляющих RGB трансформаций цветов клипа относительно его исходных цветов.

Значения смещений легче всего прочесть с помощью метода *getTransform( )*, который возвращает значения каждой составляющей в отдельности, а не общей суммой, как это делает *getRGB( )*. Это особенно относится к установке отрицательных смещений с помощью *setTransform( )* благодаря способу представления отрицательных чисел в двоичном виде.

***Пример***

```
// Создать новый объект Color для клипа с именем box
boxColor = new Color("box");
// Установить новое смещение RGB для "box"
boxColor.setRGB(0x333366);
// Проверить смещение RGB для "box"
trace(boxColor.getRGB());       // Выводит: 3355494
```

***См. также***

*Color.getTransform( )*, *Color.setRGB( )*.

---

**Color.getTransform( ) метод** извлечь как смещения, так и процентные отношения для составляющих клипа Red, Green, Blue и Alpha

***Доступность*** Flash 5

***Краткий обзор*** *colorObj*.getTransform()

***Возвращаемое значение***

Объект, в свойствах которого содержатся значения трансформаций цвета целевого клипа *colorObj*.

### *Описание*

Метод *getTransform( )* возвращает объект со свойствами, сообщающими, какие трансформации применены в данный момент к целевому клипу объекта *Color*. Названия свойств и их значения для возвращаемого объекта описаны в табл. 4.

*Таблица 4. Свойства объекта, возвращаемого getTransform( )*

| Имя свойства | Значение свойства | Описание свойства |
|---|---|---|
| ra | –100 to 100 | Процентное отношение трансформации Red |
| rb | –255 to 255 | Величина смещения Red |
| ga | –100 to 100 | Процентное отношение трансформации Green |
| gb | –255 to 255 | Величина смещения Green |
| ba | –100 to 100 | Процентное отношение трансформации Blue |
| bb | –255 to 255 | Величина смещения Blue |
| aa | –100 to 100 | Процентное отношение трансформации Alpha |
| ab | –255 to 255 | Величина смещения Alpha |

### *Особенности применения*

Обратите внимание на то, что и процентное отношение, и смещение могут быть отрицательными, однако это лишь один из участников расчета составляющих RGB-цветов, которые всегда находятся в диапазоне 0...255. Значения, выходящие за диапазон, обрезаются до допустимых величин. Описание расчета, производимого для определения итоговых составляющих RGB и Alpha, см. в описании класса *Color*.

### *Пример*

Метод *getTransform( )* в сочетании с *setTransform( )* может быть использован для того, чтобы отдельно изменять составляющие преобразований Red, Green, Blue и Alpha. 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;      // Установить смещение Red равным 200
boxTransform.aa = 60;       // Установить процент Alpha равным 60

// Применить новую трансформацию к box через boxColor
boxColor.setTransform(boxTransform);
```

***См. также***

*Color.setTransform( ).*

---

**Color.setRGB( ) метод** — присваивает новые значения смещений для Red, Green и Blue

***Доступность*** Flash 5

***Краткий обзор*** `colorObj.setRGB(offset);`

***Аргументы***

*offset* — Число в диапазоне от 0 до 16777215 (0xFFFFFF), представляющее новые смещения RGB для целевого клипа `colorObj`. Может быть десятичным или шестнадцатеричным целым.

Числа, не входящие в допустимый диапазон, преобразуются в числа из допустимого диапазона (при помощи дополнения до двух в двоичной записи). Поэтому *setRGB( )* нельзя использовать для установки отрицательных смещений (а *setTransform( )* можно).

***Описание***

Метод *setRGB( )* присваивает новые значения трансформации для RGB-составляющих клипа. Новое значение `offset` легче всего задать как шестизначное шестнадцатеричное число вида 0x*RRGGBB*, где `RR`, `GG` и `BB` являются двузначными числами в диапазоне от 00 до FF, представляющими составляющие Red, Green и Blue. Например, RGB-тройка (51, 51, 102) эквивалентна шестнадцатеричному значению:

```
0x333366
```

Следовательно, чтобы присвоить RGB-смещение серого клипу с именем `menu`, можно выполнить код:

```
var menuColor = new Color("menu");
menuColor.setRGB(0x999999);
```

Сетевым разработчикам, привыкшим к 6-значным шестнадцатеричным значениям цветов в HTML, будет легко использовать *setRGB( )* и предыдущий шестнадцатеричный формат. Учебник для начинающих по десятичным, восьмеричным и шестнадцатеричным числам можно найти по адресу: *http://www.moock.org/asdg/technotes*.

Обратите внимание, что, помимо смещений, *setRGB( )* также автоматически задает процентные отношения Red, Green и Blue трансформации цветов клипа в 0, т. е. изменения цвета, осуществляемые *setRGB( )*, действуют как непосредственное назначение цветов (а не просто настройка первоначальных цветов клипа). Для настройки цвета клипа относительно его исходных цветов нужно использовать метод *setTransform( )*.

***Пример***

Вот удобный способ генерации числа для использования в методе *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:

```
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 (201, 160, 21)
boxColor.setRGB(combineRGB(201, 160, 21));
```

Дополнительные сведения о поразрядных операциях см. в главе 15 «Более сложные темы»

***См. также***

*Color.getRGB( )*, *Color.setTransform( )*.

---

**Color.setTransform( ) метод** — присвоить новые значения смещений и/или процентных отношений для Red, Green, Blue и Alpha

***Доступность*** Flash 5

***Краткий обзор*** *colorObj*.setTransform(*transformObject*)

***Аргументы***

*transformObject* Объект, свойства которого содержат новые значения трансформации цвета для целевого клипа *colorObj*.

***Описание***

Метод *setTransform( )* предоставляет точный контроль над процентными отношениями и смещением составляющих Red, Green, Blue и Alpha цвета клипа. Для использования *setTransform( )* нужно сначала создать объект с рядом предопределенных свойств. Трансформация, которую нужно применить к объекту *Color*, описывается через эти свойства, перечисленные в табл. 4 для метода *getTransform( )* (разница в том, что *setTransform( )* задает эти значения, а не читает их).

Когда объект со свойствами, описанными в табл. 4, создан, он передается методу *setTransform( )*. Значения свойств *transformObject* становятся новыми значениями смещения и процентного отношения трансформации *colorObj* и, следовательно, целевого клипа *colorObj*. Окончательные значения цветов клипа определяются после этого в соответствии с расчетом, приведенным в описании класса *Color*.

Чтобы получить текущие смещения и процентные отношения некоторого объекта *Color*, нужно вызвать метод *getTransform( )*.

***Пример***

```
// Создать новый объект Color для клипа box
```

```
boxColor = new Color("box");

// Создать новый анонимный объект и сохранить его в boxTransform
boxTransform = new Object();

// Присвоить требуемые значения свойствам boxTransform, настроив
// в результате значения 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

// Теперь, когда объект трансформации готов,
// передать 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, получив в результате
// правильно построенный 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** строка, представляющая текущие дату и время

***Доступность*** Flash 5

***Краткий обзор*** `Date()`

***Возвращаемое значение***

Строка, содержащая текущие дату и время.

***Описание***

Функция *Date( )* возвращает читаемую строку, описывающую текущие дату и время для местного часового пояса. Строка содержит также разни-

цу 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 строки. Ее удобно читать, но не очень удобно использовать в программе, когда требуется осуществлять действия над датой и временем. Для таких целей следует предпочесть объекты класса *Date*, которые позволяют легко обращаться отдельно к году, месяцу, числу и времени.

***Пример***

Чтобы с минимальными усилиями поместить текущие дату и время в текстовое поле, можно воспользоваться функцией *Date( )* следующим образом:

```
myTextField = Date();

// Присваивает myTextField строку следующего формата:
// "Mon Aug 28 16:23:09 GMT-0400 2000"
```

***См. также***

*Date* класс, *Date.UTC( )*.

---

**Date класс** текущее время и структурированные сведения о дате

***Доступность*** Flash 5

***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 (Универсального времени, типа GMT). Положительное, если новая дата после этой даты, и отрицательное, если до нее. Всякая необходимая настройка местного часового пояса происходит после определения даты по UTC. Например, задав аргумент *milliseconds* равным 1000 при восточном стандартном 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 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 2011). Для задания года после 2000 используйте четыре цифры (например, указывайте 2010, а не 10). Год из трех цифр рассматривается как год до 1000 г. н.э. Учтите, что если *year* отрицателен или меньше 800, расчет не вполне надежен. Для

задания дат до 1000 года н.э. надежнее использовать формат конструктора *milliseconds*.

*month* — Целое число, задающее месяц, от 0 (январь) до 11 (декабрь), а не от 1 до 12. Обязательно 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, используется 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, используется значение по умолчанию 0. Обозначения AM и PM (до и после полудня) не поддерживаются. 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, используется значение по умолчанию 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, используется значение по умолчанию 0. Секунды, выходящие за этот диапазон, переносятся 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, используется значение по умолчанию 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( )* — Получить число миллисекунд между 1 января 1970 года и переданной 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 суток от 0 до 23. |
| *getMilliseconds( )* | Получить миллисекунды. |
| *getMinutes( )* | Получить минуты. |
| *getMonth( )* | Получить месяц года как число от 0 (январь) до 11 (декабрь). |
| *getSeconds( )* | Получить секунды. |
| *getTime( )* | Получить дату во внутреннем формате (т. е. 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. |
| *getUTCDay( )* | Получить день недели по универсальному времени. |
| *getUTCFullYear( )* | Получить год в формате четырех цифр по универсальному времени. |
| *getUTCHours( )* | Получить час суток по универсальному времени. |
| *getUTCMilliseconds( )* | Получить миллисекунды по универсальному времени. |
| *getUTCMinutes( )* | Получить минуты по универсальному времени. |
| *getUTCMonth( )* | Получить месяц года по универсальному времени. |
| *getUTCSeconds( )* | Получить секунды по универсальному времени. |
| *getYear( )* | Получить год относительно 1900. |
| *setDate( )* | Установить число месяца. |
| *setFullYear( )* | Установить год в формате четырех цифр. |
| *setHours( )* | Установить час суток. |
| *setMilliseconds( )* | Установить миллисекунды. |
| *setMinutes( )* | Установить минуты. |
| *setMonth( )* | Установить месяц года. |
| *setSeconds( )* | Установить секунды. |
| *setTime( )* | Установить дату во внутреннем формате (т. е. как число миллисекунд между датой и 1 января 1970 года). |
| *setUTCDate( )* | Установить число месяца по универсальному времени. |
| *setUTCFullYear( )* | Установить год в формате четырех цифр по универсальному времени. |
| *setUTCHours( )* | Установить час суток по универсальному времени. |

*setUTCMilliseconds( )*
Установить миллисекунды по универсальному времени.

*setUTCMinutes( )* Установить минуты по универсальному времени.

*setUTCMonth( )* Установить месяц года по универсальному времени.

*setUTCSeconds( )* Установить секунды по универсальному времени.

*setYear( )* Установить год в формате 4 цифр или, для 20 столетия, 2 цифр.

*toString( )* Удобная для чтения строка, представляющая дату.

*valueOf( )* Число миллисекунд между датой и полуночью 1 января 1970 года по универсальному времени.

***Описание***

Объекты класса *Date* используются как механизм для определения текущих времени и даты и как средство хранения 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 число миллисекунд положительное, дата следует за полуночью 1 января 1970 года; если число миллисекунд отрицательное, дата предшествует полуночи 1 января 1970 года. Например, если значение даты 10000, оно соответствует времени 12:00:10, 1 января 1970 года. Если значение даты –10000, 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 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*:

- Можно вызвать конструктор *Date( )* без аргументов. В результате новый объект *Date* устанавливается в текущее время.
- Можно вызвать конструктор *Date( )* с одним числовым аргументом: числом миллисекунд между полуночью 1 января 1970 года и создаваемой датой.
- Можно 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 хранятся внутренне как одно число, у объектов *Date* нет свойств, которые можно читать или задавать. Вместо этого есть методы, используемые для доступа к различным компонентам даты в понятном че-

ловеку виде (т. е. в более удобных единицах). Например, для определения месяца конкретного объекта *Date* используется:

```
myMonth = myDate.getMonth();
```

Нельзя писать:

```
myMonth = myDate.month;  // Нет такого свойства! Используются методы.
```

Многие методы *Date* содержат буквы «UTC», служащие сокращением для Coordinated Universal Time (универсального времени). В большинстве случаев UTC служит просто синонимом более привычного Гринвичского времени (Greenwich Mean Time, или GMT) – времени, измеренного на гринвичском меридиане. UTC просто является новым стандартом, менее неоднозначным, чем GMT, которое в разные времена имело 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.

***Особенности применения***

Все даты и моменты времени определяются согласно настройкам операционной системы, на которой выполняется фильм Flash (а не веб-сервера) и учитывают региональные смещения. Поэтому точность времени и даты определяется системой пользователя.

Обратите внимание, что конструктор *Date( )* может использоваться и как глобальная функция для создания строки, содержащей текущее время в таком же формате, который используется `myDate.toString()`.

Во Flash 5 ActionScript нельзя преобразовать строку в объект *Date*, как это можно сделать в JavaScript.

***Пример***

Даты можно складывать и вычитать для получения суммарного времени или истекшего времени. Допустим, наш друг Грэм решил отправиться в путешествие чуть меньше, чем на год. Пока он отсутствует, нам нужно 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 число
// миллисекунд в сутках. В результате узнаем, сколько дней мы ждем.
```

```
var numDaysGone = Math.floor(gone/86400000);

// Таким же способом узнаем, сколько дней осталось ждать
var left = returns.getTime() - now.getTime();
var numDaysLeft = Math.floor(left/86400000);

// Вывести полученные количества дней в текстовых полях
goneOutput = numDaysGone;
leftOutput = numDaysLeft;
```

При добавлении или вычитании суток из имеющейся даты обычно для удобства количество миллисекунд в сутках (86400000) 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, нужно вручную преобразовать значения, возвращаемые *getDate( )*, *getDay( )*, *getHours( )*, и так далее 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 использования класса *Date* для создания часов в аналоговом виде есть в разделе «Создание часов с помощью клипов» в главе 13 «Клипы». Заметим, что программирование поведения, зависящего от времени, например, 10-секундной паузы, часто легче осуществить с помощью глобальной функции *getTimer( )*.

***См. также***

*Date( )*, *Date.UTC( )*, *getTimer( )*.

---

**Date.getDate( ) метод** число месяца

***Доступность*** Flash 5

***Краткий обзор*** *date*.getDate()

***Возвращаемое значение***

Целое число от 1 до 31, представляющее число месяца для *date*.

---

**Date.getDay( ) метод** день недели

***Доступность*** Flash 5

***Краткий обзор*** *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 так далее):

```
now = new Date();
today = now.getDay();
days = ["sun", "mon", "tue", "wed", "thu", "fri", "sat"];
welcomeHeader.loadMovie(days[today] + ".swf");
```

---

**Date.getFullYear( ) метод** год в формате четырех цифр

***Доступность*** Flash 5

***Краткий обзор*** *date*.getFullYear()

***Возвращаемое значение***

Четырехзначное целое число, представляющее год для *date*, например, 1999 или 2000.

**Date.getHours( ) метод** час суток

***Доступность*** Flash 5

***Краткий обзор*** *date*.getHours()

***Возвращаемое значение***

Целое число от 0 (полночь) до 23 (11 часов вечера), представляющее час суток *date*. Обозначения A.M. и P.M. не поддерживаются, но могут быть созданы вручную, как показано в примерах к классу *Date*.

---

**Date.getMilliseconds( ) метод** миллисекунды даты

***Доступность*** Flash 5

***Краткий обзор*** *date*.getMilliseconds()

***Возвращаемое значение***

Целое число от 0 до 999, представляющее миллисекунды *date*. Учтите, что это не миллисекунды после 1970 года (см. *getTime( )*), а дробная часть секунд, указанных в данном объекте *Date*.

***См. также***

*Date.getTime( )*.

---

**Date.getMinutes( ) метод** минуты даты

***Доступность*** Flash 5

***Краткий обзор*** *date*.getMinutes()

***Возвращаемое значение***

Целое число от 0 до 59, представляющее минуты часа в *date*.

---

**Date.getMonth( ) метод** месяц года

***Доступность*** Flash 5

***Краткий обзор*** *date*.getMonth()

***Возвращаемое значение***

Целое число от 0 (январь) до 11 (декабрь), а не от 1 до 12, представляющее месяц года в *date*.

***Особенности применения***

Помните, что 1 – это не январь! Значения, возвращаемые *getMonth( )*, начинаются с 0, а не 1.

***Пример***

Следующий код преобразует число, возвращаемое *getMonth( )*, в читаемое сокращенное название:

```
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** секунды даты

***Доступность*** Flash 5

***Краткий обзор*** *date*.getSeconds()

***Возвращаемое значение***

Целое число от 0 до 59, представляющее количество секунд, оставшихся от целых минут в *date*.

---

**Date.getTime( ) метод** получить количество миллисекунд между 1 января 1970 года и временем в объекте даты

***Доступность*** Flash 5

***Краткий обзор*** *date*.getTime()

***Возвращаемое значение***

Целое число, выражающее количество миллисекунд между временем *date* и полуночью 1 января 1970 года. Положительное для дат после 1 января 1970 года, и отрицательное для дат до 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 для создания других дат или сравнения времени, истекшего 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();
```

С помощью следующего кода можно определить промежуток времени, истекший между 10-м и 20-м кадрами фильма, в миллисекундах:

```
elapsedTime = time2.getTime() - time1.getTime();
```

Заметим, что глобальная функция *getTimer( )* тоже дает доступ к величине истекшего времени в фильме.

***См. также***

*Date.setTime( )*, *getTimer( )*.

---

**Date.getTimezoneOffset( ) метод** количество минут между местным временем и UTC (или, что то же, GMT)

***Доступность*** Flash 5

***Краткий обзор*** *date*.getTimezoneOffset()

***Возвращаемое значение***

Целое число, представляющее текущее количество *минут* между местным часовым поясом и фактическим временем UTC (или Гринвичского меридиана). Положительное, если местное время отстает от UTC; отрицательное, если местное время впереди UTC. Содержит поправки для летнего времени в зависимости от дня в году.

***Пример***

При вызове по восточному летнему 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) и UTC.

---

**Date.getUTCDate( ) метод** число месяца (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *date*.getUTCDate()

***Возвращаемое значение***

Целое число от 1 до 31, представляющее число месяца для *date*, где *date* рассматривается как время UTC.

---

**Date.getUTCDay( ) метод** день недели (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *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** год в четырехзначном формате (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *date*.getUTCFullYear()

***Возвращаемое значение***

Четырехзначное целое число, представляющее год для *date*, где *date* рассматривается как время UTC. Например, 1999 или 2000.

---

**Date.getUTCHours( ) метод** час суток (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *date*.getUTCHours()

***Возвращаемое значение***

Целое число от 0 (полночь) до 23 (11 часов вечера), представляющее час суток для *date*, где *date* рассматривается как время UTC.

---

**Date.getUTCMilliseconds( ) метод** миллисекунды даты (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *date*.getUTCMilliseconds()

***Возвращаемое значение***

Целое число от 0 до 999, представляющее миллисекунды *date*, где *date* рассматривается как время UTC.

---

**Date.getUTCMinutes( ) метод** минуты даты (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *date*.getUTCMinutes()

***Возвращаемое значение***

Целое число от 0 до 59, представляющее минуты часа *date*, где *date* рассматривается как время UTC.

---

**Date.getUTCMonth( ) метод** месяц года (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *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.

***Особенности применения***

Помните, что 1 – это не январь! Значения, возвращаемые *getUTCMonth( )*, начинаются с 0, а не с 1.

---

**Date.getUTCSeconds( ) метод** секунды даты (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *date*.getUTCSeconds()

***Возвращаемое значение***

Целое число от 0 до 59, представляющее секунды минуты *date*, где *date* рассматривается как время UTC.

---

**Date.getYear( ) метод** год относительно 1900

***Доступность*** Flash 5

***Краткий обзор*** `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 20-го века.

---

**Date.setDate( ) метод** задать число месяца

***Доступность*** Flash 5

***Краткий обзор*** `date.setDate(day)`

***Аргументы***

*day* Целое число от 1 до 31, представляющее новое число месяца для `date`. Если задать число, большее количества дней в текущем месяце, месяц будет соответствующим образом увеличен. Например, если текущий месяц 8 (сентябрь), а для нового `day` задается 31, это воспринимается как 1 октября. `day` станет 1, а `month` станет 9 (октябрь).

***Возвращаемое значение***

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

***См. также***

*Date.getDate( )*.

---

**Date.setFullYear( ) метод** задать век и год в четырехзначном формате

***Доступность*** Flash 5

***Краткий обзор*** `date.setFullYear(year, month, day)`

***Аргументы***

*year* Четырехзначное целое число, 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.setYear( )*, *Date.getFullYear( )*.

**Date.setHours( ) метод** установить час суток

***Доступность*** Flash 5

***Краткий обзор*** `date.setHours(hour)`

***Аргументы***

*hour* Целое число от 0 (полночь) до 23 (11 часов 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** установить миллисекунды даты

***Доступность*** Flash 5

***Краткий обзор*** `date.setMilliseconds(ms)`

***Аргументы***

*ms* Целое число от 0 до 999, представляющее новое значение миллисекунд *date*, а не количество миллисекунд после 1970. Значения больше 999 и меньше 0 переводятся 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** установить минуты даты

***Доступность*** Flash 5

***Краткий обзор*** `date.setMinutes(minute)`

***Аргументы***

*minute* Целое число от 0 до 59, представляющее новое значение минут часа *date*. Значения больше 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** установить месяц года

***Доступность*** Flash 5

***Краткий обзор*** `date.setMonth(month)`

***Аргументы***

*month* Целое число от 0 (январь) до 11 (декабрь), а не от 1 до 12, представляющее новый месяц года *date*. Значения больше 11 или меньше 0 переносятся в год *date*.

***Возвращаемое значение***

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года.

***Особенности применения***

Помните, что 1 – это не январь! Нумерация месяцев начинается с 0, а не с 1.

***См. также***

*Date.getMonth( )*.

---

**Date.setSeconds( ) метод** установить секунды даты

***Доступность*** Flash 5

***Краткий обзор*** `date.setSeconds(second)`

***Аргументы***

*second* Целое число от 0 до 59, представляющее число секунд минуты *date*. Значения больше 59 и меньше 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** присвоить новое значение даты по числу миллисекунд между 1 января 1970 года и новой датой

***Доступность*** Flash 5

***Краткий обзор*** `Date.setTime(milliseconds)`

***Аргументы***

*milliseconds* Целое число, представляющее число миллисекунд между временем заданной даты 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 число 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***

Совместно используя *setTime( )* и *getTime( )*, можно настраивать имеющуюся дату, 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 число миллисекунд в часе и число миллисекунд в сутках:

```
oneDay = 86400000;
oneHour = 3600000;
now = new Date();
// Отнять одни сутки и три часа.
now.setTime(now.getTime() - oneDay - (3 * oneHour));
```

***См. также***

*Date.getTime( )*, *Date.setMilliseconds( )*, *Date.UTC( )*, *getTimer( )*.

---

**Date.setUTCDate( ) метод** установить число месяца (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** *date*.setUTCDate(*day*)

***Аргументы***

*day* Целое число от 1 to 31, представляющее новое число месяца для *date* по времени UTC. Если задать число месяца, большее количества дней в текущем месяце, месяц соответственно меняется. Например, если текущий месяц 8 (сентябрь), и в качестве нового *day* задать 31, получится 1 октября. *day* станет 1, а *month* станет 9 (октябрь).

***Возвращаемое значение***

Целое число, представляющее количество миллисекунд между новой датой и полуночью 1 января 1970 года по времени UTC.

**Date.setUTCFullYear( ) метод** установить век и год в формате четырехзначного числа (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** `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** установить час суток (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** `date.setUTCHours(hour)`

***Аргументы***

*hour* Целое число от 0 (полночь) до 23 (11 часов 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** установить миллисекунды для даты (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** `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** установить значение минут для даты (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** `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** установить месяц года (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** `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.

***Особенности применения***

Помните, что 1 – это не январь! Значения месяцев начинаются с 0, а не с 1.

---

**Date.setUTCSeconds( ) метод** присвоить значение секунд даты (время UTC)

***Доступность*** Flash 5

***Краткий обзор*** `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** установить год относительно 1900

***Доступность*** Flash 5

***Краткий обзор*** `date.setYear(year, month, day)`

***Аргументы***

*year* Обязательное целое число, задающее новый год *date*. Если передаются одна или две цифры, год считается относящимся к 20-му веку. Например, 1 задает год 1901, а 99 задает год 1999. Год из трех цифр считается относящимся к периоду до 1000 г. н.э. Четыре цифры используются для задания 2000 года и далее.

*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( )*, но год из одной или двух цифр интерпретируется относительно 1900, тогда как *setFullYear( )* рассматривает их относительно 0 н.э.

***См. также***

*Date.getYear( )*, *Date.setFullYear( )*.

---

**Date.toString( ) метод** пригодная для чтения строка, представляющая дату

***Доступность*** Flash 5

***Краткий обзор*** `date.toString()`

***Возвращаемое значение***

Строка, выводящая время и дату *date* в пригодном для чтения виде, включая разницу с UTC (GMT). Для создания специальных представлений даты используйте методы получения числа, часа, минут и так далее и преобразуйте 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( ) Class метод** получить количество миллисекунд между 1 января 1970 года и заданной датой UTC

***Доступность*** Flash 5

***Краткий обзор*** `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 см. 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 1970 года и полуночью 2000 года по UTC:

```
trace(Date.UTC(2000, 0) + " milliseconds passed between 1970 and 2000.");
// Выводит: "946684800000 milliseconds passed between 1970 and 2000."
```

Теперь используем 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 проверить час с помощью не использующего UTC метода *getHours( )*, получится местное время 19 часов:

```
trace(nowUTC.getHours());  // Выводит: 19
```

Но если получить час с помощью использующего UTC метода *getUTCHours( )*, это будет правильное время по UTC, 0 (полночь):

```
trace(nowUTC.getUTCHours());  // Выводит: 0
```

***См. также***

*Date( )*, *Date.setTime( )*, *Date* класс.

---

**Date.valueOf( ) метод** число миллисекунд между 1 января 1970 года и временем объекта *Date*

***Доступность*** Flash 5

***Краткий обзор*** *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( )*.

---

**duplicateMovieClip( ) глобальная функция** создать копию клипа

***Доступность*** Flash 4 и последующие

***Краткий обзор*** `duplicateMovieClip(target, newName, depth)`

***Аргументы***

*target* — Строка, указывающая путь к клипу, который нужно дублировать (называемому также *порождающим клипом* (*seed clip*)). Ссылаться на вложенные клипы можно с помощью синтаксиса точки, например, *duplicateMovieClip("_root.myClip","myClip2",0)*. Так как ссылка на клип преобразуется в путь при использовании в строковом контексте, *target* может также быть ссылкой на объект клипа, как в *duplicateMovieClip(myClip,"myClip2",0)*.

*newName* — Строка, которая станет именем экземпляра дублируемого клипа. Она должна удовлетворять правилам создания идентификаторов, приведенным в разделе «Идентификаторы» главы 14 «Лексическая структура».

*depth* — Целое число, показывающее уровень стека программно создаваемых клипов, на который должен быть помещен продублированный клип. Клипы на более низких уровнях визуально помещаются позади клипов на более высоких уровнях. Клип с самым высоким значением *depth* в стеке закрывает все клипы, лежащие ниже. Например, клип на *depth* −1 оказывается за клипом на *depth* 0, который в свою очередь оказывается за клипом на *depth* 1. Если назначаемое значение *depth* уже используется, прежний клип удаляется, а его место занимает новый клип. Дополнительные сведения есть в главе 13. Отрицательные значения уровней действуют, но официально не поддерживаются ActionScript – для гарантии совместимости с будущими версиями используйте уровни 0 и выше.

***Описание***

Функция *duplicateMovieClip( )* предоставляет один из способов создать новый клип во время воспроизведения фильма (другой способ дает *attachMovie( )*). *duplicateMovieClip( )* создает полную копию *target* и помещает ее в стек клипов *target* на уровень *depth*. Дубликат клипа начинает воспроизведение 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*. Доступ к глобальной функции *duplicateMovieClip( )* можно осуществить и как к методу клипа, но в таком формате не используется аргумент *target*.

***Пример***

```
// Копирует клип ball и дает копии имя ball2
duplicateMovieClip(ball, "ball2", 0);
// Перемещает созданный клип ball2, чтобы его можно было видеть
ball2._x += 100;
```

***См. также***

*MovieClip.duplicateMovieClip( )*, *removeMovieClip( )*; разделы «Создание экземпляров с помощью duplicateMovieClip()», «Добавление в стек клипов, созданных с помощью duplicateMovieClip» и раздел «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13.

---

**escape( ) глобальная функция** кодировать строку для безопасной передачи по сети

***Доступность*** Flash 5

***Краткий обзор*** `escape(string)`

***Аргументы***

*string* Строка (или выражение, возвращающее строку), которая должна быть закодирована.

***Возвращаемое значение***

Версия *string*, почти совпадающая с URL-кодировкой.

***Описание***

Функция *escape( )* создает на основе переданной ей строки новую закодированную строку. Вместо символов строки, отличных от цифр и базовых латинских букв от A до Z и от a до z без акцентов, в новую строку помещаются шестнадцатеричные escape-последовательности. Эти замещающие последовательности имеют формат `%xx`, где `xx` является шестнадцатеричным значением кодовой позиции символа в наборе символов Latin 1. Двухбайтовые символы кодировки Shift-JIS преобразуются в две шестнадцатеричные escape-последовательности вида `%xx%xx`.

Функция *escape( )* фактически осуществляет URL-кодировку строки, только пробелы преобразуются в `%20`, а не в `+`. *escape( )* иногда применяется при отправке информации из фильма Flash приложениям на сервере или записи cookies в броузере.

Для раскодирования закодированной строки используется глобальная функция *unescape( )*.

***Пример***

```
var phoneNumber = "(222) 515-1212"
escape(phoneNumber);  // дает %28222%29%20515%2D1212
```

***См. также***

*unescape( )*; приложение B «Набор символов Latin 1 и коды клавиш».

---

**eval( ) глобальная функция** преобразовать строку в идентификатор

***Доступность*** Flash 4 и выше

***Краткий обзор*** `eval(stringExpression)`

***Аргументы***

*stringExpression* Строка или выражение, возвращающее строку. Должна соответствовать имени некоторого идентификатора.

***Возвращаемые значения***

Значение переменной, представленной *stringExpression*, или ссылка на объект, клип или функцию, представленные *stringExpression*. Если *stringExpression* не представляет переменной или клипа, возвращается значение undefined.

***Описание***

Функция *eval( )* предоставляет способ создания динамической ссылки на идентификатор, представленный строкой текста. *eval( )* преобразует строку в переменную, клип, свойство объекта или другой идентификатор, а затем вычисляет это идентификатор. Вот пример использования значения, возвращаемого *eval( )*, для присвоения значения переменной:

```
name1 = "Kathy";
count = 1;
currentName = eval("name" + count); // Присваивает currentName "Kathy"
```

А вот пример управления клипом с динамически создаваемым именем star1:

```
eval("star" + count)._x += 100;      // Переместить star1 вправо
                                     // на 100 пикселов
```

Но можно использовать вызов *eval( )* и вместо идентификатора в левой части выражения присваивания:

```
eval("name" + count) = "Simone";     // Присваивает name1 значение "Simone"
```

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 в проигрыватель компилятор ActionScript, из-за чего размер проигрывателя слишком бы возрос.

***Особенности применения***

Во Flash 5 функция *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 в ссылку на объект клипа. В следующем примере предполагается, что есть ряд анимированных изображений человеческих лиц. Когда пользователь «опускает» клип food на одно из них, при помощи свойства _droptarget извлекается путь к соответствующему «лицу», а затем посредством *eval( )* создается ссылка на объект этого «лица». Наконец, «лицо» отправляется в кадр, отображающий рот, полный еды:

```
// Преобразовать строку _droptarget в ссылку
theFace = eval(food._droptarget);
// Выбрать нужный клип лица с помощью полученной ссылки
theFace.gotoAndStop("full");
```

***См. также***

раздел «Выполнение кода в строке с помощью eval()» в главе 4.

---

**_focusrect глобальное свойство** — выделенное состояние, используемое для кнопок, активизируемых через клавиатуру

***Доступность*** Flash 4 и выше

***Краткий обзор*** `_focusrect`

***Тип доступа*** Чтение/запись

***Описание***

Когда мышь находится над кнопкой во Flash, выводится 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 над этой кнопкой желтый прямоугольник, что не всегда желательно по эстетическим соображениям. Отключить этот желтый прямоугольник выделения 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. Когда `_focusrect` имеет значение `true` (по умолчанию), Flash выводит желтый прямоугольник.

***Особенности применения***

Хотя `_focusrect` используется в булевом смысле, в нем фактически хранится число, а не булево значение. Хотя необходимость в этом редко возникает, но если посмотреть на значение `_focusrect`, то оно равно либо 1 (что означает `true`), либо 0 (что означает `false`).

---

**fscommand( ) глобальная функция** — отправить сообщение автономному проигрывателю или приложению – хозяину проигрывателя

***Доступность*** Flash 3 и выше (усилена во Flash 5 включением `trapallkeys`)

***Краткий обзор*** `fscommand(`*`command, arguments`*`)`

***Аргументы***

*command* — Строка, передаваемая приложению-хозяину, часто имя функции JavaScript.

*arguments* — Строка, передаваемая приложению-хозяину, часто аргумент для функции, указанной в *command*.

***Описание***

С помощью функции *fscommand( )*, фильм Flash может связываться с автономным проигрывателем или приложением – хозяином проигрывателя:

средой, в которой выполняется Flash Player (веб-броузером или Macromedia Director). Функция *fscommand( )* обычно используется одним из трех способов:

Для отправки одной из немногочисленных встроенных команд автономному проигрывателю Flash

Для отправки команд языку сценариев веб-броузера, например, JavaScript или VBScript

Для связи с Lingo в фильме Macromedia Director

При использовании с автономным плеером *fscommand( )* принимает одну из встроенных пар *команда/ аргумент*, как показано в табл. 5.

*Таблица 5. Пары команда/аргумент в автономном проигрывателе*

| Команда | Аргумент | Описание |
|---|---|---|
| "allowscale" | "true" или "false" | В случае "false" предотвращает увеличение или сжатие содержимого фильма в зависимости от размера окна проигрывателя. "allowscale" часто используется в сочетании с "fullscreen", чтобы создать Projector, занимающий весь экран при сохранении исходного размера фильма |
| "exec" | "application_name" | Запускает внешнее приложение. Путь к приложению задается в строке *application_name*. Путь разрешается относительно фильма Flash, если только *application_name* не задано как абсолютный тип, например: "C:/WINDOWS/NOTEPAD.EXE". Обратите внимание, что в пути используется *прямая косая черта* ( / ), а не обратная ( \ ). |
| "fullscreen" | "true" или "false" | В случае "true" влечет максимизацию окна проигрывателя, которое занимает весь экран пользователя. |
| "quit" | н/а | Закрывает фильм и выходит из проектора Flash (автономный проигрыватель). |
| "showmenu" | "true" или "false" | В случае "false" подавляет вывод управляющих элементов в контекстном меню проигрывателя, оставляя только опцию About Macromedia Flash Player. Доступ к контекстному меню осуществляется через щелчок правой кнопкой в Windows и Ctrl-щелчок в Macintosh. |
| "trapallkeys" | "true" или "false" | В случае "true" все нажатия клавиш – даже клавиш сокращенного набора – посылаются фильму Flash. trapallkeys используется 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 режима Stop/ exit Full Screen mode, и так далее). Дополняет *fscommand* во Flash 5. |

При использовании в броузере функция *fscommand( )* в фильме влечет вызов специальной функции JavaScript (Netscape) или VBScript (Internet Explorer) на странице, содержащей фильм. Имя этой специальной функции имеет общий вид *movieID_DoFSCommand*, где *movieID* является именем, указанным в атрибуте фильма `OBJECT ID` (Internet Explorer) или атрибуте `EMBED NAME` (Netscape) в содержащем фильм документе HTML. При вызове *movieID_DoFSCommand( )* значения параметров *fscommand( )* `command` и `arguments` передаются в качестве аргументов функции *movieID_DoFSCommand( )*. Если на содержащей фильм странице не существует функции *movieID_DoFSCommand( )*, происходит отказ функции *fscommand( )* без каких-либо сообщений.

Обратите внимание, что для работы *fscommand( )* с Netscape атрибут `swLiveConnect` тега фильма `EMBED` должен быть задан как `"true"`. Например:

```
<EMBED
  NAME="testmovie"
  SRC="myMovie.swf"
  WIDTH="100%"
  HEIGHT="100%"
  swLiveConnect="true"
  PLUGINSPAGE="http://www.macromedia.com/go/flashplayer/"
</EMBED>
```

***Особенности применения***

Связь с броузером через *fscommand( ) не*возможна в следующих конфигурациях системы:

- Internet Explorer под Macintosh OS
- Любой броузер на Macintosh серии 68K
- Любой броузер под Windows 3.1
- Netscape 6

Заметим, что *fscommand( )* не всегда дает лучший способ связи с фильмом Director из Flash. Предпочтительным средством коммуникации с Director является функция *getURL( )* с протоколом `event:` или `lingo:`. За подробностями обратитесь к описанию функции *getURL( )* или к 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***

Чтобы выйти из автономного проектора, используйте такой код:

```
fscommand("quit");
```

Чтобы создать автономный проектор, выполняемый в полноэкранном режиме:

```
fscommand("fullscreen", "true");
```

Чтобы создать автономный проектор, выполняемый в полноэкранном режиме, но сохраняющий размер исходного фильма:

```
fscommand("fullscreen", "true");
fscommand("allowscale", "false");
```

Информацию о запуске фильма в полноэкранном окне броузера можно получить по адресу: *http://www.moock. org/webdesign/flash/launchwindow/fullscreen*.

Для запуска Notepad на большинстве систем Windows можно использовать:

```
fscommand ("exec", "C:/WINDOWS/NOTEPAD.EXE");
```

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:

```
<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>
```

Чтобы вызвать предшествующую функцию JavaScript *testmovie_DoFSCommand( )* из фильма *flash-to- javascript.swf*, можно использовать:

```
fscommand("hello", "world");
```

***См. также***

Дополнительные сведения о *fscommand( )* и управлении Flash с помощью JavaScript см. по адресу:

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

*http://www.macromedia.com/support/flash/publishexport/scriptingwithflash*

---

**getсвойство( ) глобальная функция** получить значение свойства клипа

***Доступность*** Flash 4; устарела во Flash 5

***Краткий обзор*** `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 преобразуются в пути при использовании в строковом контексте.

*property* Имя извлекаемого встроенного свойства. Должно быть идентификатором, а не строкой (т. е. `_x`, а не `"_x"`).

***Возвращаемое значение***

Значение *property* для *movieClip*.

***Описание***

Функция *getProperty( )* получает одно из встроенных свойств 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 `.` 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 одному и тому же свойству с использованием операторов точки и []:

```
ball._x;
_root.ball._x;
ball["_x"];
_root["ball"]["_x"];
```

***См. также***

*setProperty( )*; раздел ««Объектность» клипов» в главе 13.

---

**getTimer( ) глобальная функция** — определить возраст фильма в миллисекундах

***Доступность*** Flash 4 и выше

***Краткий обзор*** `getTimer()`

***Возвращаемое значение***

Количество миллисекунд, истекших с начала воспроизведения фильма.

***Описание***

Функция *getTimer( )* показывает продолжительность воспроизведения фильма в миллисекундах. Можно многократно пользоваться *getTimer( )* для управления выполнением блока кода во времени или добавления в фильм функций, зависящих от времени, скажем, обратного отсчета в видеоигре. Например, к клипу с текстовым полем `counterOutput`, можно прикрепить следующий код, который осуществляет обратный отсчет от 60 до 0:

```
onClipEvent (load) {
  // Записать текущее время
  var startTime = getTimer();
  // Установить количество секунд для обратного отсчета
  var countAmount = 60;
  // Инициализировать переменную, в которой отслеживается
  // истекшее время
  var elapsed = 0;
}

onClipEvent (enterFrame) {
  // Узнать, сколько времени прошло
  elapsed = getTimer() - startTime;
  // Если времени прошло меньше, чем продолжительность обратного
  // отсчета...
  if (elapsed < countAmount * 1000) {
    // ...вывести в текстовом поле оставшееся время
    counterOutput = countAmount - Math.floor(elapsed / 1000);
```

```
  } else {
    // ...иначе отсчет завершен, о чем сообщается пользователю
    counterOutput = "Время истекло!";
  }
}
```

Чтобы узнать продолжительность воспроизведения фильма в целых секундах, а не миллисекундах, нужно разделить значение, возвращенное *getTimer( )*, на 1000 и убрать дробную часть с помощью *Math.floor( )*, *Math.round( )* или *Math.ceil( )*. Например:

```
numSeconds = Math.floor(getTimer()/1000);
```

***Пример***

Следующий код перемещается между двумя кадрами, пока возраст фильма не превысит 10 секунд, а затем воспроизводит фильм:

```
now = getTimer();
if (now > 10000) {
  play();
} else {
  gotoAndPlay(_currentframe - 1);
}
```

***См. также***

*Date( )*, *Date* класс.

---

**getURL( ) глобальная функция** — загрузить документ в броузер, выполнить сценарии на стороне сервера или генерировать события Macromedia Director

***Доступность*** Flash 2 и Flash 3; усилена во Flash 4 добавлением параметра *method*; Flash 5

***Краткий обзор***

```
getURL (URL)
getURL (URL, window)
getURL (URL, window, method)
```

***Аргументы***

*URL* — Строка, задающая абсолютное или относительное местонахождение документа, который нужно загрузить, или внешнего сценария, который нужно выполнить.

*window* — Необязательная строка, задающая имя окна броузера или фрейма, в который должен быть загружен документ. Может быть пользовательским именем или одним из четырех зарезервированных: `"_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 всегда использует метод "GET" независимо от того, что указано в *method*.

***Описание***

Функция *getURL( )* используется для того, чтобы:

- Загрузить документ (обычно веб-страницы) в фрейм или окно броузера
- Выполнить сценарий на стороне сервера и получить результат во фрейм или окно броузера
- Выполнить код JavaScript в веб-броузере
- Генерировать события из составляющих Flash, импортированных в Director как спрайты

Чтобы загрузить документ в текущее окно или фрейм, просто укажите URL документа, не передавая аргументы *window* или *method*. Естественно, 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, имеющие имя, 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 внешним видом нового окна (размером, конфигурацией панелей управления, местонахождением и пр.). Чтобы открывать настраиваемые окна с помощью *getURL( )*, нужно вызвать на содержащей фильм странице функцию JavaScript. Приемы открытия окон с помощью JavaScript описаны по адресу *http://www.moock.org/webdesign/flash*.

С помощью функции *getURL( )* можно также посылать переменные удаленным приложениям или сценариям на сервере. Чтобы передать переменные текущей временной диаграммы во внешний сценарий, нужно 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 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).

Чтобы загрузить результаты выполнения сценария в текущие фрейм или окно, нужно задать в качестве аргумента *window* значение "_self":

```
getURL("http://www.someserver.com/cgi-bin/search.pl", "_self", "GET");
```

Если значением аргумента *method* является "GET", 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* содержит "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.

Поскольку для большинства веб-серверов максимальная допустимая длина URL лежит между 255 и 1024 символами, при передаче больших объемов данных используйте "POST", а не "GET".

Обратите внимание, что любые данные, возвращаемые сценарием, который вызван *getURL( )*, выводятся как обычное веб-содержание в броузере, а не во Flash. Чтобы получить результаты выполнения сценария во Flash, используйте *loadVariables( )*.

Функция *getURL( )* может применяться и с другими протоколами помимо «http:», как показано в табл. 6.

*Таблица 6. Протоколы, поддерживаемые getURL*

| Протокол | Формат | Назначение |
|---|---|---|
| event | "event: *eventName params*" | Послать событие в Director, если принадлежность Flash является спрайтом Director. |
| file | "file:///driveSpec/folder/filename" | Обратиться к локальному файлу. |

*Таблица 6 (продолжение)*

| Протокол | Формат | Назначение |
|---|---|---|
| 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: *comand*" | Выполнить команду Lingo, если принадлежность Flash является спрайтом Director. |
| print | "print:", "*targetClip*" | Вывод во Flash 4, до появления функции *print( )* во Flash 5. |
| vbscript | "vbscript: command" | Выполнить в броузере команду VBScript. |

Как показывает табл. 6, если принадлежность Flash импортируется в файл Macromedia Director, с помощью *getURL( )* можно генерировать события Lingo или выполнять команды Lingo. (Lingo – это язык сценариев в Director, аналогичный 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*. Вот команда *getURL( )*, генерирующая событие с именем «myEvent» и передающая 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 Flash в Director, чтобы получить событие. Заметим, что спрайт в Director примерно соответствует экземпляру клипа во Flash; ключевое слово Lingo *put* родственно команде ActionScript's *trace( )*, а && в Lingo является оператором конкатенации строк:

```
on myEvent msg
  put "The message received from Flash was " && msg
end
```

Можно также спровоцировать запуск Lingo из спрайта Flash в Director с помощью ключевого слова "lingo:":

```
getURL ("lingo: beep");  // Пусть Director даст звуковой сигнал
```

Отметим, что Director 8.0 не может импортировать файлы *.swf* Flash 5, но ожидается скорое появление обновленной версии принадлежности Flash Xtra.

Наконец, с помощью *getURL( )* можно выполнять код JavaScript. Вызовем простую функцию *alert* JavaScript с помощью *getURL( )*:

```
getURL ("javascript: alert('hello world');");
```

Заметим, что выполнение кода JavaScript из URL не действует в Internet Explorer 4.5 для Macintosh.

***Пример***

Вот код стандартной кнопки со ссылкой на веб-страницу:

```
on (release) {
  getURL("http://www.macromedia.com/");
}
```

*Ошибки в реализации*

Internet Explorer 4.5 (IE 4.5) и более старые версии для Macintosh не поддерживают метод `"POST"` в вызове *getURL( )*. Для обслуживания этих броузеров используйте `"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 Macintosh относительные ссылки разрешаются относительно расположения файла *.swf*, а не файла HTML, что приводит к ошибкам, если они находятся в разных каталогах. Чтобы избежать этой проблемы, располагайте файлы *.swf* и *.html* в одном и том же каталоге или указывайте при вызове *getURL( )* абсолютные адреса URL:

```
getURL ("http://www.someserver.com/")
```

***См. также***

*loadVariables( )*, *MovieClip.getURL( )*, *movieClip.loadVariables( )*; раздел «<A> (якорь или гипертекстовая ссылка)» в главе 18 «Экранные текстовые поля».

---

**getVersion( ) глобальная функция** узнать платформу и номер версии Flash Player

***Доступность*** Flash 5

***Краткий обзор*** `getVersion()`

***Возвращаемое значение***

Строка с данными о платформе и версии Flash Player, в котором расположен текущий фильм.

***Описание***

Функция *getVersion( )* сообщает о платформе и версии Flash Player, используемого для просмотра фильма. Она может применяться для условного выполнения различного кода для определенных версий Flash Player или в определенных операционных системах. Строка, возвращаемая *getVersion( )*, имеет вид:

```
platform majorVersion,minorVersion,buildNumber,patch
```

Здесь *platform* указывает на код платформы (`"WIN"`, `"MAC"` или `"UNIX"`), затем идут 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) под Macintosh
UNIX 4,0,12,0  // Версия 4.0, сборка 12 (4.0r12) под Unix
```

Несмотря на предложение в документации Macromedia обратного, *getVersion( )* работает в режиме Test Movie редактора Flash. Она сообщает номер версии проигрывателя, встроенного в редактор (который может не совпадать с версией самого редактора). Например, в редактор Flash 5 встроена версия проигрывателя 5.0 r30, поэтому его функция *getVersion( )* сообщает:

```
WIN 5,0,30,0
или
MAC 5,0,30,0
```

При каждом появлении новой основной или дополнительной версии редактора *buildNumber* опять обнуляется. Однако типичный цикл развития редактора Flash таков, что изготавливается несколько сборок Flash Player, прежде чем появляется окончательная версия редактора. Поэтому номер сборки первой новой основной версии проигрывателя обычно больше 0. Например, Flash 5 Player был впервые официально выпущен как Build 30. Можно предположить, что при выходе Flash 6 проигрыватель Flash 6 Player будет возвращать нечто вроде такого:

```
WIN 6,0,xx,0
```

Но если фильм, созданный во Flash 6, будет воспроизводиться прежним Flash 5 Player, *getVersion( )* по-прежнему возвратит:

```
WIN 5,0,30,0
```

Обычно интерес представляют только платформа, основной номер версии и номер сборки (build) проигрывателя Player. Для извлечения интересующей нас части строки, возвращаемой *getVersion( )*, можно воспользоваться средствами работы со строками, описанными в главе 4 «Элементарные типы данных», либо создать свой объект, в котором каждому свойству присвоена соответствующая часть строки *getVersion( )*, как показано в следующем примере.

Если не нужно создавать внутренние блоки кода ActionScript для некоторой конкретной версии Flash Player, то JavaScript и VBScript предоставляют лучшие средства для определения номера версии, броузера и автоматической переадресации страниц. Кроме того, *getVersion( )* можно использовать только тогда, когда у пользователя стоит версия проигрывателя 5.0 или выше. Подробности относительно обнаружения наличия Flash Player и определения его версии с помощью JavaScript и VBScript можно найти по адресу *http://www.moock.org/webdesign/flash/detection/moockfpi*.

### *Пример*

Следующий код извлекает различные части строки, возвращаемой *getVersion( )*, и сохраняет их в виде свойств объекта, к которым легко обращаться:

```
// Расщепить строку getVersion() на удобные части
var version = getVersion();
var firstSpace = version.indexOf(" ");
var tempString = version.substring(firstSpace + 1, version.length);
var tempArray = tempString.split(",");

// Присвоить объекту различные части строки getVersion().
// Обратите внимание на преобразование номеров версий 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]);

// Теперь используем объект для выполнения кода для конкретной версии
if (thePlayer.platform == "WIN") {
  // Здесь выполняется код для Windows.
} else if (thePlayer.platform == "MAC") {
  // Здесь выполняется код для Mac.
} else if (thePlayer.platform == "UNIX") {
  // Здесь выполняется код для Unix.
}

if ((thePlayer.majorVersion == 5) && (thePlayer.build == 30))  {
  trace ("Рекомендую обновить проигрыватель для устранения ошибок вывода
          текста ");
}
```

***См. также***

`$version`; приложение C «Обратная совместимость».

---

**gotoAndPlay( ) глобальная функция** — переместить воспроизводящую головку в заданный клип, затем воспроизвести текущий клип или фильм

***Доступность*** Flash 2 и выше

***Краткий обзор***

```
gotoAndPlay(frameNumber)
gotoAndPlay(frameLabel)
gotoAndPlay(scene, frameNumber)
gotoAndPlay(scene, frameLabel)
```

***Аргументы***

*frameNumber* Положительное целое число, задающее номер кадра, в который воспроизводящая головка текущей временной диаграммы должна переместиться перед воспроизведением. Если `frameNumber` меньше 1 или больше числа кадров на временной диаграмме, головка направляется в первый или последний кадры, соответственно.

*frameLabel* Строка, задающая метку кадра, в который воспроизводящая головка текущей временной диаграммы должна переместиться перед воспроизведением. Если `frameLabel` не найдена, головка направляется в первый кадр временной диаграммы.

*scene* Необязательная строка, задающая имя сцены, содержащей заданные `frameNumber` или `frameLabel`. Если не задана, предполагается текущая сцена.

***Описание***

При вызове без аргумента `scene` функция *gotoAndPlay( )* отправляет воспроизводящую головку текущей временной диаграммы в кадр, заданный `frameNumber` или `frameLabel`, и затем воспроизводит временную диаграмму с этого места. «Текущей временной диаграммой» служит клип или фильм, из которого вызвана функция *gotoAndPlay( )*.

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`, и используется текущая сцена.

При вызове с аргументом `scene` функция *gotoAndPlay( )* перемещает воспроизводящую головку в указанный номером или меткой кадр заданной сцены и воспроизводит эту сцену. При использовании аргумента `scene` функция *gotoAndPlay( ) должна* вызываться из временной диаграммы `_root`; в противном случае, операция не выполняется, и воспроизводящая головка не перемещается в указанный кадр. Обратите внимание, что сцены выравниваются в одну временную диаграмму во время воспроизведения. Это значит, что если временная диаграмма сцены 1 содержит 20 кадров, а временная диаграмма сцены 2 содержит 10 кадров, то направить головку 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. Обычно 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. Для воспроизведения других клипов нужно вызывать отдельные команды *play( )* или *gotoAndPlay( )* для каждого клипа. Чтобы применить функцию *gotoAndPlay( )* к клипу, находящемуся вне текущей временной диаграммы, нужно использовать формат метода клипа `myClip.gotoAndPlay()`.

***Ошибки в реализации***

В сборке Flash Player 5.0r30 функция *gotoAndPlay( )* не работает при вызове из обработчика события *onClipEvent( )* с заданием строкового литерала

в параметре *frameLabel*. Чтобы обойти эту ошибку, можно использовать версию в виде метода клипа и слово this для задания текущего клипа, т. е. 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);
```

***См. также***

*gotoAndStop( )*, *MovieClip.gotoAndPlay( )*, *play( )*, *stop( )*.

---

**gotoAndStop( ) глобальная функция** — переместить воспроизводящую головку в указанный кадр и остановить текущий клип

***Доступность*** Flash 2 и выше

***Краткий обзор***

```
gotoAndStop(frameNumber)
gotoAndStop(frameLabel)
gotoAndStop(scene, frameNumber)
gotoAndStop(scene, frameLabel)
```

***Аргументы***

*frameNumber* — Положительное целое число, задающее номер кадра, в который должна переместиться воспроизводящая головка текущей временной диаграммы. Если frameNumber меньше 1 или больше числа кадров на временной диаграмме, воспроизводящая головка направляется в первый или последний кадры, соответственно.

*frameLabel* — Строка, задающая метку кадра, в который должна переместиться воспроизводящая головка текущей временной диаграммы. Если *frameLabel* не найдена, головка направляется в первый кадр временной диаграммы.

*scene* — Необязательная строка, задающая имя сцены, содержащей заданные *frameNumber* или *frameLabel*. Если она не задана, предполагается текущая сцена.

***Описание***

При вызове без аргумента *scene* функция *gotoAndStop( )* отправляет воспроизводящую головку текущей временной диаграммы в кадр, заданный аргументами *frameNumber* или *frameLabel*. «Текущей временной диаграммой» служит клип или фильм, из которого вызвана функция *gotoAnd-*

*Stop( )*. Воспроизводящая головка останавливается в целевом кадре: попав туда, она не начнет перемещаться автоматически.

Если в вызове функции *gotoAndStop( )* указаны два аргумента, предполагается, что первый из них *scene*. Если задан только один аргумент, предполагается, что это *frameNumber* или *frameLabel*, и используется текущая сцена.

При вызове с аргументом *scene* функция *gotoAndStop( )* перемещает воспроизводящую головку в указанный номером или меткой кадр заданной сцены и прекращает воспроизведение. При использовании аргумента *scene* функция *gotoAndStop( ) должна* вызываться из временной диаграммы _root; в противном случае, операция не выполняется, и воспроизводящая головка не перемещается в указанный кадр. Обратите внимание, что сцены во время воспроизведения выравниваются в одну временную диаграмму.

Глобальная функция *gotoAndStop( )* действует только на текущую временную диаграмму. На кадры или состояние других клипов текущей временной диаграммы она не оказывает влияния. Для воспроизведения других клипов нужно вызывать отдельные команды *gotoAndStop( )* для каждого клипа. Чтобы применить функцию *gotoAndStop( )* к клипу, находящемуся вне текущей временной диаграммы, нужно использовать формат метода клипа, *myClip*. gotoAndStop().

### *Ошибки в реализации*

В сборке Flash Player 5.0r30 функция *gotoAndStop( )* не работает при вызове из обработчика события *onClipEvent( )* с заданием строкового литерала в параметре *frameLabel*. Чтобы обойти эту ошибку, можно использовать версию в виде метода клипа и слово this для задания текущего клипа, т. е. 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)
```

### *См. также*

*gotoAndPlay( )*, *MovieClip.gotoAndStop( )*, *play( )*, *stop( )*.

---

**_highquality глобальное свойство** качество вывода Flash Player

***Доступность*** Flash 4; устарело во Flash 5 благодаря _quality

***Краткий обзор*** _highquality

***Тип доступа*** Чтение/запись

***Описание***

Глобальное свойство `_highquality` содержит целое число от 0 до 2, управляющее качеством вывода Flash Player следующим образом:

*0*

Низкое качество. Не производится сглаживание (antialiasing) ни для растровой, ни для векторной графики.

*1*

Высокое качество. Сглаживается векторная графика. Растровая графика сглаживается, когда не воспроизводится анимация.

*2*

Высшее качество. Всегда производится сглаживание как растровой, так и векторной графики.

Начиная с Flash 5, `_highquality` заменено `_quality`, которое может использоваться для задания «среднего» качества дополнительно к «низкому», «высокому» и «высшему».

***См. также***

`_quality`, *toggleHighQuality( )*.

---

**#include директива** импорт текста из внешнего файла ActionScript

***Доступность*** Flash 5

***Краткий обзор*** `#include` *path*

***Аргументы***

*path* Строка, задающая имя и местонахождение импортируемого файла сценария, которое можно задать относительно файла *.fla* или как абсолютный путь (см. примеры ниже). Обратите внимание, что в пути должна использоваться прямая, а не обратная косая черта. 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`. Операция *#include* осуществляется на этапе компиляции, и это означает, что включаемый в фильм текст существовал во время тестирования, экспорта или публикации из среды разработки. Изменения, произведенные во внешнем файле после экспорта фильма, не будут отражены в фильме. Чтобы добавить эти изменения в фильм, его нужно повторно экспортировать.

Директива `#include` используется для включения одного и того же блока кода в несколько сценариев или в разные проекты Flash (подобно тому, как используется внешняя библиотека принадлежностей Flash). Это следует делать для централизации кода, сопровождения кода в системе контроля версий (например, CVS или Microsoft Visual Source Safe), либо при

использовании внешнего текстового редактора, который предпочтительней редактора ActionScript. Это удобно и тогда, когда программист работает отдельно от, скажем, разработчика графических анимаций Flash. Внешние файлы удобны при использовании хранилищ кода, например, библиотек функций, независимых от текущей временной диаграммы или клипа. Менее полезными они оказываются в коде, который требует тесной интеграции с файлом Flash.

***Особенности применения***

Обратите внимание, что директива `#include` начинается с символа (#), не использует скобок, и *не* должна заканчиваться точкой с запятой. Любое предложение `#include`, завершающееся точкой с запятой, вызовет ошибку и не выполнит импорта внешнего сценария. Если файла по указанному пути нет, директива вызовет ошибку, и внешний файл не будет включен. При выполнении синтаксической проверки по команде Check Syntax (Ctrl-T или Command-T) меню панели Actions (которое находится под кнопкой со стрелкой в правом верхнем углу панели) текст во внешнем файле тоже проверяется.

***Пример***

Следующий код импортирует внешний файл *.as* с именем *myScript.as* в текущий файл *.fla*. При использовании относительного пути *myScript.as* должен быть в той же папке, что и файл *.fla*, содержащий директиву включения:

```
#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"
```

Для обозначения папки над текущей используются две точки. 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***

раздел «Размещение кода ActionScript во внешних файлах» в главе 16.

---

**Infinity глобальное свойство** константа, представляющая бесконечно большое число

***Доступность*** Flash 5

***Краткий обзор*** `Infinity`

***Тип доступа*** Только для чтения

***Описание***

В ActionScript любое число, выходящее за положительную границу числового диапазона, обозначается числовой константой `Infinity`. Самое большое допустимое значение в ActionScript представляется числом `Number.MAX_VALUE` (1.7976931348623157e+308).

***Пример***

Результатом вычисления, превосходящим максимальное допустимое число, является `Infinity`. Например:

```
Number.MAX_VALUE * 2;    // Дает Infinity
```

`Infinity` получается также при делении положительного числа на ноль:

```
1000 / 0;                // Дает  Infinity
```

***Особенности применения***

`Infinity` служит сокращенной записью для `Number.POSITIVE_INFINITY`.

***См. также***

`-Infinity`, `Number.POSITIVE_INFINITY`; раздел «Особые значения числового типа данных» в главе 4.

---

**–Infinity глобальное свойство** константа, представляющая бесконечно малое отрицательное число

***Доступность*** Flash 5

***Краткий обзор*** `-Infinity`

***Тип доступа*** Только для чтения

***Описание***

В ActionScript любое число, выходящее за отрицательную границу числового диапазона, представляется числовой константой `-Infinity`. Наименьшим отрицательным значением (имеющим наибольшую абсолютную величину), допустимым в ActionScript, является `-Number.MAX_VALUE`, равное –1. 7976931348623157e+308.

***Пример***

Результатом вычисления, меньшим минимального допустимого отрицательного числа, является `-Infinity`:

```
-Number.MAX_VALUE * 2;    // Дает -Infinity
```

-Infinity получается также при делении отрицательного числа на ноль:

```
-1000 / 0;                // Дает -Infinity
```

***Особенности применения***

-Infinity служит сокращенной записью для Number.NEGATIVE_INFINITY.

***См. также***

Infinity, Number.NEGATIVE_INFINITY; раздел «Особые значения числового типа данных» в главе 4.

---

**int( ) глобальная функция** — отсечь дробную часть числа

***Доступность*** Flash 4; устарела во Flash 5 благодаря аналогичным методам *Math*

***Краткий обзор*** int(*number*)

***Аргументы***

*number* — Число или выражение, возвращающее число, обычно число с дробной (десятичной) частью.

***Возвращаемое значение***

Целая часть *number*.

***Описание***

Функция *int( )* использовалась во Flash 4 в качестве метода извлечения целой части числа грубой силой. Фактически она округляет положительные числа вниз (до наибольших не превосходящих их целых чисел), а отрицательные числа вверх (до наименьших превосходящих их целых чисел). Функция *int( )* работает только с числами в диапазоне от $-2147483648$ ($-2^{31}$) до $2147483647$ ($2^{31}-1$); результаты ее действия на числа вне этого диапазона непредсказуемы. Если *number* является строкой, состоящей только из чисел, *int( )* предварительно преобразует эту строку в число. Если *number* является булевым значением true, *int( )* возвращает 1. Для всех остальных нечисловых данных (включая undefined и null), *int( )* возвращает значение 0.

***Особенности применения***

Функция *int( )* является устаревшей по сравнению с более точными и стандартными методами *Math.floor( )*, *Math.ceil( )* и *Math.round( )*. Для преобразования нечисловых данных в целые или действительные числа обращайтесь к методам *parseInt( )* или *Number( )*.

***Пример***

```
int(4.5)      // Дает 4
int(-4.5)     // Дает -4
int(3.999)    // Дает 3
```

Функцией *int( )* удобна для проверки принадлежности числа к целым, выполняемой путем сравнения исходного числа с результатом функции *int( )*:

```
if (int(x) != x) {
  trace ("Пожалуйста, введите целое число лет своего возраста");
}
```

***См. также***

*Math.ceil( ), Math.floor( ), Math.round( ), Number( ), parseFloat( ), parseInt( ).*

---

**isFinite( ) глобальная функция** проверить, что число меньше Infinity и больше –Infinity

***Доступность*** Flash 5

***Краткий обзор*** `isFinite(number)`

***Аргументы***

*number* Любое число или выражение, дающее числовое значение.

***Возвращаемое значение***

Булево значение: `true`, если число находится в диапазоне между `Number.MAX_VALUE` и –`Number.MAX_ VALUE` (включая границы), и `false`, если нет. Если *number* не относится к типу данных *number*, то предварительно преобразуется в тип *number*.

***Описание***

Функция *isFinite( )* просто проверяет, находится ли число в допустимом для ActionScript диапазоне. Используйте *isFinite( )* перед выполнением 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(Number.MAX_VALUE * 2)   // Возвращает false
```

***См. также***

`-Infinity`, `Infinity`, *isNan( )*, *Number.MAX_VALUE*, *Number.MIN_VALUE*; раздел «Особые значения числового типа данных» в главе 4.

---

**isNaN( ) глобальная функция** проверить равенство с особым значением `NaN`

***Доступность*** Flash 5

***Краткий обзор*** `isNaN(`*value*`)`

***Аргументы***

*value* Проверяемое выражение.

***Возвращаемое значение***

Булево значение: `true`, если *value* является особым числовым значением `NaN`; в противном случае, `false`.

***Описание***

Чтобы проверить, равно ли значение специальному числовому значению `NaN`, необходимо воспользоваться функцией *isNaN( )*, поскольку `NaN` не равен сам себе себе в обычной проверке. Например, выражение:

```
NaN == NaN;
```

возвращает значение `false`. Функцию *isNaN( )* часто используют для проверки того, произошла ли в выражении кода математическая ошибка (такая, как деление нуля на ноль) или неудачным оказалось преобразование величины в допустимое число. Так как *isNaN( )* возвращает `true`, если выражение *не* является допустимым числовым выражением, то вместе с *isNaN( )* часто используется оператор логического «НЕ» (`!`) (то, что *не* является не числом, 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";
// Проверить, что x является допустимым числом, перед использованием в
// математическом выражении. 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`; раздел «Особые значения числового типа данных» в главе 4.

---

**Key объект** определить состояние клавиш на клавиатуре

***Доступность*** Flash 5

***Краткий обзор***

```
Key.property
Key.methodName()
```

***Свойства***

В табл. 7 перечислены свойства объекта *Key*.

*Таблица 7. Свойства 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 |

*Таблица 7 (продолжение)*

| Свойство | Эквивалентный код клавиши | Свойство | Эквивалентный код клавиши |
|---|---|---|---|
| ESCAPE | 27 | TAB | 9 |
| HOME | 36 | UP | 38 |

***Методы***

| | |
|---|---|
| *getAscii( )* | Возвращает ASCII-значение 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 |

***Описание***

Объект *Key* используется для определения того, какие клавиши нажаты в данный момент и какая клавиша была нажата последней. Его можно использовать для создания интерфейсов, управляемых клавиатурой, например, для игры, в которой движение космического корабля управляется клавишами со стрелками.

Поскольку не все клавиатуры одинаковы, создавать интерфейсы, управляемые с клавиатуры, бывает трудно. 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 ввод с клавиатуры необходим постоянно, например, в видеоиграх.
- Узнать, какая клавиша была нажата последней, можно с помощью методов *getCode( )* и *getAscii( )*. Это рекомендуется в типичных интерфейсах, управляемых клавиатурой, в которых конкретные операции выполняются при нажатии клавиш. Обычно эти методы используются в обработчике события *keyDown* для различения клавиш. При этом нет необходимости постоянно проверять (т. е. *опрашивать*) последнюю нажатую клавишу. 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 обработчике события *keyDown*, потому что гарантируется вызов обработчика один и только один раз для каждого нажатия клавиши.

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. Благодаря использованию кода клавиш можно идентифицировать клавиши даже при воспроизведении фильма в раз-

ных 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. Коды клавиш для 0…9 тоже совпадают со значениями этих чисел в Latin 1 (48–57). Коды других клавиш не соответствуют кодовым позициям в Latin 1. Однако многие коды небуквенных и нецифровых клавиш доступны как свойства *Key*. Например, не нужно помнить, что для стрелки вверх код клавиши равен 38: можно просто использовать свойство `Key.UP`. В следующем коде проверяется, нажата ли в данный момент клавиша «стрелка вверх»:

```
if (Key.isDown(Key.UP)) {
  trace("The up arrow is being pressed");
}
```

При работе с клавишами, не являющимися буквами или цифрами и недоступными в виде свойства *Key* – например, с функциональными клавишами (F1, F2, и так далее), – надежнее всего быстро создать тестовый фильм и узнать код нужной клавиши, например:

```
trace(Key.getCode());
```

Коды клавиш перечислены в приложении B.

***См. также***

разделы «KeyUP» и «KeyDown» в главе 10 и приложение B «Набор символов Latin 1 и коды клавиш».

---

**Key.getAscii( ) метод** возвращает код ASCII последней нажатой клавиши

***Доступность*** Flash 5

***Краткий обзор*** `Key.getAscii()`

***Возвращаемое значение***

Целое число, представляющее код ASCII последней нажатой клавиши.

***Описание***

Метод *getAscii( )* возвращает ASCII-значение последней нажатой клавиши. Поскольку ASCII-значение есть не у всех клавиш, *getAscii( )* обычно используется для определения букв и цифр из набора символов Latin 1 (западноевропейские языки). 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 клавиатуре.

Для обнаружения нажатия определенных клавиш по их относительному физическому положению на клавиатуре, а не значению ASCII (что желательно при крестообразном расположении клавиш для управления игрой), используйте *getCode( )*.

***Пример***

В следующем примере показано обнаружение нажатий клавиш в простой игре в «виселицу». Обработчик события *keyDown* определяет нажатую клавишу и добавляет ее в список предположений пользователя:

```
onClipEvent (keyDown) {
  var lastKey = Key.getAscii();
  guessNum++;
  userGuesses[guessNum] = String.fromCharCode(lastKey);
}
```

***См. также***

*Key.getCode( )*; приложение B и раздел «KeyDown» в главе 10.

---

**Key.getCode( ) метод** возвращает код последней нажатой клавиши

***Доступность*** Flash 5

***Краткий обзор*** `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 платформо-независимый способ ссылки на определенные клавиши. Метод *getCode( )* может также применяться для различения клавиш с одинаковыми значениями ASCII. Например, он может различить клавишу 8 на основной клавиатуре и клавишу 8 на числовой клавиатуре, чего *getAscii( )* сделать не может. Зато *getCode( )* не может различить прописные и строчные буквы (например, *A* и *a* используют один код клавиши, потому что вводятся одной и той же клавишей).

Многие часто используемые значения кодов клавиш доступны как свойства объекта *Key* (например, `Key.UP`, `Key.BACKSPACE`). Чтобы определить код конкретной клавиши, обратитесь к приложению B или создайте тестер кодов клавиш сами:

1. Создайте новый документ Flash.
2. В кадр 2 временной диаграммы добавьте кадр.
3. В кадр 1 поместите код:

```
trace(Key.getCode());
```

4. Выберите меню Control → Test Movie.
5. Щелкните по рабочему столу фильма.
6. Нажмите клавишу. Ее код появится в окне Output.

### *Пример*

В отличие от *isDown( )*, метод *getCode( )* полезен при создании интерфейсов, в которых нажатие отдельной клавиши производит один непосредственный результат. Например, можно позволить пользователю пропустить вводную часть фильма нажатием пробела. Когда нажимается пробел, воспроизводящая головка перемещается в основной интерфейс фильма (на главной временной диаграмме), например:

```
// Код в клипе введения
onClipEvent (keyDown) {
  if (Key.getCode() == Key.SPACE) {
    _root.gotoAndStop("mainInterface");
  }
}
```

### *См. также*

*Key.getAscii( ), Key.isDown( )*; приложение B и раздел «KeyDown» в главе 10.

---

**Key.isDown( ) метод** — проверяет, нажата ли конкретная клавиша в данный момент

***Доступность*** Flash 5

***Краткий обзор*** `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*. Предоставляет произвольный и незамедлительный доступ к состоянию клавиатуры, и больше всего подходит для систем, требующих постоянного ввода с клавиатуры или обнаружения одновременного нажатия нескольких клавиш.

Важным преимуществом *isDown( )* по сравнению с методами *getCode( )* и *getAscii( )* является его способность обнаруживать одновременное нажатие нескольких клавиш. Например, проверив обе клавиши `Key.UP` и `Key.RIGHT`, можно установить, что космический корабль в игре должен быть перемещен по диагонали. При некотором расположении конкретных проверяемых клавиш максимальное число одновременно обнаруживаемых клавиш может не превышать трех.

### *Пример*

Метод *isDown( )* обычно используется 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 нужно обнаружить две клавиши, нажатые одновременно, следует разделить предложения *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 отдельном предложении *if*. Работающую версию этого примера космического корабля можно получить в сетевом хранилище кода:

```
// Код в клипе космического корабля
onClipEvent (enterFrame) {
  if (Key.isDown(Key.LEFT)) {          // Стрелка влево
    _rotation -= 10;
  } else if (Key.isDown(Key.RIGHT)) {  // Стрелка вправо
    _rotation += 10;
  }
  if (Key.isDown(Key.UP)) {            // Стрелка вверх
    thrust += 10;
  }
}
```

***См. также***

*Key.getCode( )*; раздел «KeyDown» в главе 10.

---

**Key.isToggled( ) метод** — проверяет, включены ли клавиши Caps Lock, Num Lock или Scroll Lock

***Доступность*** Flash 5

***Краткий обзор*** `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, что клавиша с кодом *keycode* включена (`true`) или выключена (`false`).

***Описание***

Метод *isToggled( )* определяет состояние специальных клавиш Caps Lock, Num Lock или Scroll Lock. В отличие от других клавиш, у этих есть состояние «включено» и «выключено», показывающее, действует ли функция, которую они представляют. Значение, возвращаемое методом *isToggled( )* сообщает о том, действует функция клавиши или нет. (Хотя *isToggled( )* фактически работает с любым кодом клавиши, возвращаемое им значение полезно только для специальных клавиш, действующих как переключатели. Для выяснения состояния других клавиш используются методы *isDown( )*, *getCode( )* или *getAscii( )*.)

**_leveln глобальное свойство** уровень документа во Flash Player

***Доступность*** Flash 3 и выше

***Краткий обзор*** _level0
_level1
_level2
...
_level*n*

***Тип доступа*** Только для чтения

***Описание***

Во Flash Player для одновременного воспроизведения могут быть загружены несколько файлов *.swf*. Каждый загруженный *.swf* располагается на собственном уровне в стеке уровней документа. (Файл *.swf*, находящийся на более высоком уровне, закрывает более низкие уровни, если они занимают одну и ту же часть рабочего стола.) 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 так далее.

Первый документ, загруженный в любой Flash Player, считается имеющим _level0.

***Пример***

Ссылка _level*n* обычно используется 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();
```

Ссылку _level*n* можно также использовать в качестве значения аргумента *target* нескольких функций, в том числе *loadMovie( )*, *unloadMovie( )*, *loadVariables( )* и *print( )*. Если уровня еще не существует, следует задать ссылку на уровень в кавычках. При использовании без 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, более безопасен следующий подход:

```
loadMovie("myMovie.swf", "_level1");  // Действует, даже если _level1
                                      // не существует
```

Конечно, из других уровней может понадобиться ссылка на исходный уровень с помощью _level0 – например, таким образом:

```
startDrag(_level0, true);
```

***См. также***

*loadMovie( )*, *unloadMovie( )*, -root; разделы «Импорт внешних фильмов» и «Порядок расположение фильмов и экземпляров в стеке» в главе 13 .

---

**loadMovie( ) глобальная функция** загрузить в проигрыватель файл *.swf*

***Доступность*** Flash 4 и выше. Функция *loadMovie( )* во Flash 5 соответствует *Load Movie* с путем к цели во Flash 4

***Краткий обзор***

```
loadMovie(URL, target)
loadMovie(URL, target, method)
```

***Аргументы***

*URL* Строка, задающая абсолютный или относительный путь к загружаемому файлу *.swf*. Во всех URL должна использоваться прямая косая черта, а абсолютные URL должны содержать ссылку на протокол в виде http:// или file:/// .

*target* Строка, задающая клип или уровень документа, в которых расположится внешний файл *.swf*. Может также являться ссылкой на существующий клип или уровень документа (ссылки преобразуются в пути при использовании в строковом контексте).

*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. Автономная версия Flash Player всегда использует метод «GET» вне зависимости от заданного в *method*.

***Описание***

Функция *loadMovie( )* импортирует во Flash Player файл *.swf*, размещенный по адресу *URL*.

Если *target* является ссылкой на существующий клип или строкой, задающей путь к клипу, загруженный файл *.swf* будет помещен в указанный клип (вызывая удаление предшествующего содержимого). Чтобы загрузить фильм в *текущий* клип, используйте в качестве параметра *target* пустую строку, например:

```
loadMovie("myMovie.swf", "")
```

Если *target* является ссылкой на существующий уровень документа (например, _level2) или строкой, указывающей путь к уровню документа (например, "_level2"), то файл *.swf* помещается на указанный уровень документа. Загрузка фильма на уровень _level0 очищает проигрыватель от всего содержимого и помещает новый файл *.swf* на уровень _level0.

Можно посылать переменные вместе с вызовом *loadMovie( )*, и тогда *URL* обычно является адресом сценария, возвращающего файл *.swf*, соответствующий переданным переменным. Чтобы отправить переменные при вызове *loadMovie( )*, включается аргумент *method* (заданный в "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. Метод "POST" недоступен в автономном Flash Player. Поскольку у большинства веб-серверов длина URL ограничена величиной от 255 до 1024 символов, для передачи больших объемов данных используйте "POST", а не "GET".

Вызовы *loadMovie( )* над веб-сервером, использующие метод "GET", могут передавать переменные в загруженный фильм без вмешательства сценария. 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 главной временной диаграмме этого фильма. Этот прием действует только при обработке запроса *loadMovie( )* веб-сервером. Попытки использовать метод "GET" в *loadMovie( )* для локальных файлов вызывают ошибку «Error opening URL».

***Особенности применения***

Будьте осторожны при использовании ссылок на клипы и уровни в качестве аргумента *target* функции *loadMovie( )*. Если аргумент target дает значение undefined, то функция *loadMovie( )* использует в качестве *target* текущую временную диаграмму. Аналогично, ссылки *target*, дающие пустую строку, заставляют *loadMovie( )* действовать над 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 этого предложения объект _level1 не существовал, то этот код загрузит файл *myMovie. swf* во временную диаграмму, содержавшую предложение *loadMovie( )*, а не на _level1! Чтобы избежать этой проблемы, вместо *loadMovie( )* можно использовать *loadMovieNum( )*. Кроме того, можно указать строку в качестве параметра *target* для *loadMovie( )*:

```
loadMovie("myMovie.swf", "_level1");
```

В этом случае уровень будет создан, если его еще не существует (только _level0 существует по умолчанию во всех фильмах). Дополнительные сведения см. в разделе «Проблемы, связанные с перекрытием методов и глобальных функций» главы 13.

***Пример***

```
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, используя 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");
```

***См. также***

*loadMovieNum( )*, *MovieClip.loadMovie( )*, *unloadMovie( )*; раздел «Импорт внешних фильмов» в главе 13.

---

**loadMovieNum( ) глобальная функция** загрузить внешний файл *.swf* на уровень документа

***Доступность*** Flash 3; усилена во Flash 4 включением параметра *method*; доступна во Flash 5. Функция *loadMovieNum( )* соответствует *Load Movie* Flash 3, которая принимала только числа.

***Краткий обзор***

```
loadMovieNum(URL, level)
loadMovieNum(URL, level, method)
```

***Аргументы***

*URL* Строка, задающая абсолютный или относительный путь к внешнему файлу *.swf*, который должен быть загружен.

*level* Неотрицательное целое или выражение, возвращающее неотрицательное целое число, задающее уровень документа, на котором будет размещен внешний файл *.swf*.

*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. Автономная версия Flash Player всегда использует метод `"GET"` независимо от указанного *method*.

***Описание***

Функция *loadMovieNum( )* почти идентична *loadMovie( )*, но требует задания целевого уровня загрузки в виде числа, а не строки. Это означает, что *loadMovieNum( )* может загружать фильмы только на уровни документов, а не в клипы. Если указанного уровня не существует, он будет создан. Если указанный уровень существует, его содержимое будет заменено новым файлом *.swf*. Допускается загрузка фильма на `_level2`, даже если `_level1` не создан.

Функцию *loadMovieNum( )* можно использовать, когда нужно динамически задавать уровень для загрузки фильма, например:

```
var x = 3;
loadMovieNum("myMovie.swf", x);
```

Того же самого можно достичь путем конкатенации строк и использования обычной функции *loadMovie( )*:

```
loadMovie("myMovie.swf", "_level" + x);
```

***См. также***

*loadMovie( )*, *MovieClip.loadMovie( )*.

---

**loadVariables( ) глобальная функция** получить группу внешних переменных

***Доступность*** Flash 4 и выше

***Краткий обзор***
```
loadVariables(URL, target)
loadVariables(URL, target, method)
```

***Аргументы***

*URL* Строка, задающая путь к источнику переменных – сценарию на стороне сервера, который возвращает переменные, или к 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 (ссылки преобразуются в пути при использовании в строковом контексте).

*method* Необязательная строка, указывающая метод отправки переменных внешнему сценарию. Если она задана, то переменные текущей временной диаграммы передаются сценарию, а *target* получает загруженные переменные. Если эта строка опущена, то переменные загружаются, но не отправляются. Допустимыми значениями для *method* являются `"GET"` и `"POST"`. Этот параметр должен быть литералом, а не переменным или другим выражением. В автономной версии Flash Player всегда используется метод `"GET"` независимо от задания *method*.

***Описание***

Обычно переменные внутри фильмов определяются с помощью ActionScript. Однако посредством *loadVariables( )* можно также импортировать в фильм переменные из текстового файла или выполняемого на сервере приложения – например, сценария Perl. 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 нужно использовать в качестве значения аргумента *target* пустую строку, например:

```
loadVariables("myVars.txt", "");  // Загружает переменные из myVars.txt
                                  // в текущую временную диаграмму
```

Вне зависимости от того, будут ли загружаемые переменные находиться в текстовом файле, или их создаст сценарий, они должны иметь формат, соответствующий кодировке 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 (+).
- Любой символ, не являющийся пробелом, цифрой (1–9) или буквой Latin 1 (a–z, A–Z), должен быть заменен шестнадцатеричной escape-последовательностью вида `%xx`, где `xx` является кодовой позицией символа в Latin 1 в шестнадцатеричном виде.

Вот пример содержимого текстового файла, импортируемого во Flash посредством *loadVariables( )*. Импортируются переменные `name` и `address`, имеющие значения `"stephen"` и `"65 nowhere st!"`, соответственно:

```
name=stephen&address=65+nowhere+st%21
```

Текстовый файл, используемый с *loadVariables( )*, является обычным текстовым файлом, содержащим переменные в кодировке URL, как показано выше. 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");
```

*loadVariables( )* можно также использовать со сценарием или приложением сервера, выводящими переменные в кодировке URL. Когда сценарий посылает данные в фильм Flash в ответ на вызов функции *loadVariables( )*, он должен задать MIME-тип данных как `"application/x-www-urlform-encoded"`. Вот обычная команда задания MIME в сценарии Perl:

```
print "Content-type: application/x-www-urlform-encoded\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( )* задается равным `"GET"` или `"POST"`. Переменные пересылаются в формате кодировки URL. Если *method* задан как `"GET"`, переменные передаются как строка запроса в *URL* сценария. Если *method* установлен как `"POST"`, переменные посылаются после заголовка HTTP POST. Метод `"POST"` недоступен в автономном Flash Player. Поскольку для большинства веб-серверов максимальная допустимая длина URL лежит между 255 и 1024 символами, при передаче больших объемов данных используйте `"POST"`, а не `"GET"`.

Из соображений безопасности *loadVariables( )* действует только с хостами домена, из которого был загружен фильм. Правила, которым подчиняется

*loadVariables( )*, перечислены в табл. 8. Эти меры безопасности оказывают воздействие только на подключаемые модули и элементы ActiveX броузера Flash Player; автономный проигрыватель может загружать переменные из любого домена.

Ограничения на домены являются умышленной защитной функцией во Flash, но их можно обойти с помощью прокси-сценария, выполняемого на сайте X, действующего как посредник между Flash и сайтом Y, или псевдонима DNS на сайте X, который указывает на сайт Y. Дополнительную информацию можно получить по адресу:

*http://www.macromedia.com/support/flash/ts/documents/loadvars_security.htm*

*Таблица 8. Ограничения loadVariables( ) по доменам, связанные с безопасностью*

| Исходный домен фильма | Как подключитьсяя | Разрешено? |
|---|---|---|
| *www.somewhere.com* | *www.somewhere.com* | Да |
| *www.somewhere.com* | *other.somewhere.com* | Да |
| *www.somewhere.com* | *www.somewhere-else.com* | Нет |
| *www.somewhere.com* | *somewhere.com* | Да |
| *somewhere.com* | *www.somewhere.com* | Да |

***Особенности применения***

Результаты многократных обращений *loadVariable( )* по одному и тому же URL сценария могут некоторыми броузерами кешироваться, в результате чего новые данные с сервера не загружаются. Чтобы избежать этой проблемы, добавляйте в каждый вызов *loadVariables( )* фиктивную переменную, чтобы URL оказывался уникальным. Вот пример генерации уникального URL путем добавления времени в миллисекундах:

```
loadVariables("http://www.mysite.com/cgi-bin/myScript.pl?cacheKiller="
              + getTimer(), serverResponse);
```

***Ошибки в реализации***

Метод POST не поддерживается в Internet Explorer 4.5 для Macintosh. Эта проблема была исправлена в версии 5 броузера.

***См. также***

*loadVariablesNum( )*, *MovieClip.loadVariables( );* главу 17 «Формы Flash».

---

**loadVariablesNum( ) глобальная функция** прикрепить группу внешних переменных к уровню документа

***Доступность*** Flash 5. Во Flash 4 для размещения переменных на уровне документа используется *Load Variables*.

***Краткий обзор***

```
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 число или выражение, возвращающее неотрицательное целое число. Задает 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 извлекаются, но не загружаются. Допустимыми значениями для *method* являются `"GET"` и `"POST"`. Этот параметр должен быть литералом, а не переменной или другим выражением. В автономной версии Flash Player всегда используется метод `"GET"` независимо от задания *method*.

***Описание***

Функция *loadVariablesNum( )* почти идентична *loadVariables( )*, но требует задания целевого *level* как числа, а не как строки. 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);
```

Того же самого можно достичь путем конкатенации строк и использования обычной функции *loadVariables( )*:

```
loadVariables("myVars.txt", "_level" + myLevel);
```

***См. также***

*loadVariables( )*.

---

**Math объект** — доступ к математическим функциям и константам

***Доступность*** — Flash 5

***Краткий обзор***

```
Math.propertyName
Math.methodName()
```

***Свойства***

*E* — Константа *e* – основание натуральных логарифмов, приблизительно 2.71828.

*LN10* — Натуральный логарифм 10 ($\log_e 10$), приблизительно 2.30259.

| | |
|---|---|
| *LN2* | Натуральный логарифм 2 ($\log_e 2$), приблизительно 0.69315. |
| *LOG10E* | Логарифм *e* по основанию 10, приблизительно 0.43429. |
| *LOG2E* | Логарифм *e* по основанию 2, приблизительно 1.44270. Смотрите замечание об ошибке в подробном описании. |
| *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 до ближайшего целого сверху. |
| *cos( )* | Вычислить косинус угла. |
| *exp( )* | Возвести *e* в заданную степень. |
| *floor( )* | Возвратить ближайшее целое число, 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 используются для выполнения достаточно сложных вычислений с относительной легкостью.

Обратите внимание, что свойства и методы объекта *Math* могут использоваться в фильмах, экспортируемых в формате Flash 4, при этом Flash будет выполнять расчеты приближенно. Полученные значения могут быть достаточно хорошими приближениями, но не всегда совпадут с результатами расчетов во Flash 5. Значения, получаемые во Flash 4, достаточно

точны для таких приложений, как вывод графики, но недостаточно точны для важных финансовых или инженерных расчетов.

Заметьте, что тригонометрические функции требуют измерения углов в радианах, в то время как свойство Flash *MovieClip*._rotation измеряется в градусах. Окружность содержит 2π радиан (1 радиан составляет примерно 57.3 градуса). Для преобразования из радиан в градусы используется формула:

```
degrees = (radians / Math.PI) * 180;
```

Для преобразования из градусов в радианы в используется формула:

```
radians = (degrees / 180) * Math.PI;
```

***См. также***

*Math.atan2( )*, *Math.cos( )*; раздел «Числовой тип данных» в главе 4.

---

**Math.abs( ) метод** вычислить абсолютную величину числа

***Доступность*** Flash 5; может использоваться при экспорте в фильмы Flash 4

***Краткий обзор*** `Math.abs(x)`

***Аргументы***

***x*** Положительное или отрицательное число.

***Возвращаемые значения***

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 положительные числа неизменными, а отрицательные превращает 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** вычислить арккосинус числа

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.acos(x)`

***Аргументы***

| | |
|---|---|
| *x* | Число между −1.0 и 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 (т. е. от 0 до 3.14159...).

***Пример***

```
trace (Math.acos(1.0));  // Выводит: 0
trace (Math.acos(0.0));  // Выводит: 1.5707... (т. е. pi/2)
```

***См. также***

*Math.asin( )*, *Math.atan*(), *Math.cos( )*.

---

**Math.asin( ) метод** вычислить арксинус числа

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.asin(x)`

***Аргументы***

| | |
|---|---|
| *x* | Число между −1.0 и 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...  (т. е. pi/2)
trace (Math.asin(0.0));  // Выводит: 0
```

***См. также***

*Math.acos*, *Math.atan*, *Math.sin*.

**Math.atan( ) метод** вычислить арктангенс числа

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.atan(x)`

***Аргументы***

***x*** Число между `-Infinity` и `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... (т. е. -pi/2)
```

***См. также***

*Math.acos( )*, *Math.asin( )*, *Math.tan( )*.

---

**Math.atan2( ) метод** определить угол по точке

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.atan2(y, x)`

***Аргументы***

***y*** y-координата точки.

***x*** x- координата точки.

***Возвращаемое значение***

Угол, в радианах, точки (*x*, *y*) от центра круга, измеряемый против часовой стрелки от положительной горизонтальной оси круга (т. е. оси X). Изменяется в диапазоне от $\pi$ до $-\pi$. (Отрицательные значения соответствуют углам ниже оси X).

***Описание***

Метод *atan2( )*, как и *atan( )*, вычисляет арктангенс, но использует в качестве аргументов x- и y-координаты, 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
```

***Особенности применения***

Обратите внимание, что *y*-координата передается *atan2( )* в качестве первого аргумента, в то время как *x*-координата передается как второй аргумент. Это обязательно, и сделано преднамеренно, потому что отражает структуру тангенса, являющегося отношением стороны, противоположной углу (y) к стороне, примыкающей к углу (x).

***Пример***

Метод *atan2( )* можно использовать, чтобы поворачивать клип в сторону движущейся цели. Следующий пример, который можно загрузить из сетевого хранилища кода, показывает код, поворачивающий текущий клип в направлении указателя мыши. Его можно использовать для ориентации неприятельского космического корабля в направлении космического корабля играющего:

```
// Повернуть клип в направлении мыши
onClipEvent (load) {
  // Преобразовать радианы в градусы. 2*pi соответствует 360 градусам.
  function radiansToDegrees(radians) {
    return (radians/Math.PI) * 180;
  }
}

onClipEvent (enterFrame) {
  // Создать объект точки, в котором хранятся x- и y- координаты этого
  // клипа относительно точки регистрации его родителя
  point = {x:_x, y:_y};
  // Преобразовать локальные (родительские) координаты в глобальные
  // (рабочего стола)
  _parent.localToGlobal(point);
  // Измерить расстояние от точки регистрации этого клипа до мыши
  deltaX = _root._xmouse - point.x;
  deltaY = _root._ymouse - point.y;
  // Вычислить угол линии от точки регистрации этого клипа до мыши
  rotationRadian = Math.atan2(deltaY, deltaX);
  // Преобразовать радианную меру угла в градусы
  rotationAngle = radiansToDegrees(rotationRadian); // См. функцию выше
  // Обновить угол поворота этого клипа, чтобы он указывал на мышь
  this._rotation = rotationAngle;
}
```

**Math.ceil( ) метод** округлить число до следующего целого

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.ceil(x)`

***Аргументы***

*x* Число.

***Возвращаемое значение***

Следующее целое число, большее или равное *x*.

***Описание***

Метод *ceil( )* (т. е. «потолок») преобразует число с плавающей точкой в первое целое число, которое больше или равно *x*.

***Пример***

```
Math.ceil(1.00001);  // Возвращает 2
Math.ceil(5.5);      // Возвращает 6
Math.ceil(-5.5);     // Возвращает -5
```

***См. также***

*Math.floor( )*, *Math.round( )*.

---

**Math.cos( ) метод** вычислить косинус угла

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.cos(`*theta*`)`

***Аргументы***

*theta* Угол в радианах (не в градусах), в диапазоне от 0 до 2π.

***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 катета, прилегающего к углу, на длину гипотенузы треугольника.

***Особенности применения***

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 использовать вместе с функцией *sin( )* для вычисления координат точки на окружности, что используется в следующем примере для перемещения клипа в круговом направлении. Если есть радиус окружности *r* и угол θ, измеренный против часовой стрелки от положительной горизонтальной оси, то точка имеет координаты (r*cosθ, r*sinθ):

```
// КОД В КАДРЕ 1
```

```
var radius = 100;      // Радиус кругового пути
var centerX = 275;     // Положение центра кругового пути по горизонтали
var centerY = 200;     // Положение центра кругового пути по вертикали
var rotAngleDeg = 0;   // Угол объекта в градусах, измеренный против
                       // часовой стрелки от горизонтали (оси X)
var rotAngRad;         // rotAngleDeg в радианах

// Преобразовать градусы в радианы. В 360 градусах 2*pi радиан.
function degreesToRadians(degrees) {
  return (degrees/180) * Math.PI;
}

// КОД В КАДРЕ 2
// Увеличить угол поворота на 5 градусов
rotAngleDeg += 5;

// Расположить объект. Обратите внимание, что Flash инвертирует ось Y
// декартовых координат, поэтому мы увеличиваем y для получения
// нового местоположения
rotAngRad= degreesToRadians(rotAngleDeg);
ball._x = centerX + Math.cos(rotAngRad) * radius;
ball._y = centerY - Math.sin(rotAngRad) * radius;

// КОД В КАДРЕ 3
// Вернуться в кадр 2, где шарик будет снова перемещен
gotoAndPlay(2);
```

***См. также***

*Math.acos( ), Math.sin( ), Math.tan( ).*

---

**Math.E свойство** константа *e* (основание натуральных логарифмов)

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.E`

***Описание***

Свойство `E` хранит приближенное значение основания натуральных логарифмов (примерно 2.71828), которое в математике обозначается символом *e*. Это трансцендентное, как и π, число, используется в математических уравнениях, в том числе в уравнениях роста и уравнениях изменения. Не путайте его с `E`, используемой в экспоненциальных обозначениях, описываемых в разделе «Литералы с плавающей точкой» в главе 4. Они не связаны между собой.

***См. также***

*Math.log( ), Math.LN10( ), Math.LN2( ).*

---

**Math.exp( ) метод** возвести константу *e* в заданную степень

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.exp(x)`

***Аргументы***

***x*** Степень, в которую нужно возвести `Math.E`.

***Возвращаемое значение***

`Math.E` в степени *x*.

---

**Math.floor( ) метод** округлить число вниз до предыдущего целого

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `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, которое меньше или равно x.

***Пример***

```
Math.floor(1.99999);  // Возвращает 1
Math.floor(5.5);      // Возвращает 5
Math.floor(-5.5);     // Возвращает -6

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** натуральный логарифм 10 ($\log_e 10$), приблизительно 2.30259

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.LN10`

***Описание***

Свойство `LN10` представляет натуральный логарифм 10 (логарифм 10 по основанию *e*) – константу, приблизительно равную 2.30259.

**Math.LN2 свойство** натуральный логарифм 2 ($\log_e 2$), приблизительно 0.69315

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.LN2`

***Описание***

Свойство `LN2` представляет натуральный логарифм 2 (логарифм 2 по основанию *e*) — константу, приблизительно равную 0.69315

---

**Math.log( ) метод** вычислить натуральный логарифм числа

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.log(x)`

***Аргументы***

***x*** Положительное число.

***Возвращаемое значение***

Натуральный логарифм *x*.

***Описание***

Метод *log( )* вычисляет натуральный логарифм числа (т. е. логарифм по основанию *e*). В следующем примере показано вычисление 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** логарифм *e* по основанию 10, приблизительно 0.43429

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.LOG10E`

***Описание***

Свойство `LOG10E` представляет десятичный логарифм *e* (логарифм *e* по основанию 10) – константу, приблизительно равную 0.43429.

**Math.LOG2E свойство** логарифм *e* по основанию 2, приблизительно 1.44270

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.LOG2E`

***Описание***

Свойство `LOG2E` представляет логарифм *e* по основанию 2 ($\log_2 e$) – константу, приблизительно равную 1.44270.

***Ошибки в реализации***

Во Flash 5r30 `LOG2E` ошибочно возвращает значение `LN2` (0.69315).

---

**Math.max( ) метод** определить большее из двух чисел

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.max(x, y)`

***Аргументы***

*x* Число.

*y* Число.

***Возвращаемое значение***

Наибольшее число из *x* и *y*.

***Пример***

```
Math.max(5, 1);     // Возвращает 5
Math.max(-6, -5);   // Возвращает  -5
```

В этом примере значение ограничивается заданным диапазоном :

```
function constrainToRange (checkVal, minVal, maxVal) {
  return Math.min(Math.max(checkVal, minVal), maxVal);
}
// Ограничить ползунок областью рабочего стола
mySlider._x = constainToRange (mySlider._x, 0, 550);
```

В следующем 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.min( ).*

---

**Math.min( ) метод** определить меньшее из двух чисел

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.min(x, y)`

***Аргументы***

*x* Число.

*y* Число.

***Возвращаемое значение***

Меньшее из *x* и *y*.

***Пример***

```
Math.min(5, 1);    // Возвращает 1
Math.min(-6, -5);  // Возвращает -6
```

Упражнение для читателя: изменить пример из *Math.max( )*, чтобы возвращалось минимальное, а не максимальное значение в массиве.

***См. также***

*Math.max( ).*

---

**Math.PI свойство** отношение длины окружности к ее диаметру, приблизительно 3.14159

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.PI`

***Описание***

Свойство представляет константу π – отношение длины окружности к ее диаметру.

***Пример***

`Math.PI` чаще всего используется при вычислении 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);
}
```

---

**Math.pow( ) метод** возвести число в указнную степень

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.pow(base, exponent)`

***Аргументы***

| | |
|---|---|
| *base* | Число, представляющее основание степенного выражения. |
| *exponent* | Число, представляющее степень (т. е. показатель степени), в которую нужно возвести *base*. |

***Возвращаемое значение***

*base* в степени *exponent*.

***Описание***

Метод *pow( )* можно использовать для возведения любого числа в любую степень. Если *exponent* отрицателен, *pow( )* возвращает 1 / (*base*$^{abs(exponent)}$). Если *exponent* является дробным числом, *pow( )* можно использовать для вычисления квадратного или кубического корня из числа (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 в реализации***

В Build 30 Flash 5 Player неправильно производился расчет *Math.pow( )* для отрицательных значений *base*. Например, `Math.pow(-2, 2)` давало результат `NaN`, хотя должно быть 4.

***См. также***

*Math.exp( )*, *Math.sqrt( )*, `Math.SQRT2`, `Math.SQRT1_2`.

---

**Math.random( ) метод** генерировать случайное число между 0 и 1.0

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `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 между 0 и .99999... включительно, которое можно масштабировать применительно к потребностям. Например, для получения случайного числа между 0 и 5 используется:

```
Math.floor(Math.random() * 6)
```

А для получения случайного числа между 1 и 6 используется:

```
Math.floor(Math.random() * 6) + 1
```

Следующая пользовательская функция возвращает случайное целое число в заданном диапазоне, а не число с плавающей точкой между 0 и 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 в Build 30 Flash 5 Player такой подход чреват неточностями – хотя и редкими, но способными стать важными. В Build 30 *random( )* генерирует значения в диапазоне от 0.0 до 1.0 включительно. При умножении значения, возвращаемого *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). Это ведет к неправильному распределению случайных чисел: вероятность получить *n* значительно меньше, чем вероятность получить любое другое число из этого ряда.

В следующей версии функции *myRandom( )* проблема решается путем простого 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));
}

// Вызвать функцию
dieRoll = myRandom(1, 6);  // Безопасное моделирование шестигранной
                           // кости в Build 30 Flash 5 Player
```

***Особенности применения***

*Math.random( )* заменяет устаревшую функцию Flash 4 *random*.

***Пример***

*Math.random( )* часто используется, чтобы заставить воспроизводящую головку скачком переместиться в случайный кадр на временной диаграмме. В следующем коде вызывается функция *myRandom( )* из предыдущего примера, после чего воспроизводящая головка направляется в случайно выбранный кадр:

```
// Вызвать функцию; выбрать случайное число между 10 и 20
var destinationFrame = myRandom(10, 20);

// Направить воспроизводящую головку в этот кадр
gotoAndStop(destinationFrame);
```

---

**Math.round( ) метод** вычислить ближайшее целое к числу

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.round(x)`

***Аргументы***

***x*** Число.

***Возвращаемое значение***

Целое число, математически наиболее близкое к *x* (или сам *x*, если *x* является целым). Если дробная часть *x* равна в точности 0.5 (*x* находится на равном расстоянии от двух ближайших целых), *round( )* возвращает первое целое число, 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 в нижнюю сторону. Положительные числа с дробной частью, *большей* или равной 0.5, и отрицательные числа с дробной частью, *меньшей* или равной 0.5, округляются в большую сторону.

***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** вычислить синус угла

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.sin(`*theta*`)`

***Аргументы***

*theta* Угол в радианах (а не в градусах), в диапазоне от 0 до 2π.

***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 катета, противолежащего углу, на гипотенузу треугольника.

***Особенности применения***

Обратите внимание, что *sin( )*предполагает задание угла в радианах, а не в градусах.

***Пример***

```
trace (Math.sin(0));            // Выводит: 0
trace (Math.sin(Math.PI/2));    // Выводит: 1.0
```

Функцию *sin( )*можно использовать вместе с функцией *cos( )* для вычисления координат точки на окружности. См. пример в описании *Math.cos( )*.

***См. также***

*Math.asin( )*, *Math.cos( )*, *Math.tan( )*.

---

**Math.sqrt( ) метод** вычислить квадратный корень числа

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.sqrt(`*x*`)`

***Аргументы***

*x* Неотрицательное целое число.

***Возвращаемое значение***

Квадратный корень из *x* или `NaN`, если *x* меньше 0.

***Описание***

Метод *sqrt( )* возвращает положительный корень из своего операнда (хотя может быть математически верный отрицательный корень). Он эквивалентен следующему:

```
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
```

***См. также***

*Math.pow( )*, `Math.SQRT2`, `Math.SQRT1_2`.

---

**Math.SQRT1_2 свойство** величина, обратная квадратному корню из 2, приблизительно 0.70711

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.SQRT1_2`

***Описание***

Свойство `SQRT1_2` является константой, аппроксимирующей значение `1/Math.SQRT2` (величина, обратная квадратному корню из 2). Эта константа равна приблизительно 0.70711.

***См. также***

`Math.SQRT2`.

---

**Math.SQRT2 свойство** квадратный корень из 2, приблизительно 1.41421

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.SQRT2`

***Описание***

Свойство `SQRT2` является константой, аппроксимирующей квадратный корень из 2. Это иррациональное число, равное приблизительно 1.41421.

***См. также***

`Math.SQRT1_2`.

---

**Math.tan( ) метод** вычислить тангенс угла

***Доступность*** Flash 5; может использоваться при экспорте фильмов Flash 4

***Краткий обзор*** `Math.tan(`*theta*`)`

***Аргументы***

*theta* Угол в радианах (не в градусах), в диапазоне от $-\pi/2$ до $\pi/2$.

***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 уг-

лу. Это то же самое, что отношение *Math.sin(theta)*/*Math.cos(theta)*, поэтому когда *cos(theta)* приближается к нулю, *tan(theta)* приближается к `Infinity`. Из-за этого *tan(theta)* не может быть вычислен для значений аргумента $-\pi/2$, $\pi/2$, $-3\pi/2$, $3\pi/2$, и так далее.

***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** последняя допустимая строка текстового поля

***Доступность*** Flash 4 и выше

***Краткий обзор*** *textField*.maxscroll

***Возвращаемое значение***

Положительное целое число, 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.

Свойство `maxscroll` можно использовать вместе со свойством `scroll` для управления полем с прокруткой текста.

***См. также***

`scroll`; раздел «Свойство maxscroll» в главе 18.

---

**Mouse объект** скрыть или показать указатель мыши

***Доступность*** Flash 5

***Краткий обзор*** Mouse.*methodName*

***Методы***

| | |
|---|---|
| *hide( )* | Скрывает курсор мыши. |
| *show( )* | Включает курсор мыши. |

***Описание***

У объекта *Mouse* есть только два метода и ни одного свойства. Метод *Mouse.hide( )* используется, чтобы скрыть системный курсор мыши обычно для того, чтобы заменить его пользовательским курсором, как будет показано ниже. Желательно бывает также спрятать курсор мыши в полноэкранном режиме, в фильмах, управляемых с клавиатуры и тактильных киосках (touch-screen kiosks).

Обратите внимание, что объект *Mouse* не сообщает о местонахождении курсора мыши. Эта информация хранится в свойствах `_xmouse` и `_ymouse`

каждого объекта клипа. Для определения местоположения курсора мыши на главном рабочем столе используйте `_root._xmouse` и `_root._ymouse`.

***См. также***

*MovieClip*.`_xmouse`, *MovieClip*.`_ymouse`.

---

**Mouse.hide( ) метод** скрыть курсор мыши

***Доступность*** Flash 5

***Краткий обзор*** `Mouse.hide()`

***Описание***

Метод *hide( )* вызывает исчезновение нормального курсора мыши (обычно стрелки) при нахождении над любой частью проигрывателя. Нормальный системный курсор вновь появляется, когда мышь выходит за пределы активной области Flash Player.

***Особенности применения***

Обратите внимание, что во Flash 5 даже после вызова *Mouse.hide( )*при прохождении мыши над текстовым полем появляется обычный системный текстовый курсор длиной с букву I.

***Пример***

Метод *hide( )* применяется, чтобы скрыть системный курсор мыши, используемый по умолчанию, – обычно для того, чтобы заменить его пользовательским курсором. Во Flash пользовательский курсор является просто клипом, перемещающимся вслед за мышью. С помощью события *mouseMove* можно вызвать перемещение курсора вслед за мышью, обновляя свойства клипа `_x` и `_y` в каждом показываемом кадре. Этот прием демонстрируется в следующем коде:

```
// Код в клипе, выступающем в качестве пользовательского курсора
onClipEvent (load) {
  Mouse.hide();
}

onClipEvent (mouseMove) {
  _x = _root._xmouse;
  _y = _root._ymouse;
  updateAfterEvent();
}
```

Бывает также желательно скрыть пользовательский курсор, когда мышь не действует – например, если курсор вышел за пределы активной области Flash Player, потому что тогда появляется системный курсор, и на экране оказываются два курсора. Следующий код показывает, как изменить предыдущий код, чтобы скрыть курсор после 5 секунд бездействия:

```
onClipEvent (load) {
  Mouse.hide();
}

onClipEvent (enterFrame) {
```

```
  if (getTimer() - lastMove > 5000) {
    _visible = false;
  } else {
    _visible = true;
  }
}

onClipEvent (mouseMove) {
  _x = _root._xmouse;
  _y = _root._ymouse;
  lastMove = getTimer();
  updateAfterEvent();
}
```

Чтобы заменить курсор на ярлычок «рука», когда мышь находится над кнопкой, используйте событие кнопки *rollOver* для присваивания клипу пользовательского указателя кадра, содержащего пользовательское изображение руки, и событие *rollOut*, чтобы снова присвоить пользовательскому указателю значение по умолчанию:

```
on (rollOver) {
  _root.customPointer.gotoAndStop("hand");
}

on (rollOut) {
  _root.customPointer.gotoAndStop("default");
}
```

Во всех случаях не забывайте поместить пользовательский курсор в верхний слой фильма, чтобы он показывался поверх любого другого содержимого. Другой возможностью является динамическая генерация клипа пользовательского курсора с помощью *duplicateMovieClip( )* или *attachMovie( )* с присвоением ему очень высокого уровня.

***См. также***

*Mouse.show( )*, *MovieClip._xmouse*, *MovieClip._ymouse*.

---

**Mouse.show( ) метод** сделать курсор мыши видимым

***Доступность*** Flash 5

***Краткий обзор*** `Mouse.show()`

***Описание***

Метод *show( )* вызывает возвращение системного курсора мыши по умолчанию, спрятанного при вызове *Mouse.hide( )*. Его можно использовать для вывода обычного курсора в некоторые моменты фильма – например, когда пользователь должен заполнить форму. Это один из способов справиться с появлением текстового курсора в форме буквы I, когда системный курсор спрятан.

***См. также***

*Mouse.hide( )*.

**MovieClip класс** похожий на класс тип данных для главных фильмов и клипов главных фильмов

***Доступность*** Flash 3 и выше

***Constructor***

Отсутствует. Символы клипов создаются вручную в редакторе Flash. Экземпляры клипов можно создавать с помощью *attachMovie( )* и *duplicateMovieClip( )*.

***Свойства***

Свойства клипов описывают и контролируют различные физические характеристики клипов и главных фильмов. Доступ к свойствам производится посредством оператора точки, как и для любых других объектов. О подробностях использования свойств клипов читайте в разделе ««Объектность» клипов» главы 13.

Следует отметить две проблемы, связанные со свойствами:

- Если какое-либо физическое свойство объекта *MovieClip* изменяется, то он начинает управляться программно, разрывая связь со своей внутренней временной диаграммой. В результате останавливается любая трансформация, происходящая 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 между 0 и 255. Это приводит к потере точности при задании и чтении некоторых свойств. Можно, например, задать 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.

В табл. 9 перечислены свойства клипа. Обратите внимание на то, что все внутренние свойства клипов начинаются с символа подчеркивания. Благодаря этому их легко отличить от пользовательских свойств, которые могут быть добавлены в клип, и которые принято не начинать с подчеркивания. В колонке *Доступ* табл. 9 *R/W* указывает на возможность как извлечения, так и задания свойства (т. е. чтения/записи), а *RO* указывает, что его можно извлекать, но не задавать (т. е. использовать только для чтения). Некоторые свойства только для чтения можно косвенным образом задавать в редакторе или некоторой связанной функции (например, *gotoAndStop( )* задает свойство `_currentframe`), но непосредственно через ActionScript можно задавать только свойства, доступные для чтения и записи. В колонке *Тип* описан тип данных значений каждого свойства. В колонке *Описание свойства* дана краткая сводка назначения свойства, но в последующих полных описаниях даются важные детали. За исключением свойств `_name` и `_parent`, свойства в табл. 9 применимы как к экземплярам клипов, так и к главному фильму (т. е. `_root`). Однако значение свойства может быть различным в зависимости от того, получено ли оно из экземпляра фильма или из главного фильма, и может также различаться в

зависимости от того, где прикреплен клип. Например, свойство _x клипа различно в зависимости от того, прикреплен ли он к главной временной диаграмме или как дочерний к родительскому клипу. Кроме того, ко всем свойствам текущей временной диаграммы можно обращаться, не ссылаясь на нее явно, т. е. писать _alpha вместо myClip._alpha.

*Таблица 9. Сводка свойств клипов*

| Имя свойства | Доступ | Тип | Описание свойства |
|---|---|---|---|
| _alpha | R/W | number | Процент непрозрачности: 0 – прозрачный; 100 – непрозрачный |
| _currentframe | RO | number | Номер кадра, в котором находится воспроизводящая головка |
| _droptarget | RO | string | Путь к клипу, над которым находится или куда сброшен перетаскиваемый клип, в слэш-нотации |
| _framesloaded | RO | number | Количество загруженных кадров |
| _height | R/W | number | Высота текущего содержимого в пикселах |
| _name* | R/W | string | Идентификатор клипа в виде строки (не ссылки) |
| _parent* | RO | ссылка на MovieClip | Ссылка на экземпляр или фильм, содержащий текущий экземпляр |
| _rotation | R/W | number | Угол поворота в градусах |
| _target | RO | string | Абсолютный путь к цели в слэш-нотации |
| _totalframes | RO | number | Количество кадров во временной диаграмме |
| _url | RO | string | Адрес исходного файла *.swf* на диске или в сети |
| _visible | R/W | boolean | Видимость: true, если показывается; false, если скрыт |
| _width | R/W | number | Ширина текущего содержимого в пикселах |
| _x | R/W | number | Позиция по горизонтали в пикселах |
| _xmouse | RO | number | Позиция курсора мыши по горизонтали в пикселах |
| _xscale | R/W | number | Масштаб по горизонтали в процентах |
| _y | R/W | number | Позиция по вертикали в пикселах |
| _ymouse | RO | number | Позиция курсора мыши по вертикали в пикселах |
| _yscale | R/W | number | Масштаб по вертикали в процентах |

* Применимо к экземплярам клипов; неприменимо к главной временной диаграмме.

### *Методы*

Методы клипов могут вызываться с любым экземпляром клипа и в большинстве случаев с главной временной диаграммой. Многие методы клипов выполняют такие же действия, как аналогичные глобальные функции, но используют более удобный формат синтаксиса с точками *MovieClip.method()*. Те методы, которым не соответствуют глобальные функции, могут применяться к текущему клипу без явной ссылки на него – например, `attachMovie()` вместо `myClip.attachMovie()`. В табл. 10 перечислены методы клипов.

*Таблица 10. Сводка методов клипов*

| Имя метода | Описание метода |
|---|---|
| *attachMovie( )* | Создает новый экземпляр на основе символа, экспортированного из библиотеки текущего документа. Помещает новый экземпляр в другой клип или стек программно генерированных клипов фильма. |
| *duplicateMovieClip( )** | Создает копию текущего экземпляра и помещает ее в стек программно генерированных клипов (см. раздел «Добавление в стек клипов, созданных с помощью duplicateMovieClip()» в главе 13). |
| *getBounds( )* | Возвращает объект, свойствами которого являются координаты рамки, ограничивающей занимаемую клипом видимую область. |
| *getBytesLoaded( )* | Возвращает количество загруженных байтов экземпляра или фильма. Не может применяться к внутренним клипам. |
| *getBytesTotal( )* | Возвращает физический размер в байтах экземпляра или фильма. |
| *getURL( )* | Загружает в броузер документ (обычно веб-страницу). |
| *globalToLocal( )* | Преобразует свойства объекта *coordinates* из координат рабочего стола в координаты экземпляра. Бесполезен при вызове с объектами на главной временной диаграмме (например, `_root`), потому что пространства координат источника и приемника совпадают. |
| *gotoAndPlay( )* | Перемещает воспроизводящую головку экземпляра или фильма в заданный кадр, затем воспроизводит экземпляр или фильм. |
| *gotoAndStop( )* | Перемещает воспроизводящую головку экземпляра или фильма в заданный кадр и останавливает ее. |
| *hitTest( )* | Возвращает булево значение, показывающее, пересекается ли клип с заданной точкой или с другим клипом. |
| *loadMovie( )* | Загружает в проигрыватель внешний файл *.swf*. |
| *loadVariables( )* | Возвращает временные данные, состоящие из имен переменных и их значений, и преобразует эти данные в эквивалентные переменные ActionScript. |

*Таблица 10 (продолжение)*

| Имя метода | Описание метода |
|---|---|
| *localToGlobal( )* | Преобразует свойства объекта *coordinates* из координат объекта в координаты главного фильма. |
| *nextFrame( )* | Перемещает воспроизводящую головку фильма или экземпляра на один кадр вперед и останавливает ее там. |
| *play( )* | Приводит в движение воспроизводящую головку экземпляра или фильма (т. е. воспроизводит клип). |
| *prevFrame( )* | Перемещает воспроизводящую головку фильма или экземпляра на один кадр назад и останавливает ее там. |
| *removeMovieClip( )** | Удаляет дублированный или прикрепленный экземпляр. |
| *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 экземпляров. |
| *unloadMovie( )* | Удаляет экземпляр или главный фильм с уровня документа или главного клипа. |
| *valueOf( )* | Представляет путь к экземпляру в абсолютных терминах в точечной записи. |

* Применимо к экземплярам клипов; неприменимо к главной временной диаграмме.

### *События*

Экземпляры клипов поддерживают *обработчики событий*, которые автоматически реагируют на определенный набор событий (таких как действия мыши или клавиатуры, либо воспроизведение фильма). Поддерживаемые обработчики событий клипов перечислены в табл. 11. Подробно о каждом из них читайте в главе 10 «События и обработчики событий».

*Таблица 11. Сводка обработчиков событий клипов*

| Обработчик события клипа | Событие происходит, когда... |
|---|---|
| *onClipEvent (enterFrame)* | Переменяется кадр во Flash Player |
| *onClipEvent (load)* | Клип впервые появляется на рабочем столе |
| *onClipEvent (unload)* | Клип удаляется с рабочего стола |
| *onClipEvent (data)* | Заканчивается поток загружаемых переменных или порции загружаемого фильма |
| *onClipEvent (mouseDown)* | Главная кнопка мыши нажата во время нахождения клипа на рабочем столе |
| *onClipEvent (mouseUp)* | Главная кнопка мыши нажата и отпущена во время нахождения клипа на рабочем столе |

*Таблица 11 (продолжение)*

| Обработчик события клипа | Событие происходит, когда... |
|---|---|
| *onClipEvent (mouseMove)* | Курсор мыши перемещается (хоть самую малость) во время нахождения клипа на рабочем столе |
| *onClipEvent (keyDown)* | Нажата клавиша во время нахождения клипа на рабочем столе |
| *onClipEvent (keyUp)* | Нажатая клавиша отпущена во время нахождения клипа на рабочем столе |

***Описание***

*MovieClip* в ActionScript фактически является не классом, а особым типом данных, дающим возможность представлять 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, а можно создавать методы клипов и обращаться к ним.

Так как *MovieClip* подлинным классом не является, для создания новых экземпляров *MovieClip* не используется конструктор. Вместо этого символы клипов создаются в среде разработки, и экземпляры вручную помещаются на рабочий стол. Однако есть методы, позволяющие копировать существующие клипы (*duplicateMovieClip( )*) или программно помещать новые клипы на рабочий стол (*attachMovie( )*).

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 на использование слова *MovieClip* для обозначения «класса» объектов и слова *movieclip* (строчными буквами) для обозначения типа данных ActionScript; *клип фильма*, *клип* или *экземпляр* обозначают конкретный клип фильма, а слово *фильм* используется 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` является необязательным: если оно опущено, используется текущая временная диаграмма.

В данном разделе *MovieClip*, когда идет речь о координатах, необходимо ссылаться на местоположение клипов фильмов. Положение клипа измеряется относительно одной характерной точки, так называемой *точки регистрации*, отмеченной перекрестием в библиотечном символе клипа.

Когда клип расположен на рабочем столе, его позиция описывается относительно левого верхнего угла рабочего стола, соответствующего точке (0,0). Когда клип находится в другом клипе, его позиция описывается относительно точки регистрации родительского клипа, которой опять-таки соответствует точка (0,0). В обоих случаях точка (0,0) является началом (*origin*) системы координат, используемой для нанесения положения клипа. Как будет показано, с помощью методов *localToGlobal( )* и *globalToLocal( )* можно осуществлять преобразование между этими двумя системами координат.

В системе координат Flash ось Y декартовых координат перевернута, т. е. значения у возрастают вниз, а не вверх. Например, у-координата 100 указывает на точку, находящуюся на 100 пикселов *ниже* оси X.

Свойства и методы, связанные с координатами, часто будут использоваться 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 его часто используют для определения того, не нужно 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* (т. е. x- и y- координат). См. в описании *MovieClip.globalToLocal( )* объяснение создания объекта точки из общего объекта.

***См. также***

Полное рассмотрение класса *MovieClip* в главе 13.

**MovieClip._alpha свойство** степень непрозрачности клипа или фильма

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._alpha

***Тип доступа*** Чтение/запись

***Описание***

Свойство _alpha, выражаемое числом с плавающей точкой, задает степень непрозрачности (или наоборот, прозрачности) *mc* в виде процента: 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 свойств _alpha. Это значит, например, что если есть клип square, содержащий другой клип, circle, и square._alpha присваивается 50, то circle будет прозрачен на 50%, но его _alpha равно 100.

***Особенности применения***

Обратите внимание, что задание свойства _alpha для клипа фильма оказывает влияние на свойство объекта aa, возвращаемое *Color.getTransform( )*.

***Пример***

```
ball._alpha = 60;   // Сделать ball частично прозрачным
ball._alpha = 0;    // Сделать ball невидимым
```

Следующий обработчик события клипа увеличивает прозрачность клипа по мере перемещения мыши вниз по экрану:

```
onClipEvent(enterFrame) {
  _alpha = 100 - (_root._ymouse / 400) * 100;
}
```

***См. также***

*Color* класс; *MovieClip._visible*.

---

**MovieClip.attachMovie( ) метод** — создать новый экземпляр клипа из экспортированного символа библиотеки

***Доступность*** Flash 5

***Краткий обзор*** *mc*.attachMovie(*symbolIdentifier*, *newName*, *depth*)

***Аргументы***

*symbolIdentifier* Строка, задающая идентификатор связи экспортированного символа клипа, как он задан в библиотеке в меню Options → Linkage.

*newName* Строка, задающая имя нового экземпляра создаваемого клипа. Имя должно удовлетворять правилам построения идентификаторов, описанным в разделе «Идентификаторы» главы 14 «Лексическая структура».

*depth* Целое число, задающее уровень, на который должен быть помещен новый клип в программном стеке клипов над *mc*. Уровень –1 находится под уровнем 0, 1 впереди 0, уровень 2 – впереди 1, и так далее. Более подробное описание см. в разделе «Добавление в стек клипов, созданных с помощью attachMovie()» главы 13. Отрицательные уровни действуют, но официально не поддерживаются в ActionScript – чтобы гарантировать совместимость с будущими версиями, используйте уровни 0 и выше.

***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 рабочего стола. Если *mc* является экземпляром клипа, новый экземпляр помещается в точку регистрации *mc*. В любом случае новый экземпляр располагается над *mc* в стеке программно генерируемых клипов.

***См. также***

*duplicateMovieClip( )*, *MovieClip.duplicateMovieClip( )*; разделы «Создание экземпляров с помощью attachMovie()» и «Добавление в стек клипов, созданных с помощью duplicateMovieClip()» в главе 13.

---

**MovieClip._currentframe свойство** — номер кадра, в котором находится воспроизводящая головка клипа или фильма

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._currentframe

***Тип доступа*** Только для чтения

***Описание***

Целочисленное свойство `_currentframe` представляет номер кадра, в котором в данное время находится воспроизводящая головка *mc*. Заметьте, что первый кадр имеет номер 1, а не 0, поэтому `_currentframe` находится в диапазоне от `1` до *mc*.`_totalframes`.

***Пример***

```
// Вернуть воспроизводящую головку на 2 кадра назад от текущего положения
gotoAndStop(_currentframe - 2);
```

***См. также***

*MovieClip.gotoAndPlay( )*, *MovieClip.gotoAndStop( )*.

---

**MovieClip._droptarget свойство** путь к клипу или фильму, на который опущен перетаскиваемый клип

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*.`_droptarget`

***Тип доступа*** Только для чтения

***Описание***

Когда *mc* перетаскивается, `_droptarget` хранит строку, указывающую путь к клипу, над которым находится *mc* (если такой есть). Если *mc* не находится над клипом, в `_droptarget` хранится значение `undefined`. Если *mc* 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.

***Пример***

Свойство `_droptarget` удобно использовать при создании интерфейсов типа drag-and-drop. В следующем примере показано, как создать простой интерфейс корзины покупок с перетаскиванием при помощи `_droptarget` (когда клип `item` бросается на клип `basket`, клипу `item` разрешается остаться в `basket`; в противном случае, `item` возвращается в исходное положение):

```
// КОД В КАДРЕ 1 item
var origX = _x;
var origY = _y;

function drag() {
  this.startDrag();
}

function drop() {
  stopDrag();
  if (_droptarget != "/basket") {
    _x = origX;
```

```
      _y = origY;
    }
  }

  // КОД В КНОПКЕ item
  on (press) {
    drag();
  }

  on (release) {
    drop();
  }
```

Обратите внимание, что в `_droptarget` хранится строка, а не ссылка на клип. Для преобразования строки `_droptarget` в ссылку на клип используется прием, показанный в примере для *eval( )*.

***См. также***

*MovieClip.startDrag( )*, *MovieClip.stopDrag( )*.

---

**MovieClip.duplicateMovieClip( ) метод** создать копию клипа

***Доступность*** Формат метода появился во Flash 5 (формат глобальной функции появился во Flash 4)

***Краткий обзор*** `mc.duplicateMovieClip(newName, depth)`

***Аргументы***

*newName* Строка, которая станет именем экземпляра продублированного клипа. Она должна удовлетворять правилам создания идентификаторов, приведенным в разделе «Идентификаторы» главы 14 «Лексическая структура».

*depth* Целое число, указывающее уровень, на который должен быть помещен новый клип в стеке программно создаваемых клипов над `mc`. Уровень –1 находится под 0, 1 впереди 0, 2 впереди 1, и так далее. Дополнительные сведения есть в главе 13. Отрицательные значения уровней действуют, но официально не поддерживаются ActionScript – для гарантии совместимости с будущими версиями используйте уровни 0 и выше.

***Описание***

Метод *MovieClip.duplicateMovieClip( )* представляет собой альтернативу глобальной функции *duplicateMovieClip( )*. При вызове в качестве метода *MovieClip duplicateMovieClip( )* не принимает параметра `target` – он дублирует `mc`. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции.

Рекомендации по использованию см. в описании глобальной функции *duplicateMovieClip( )*.

***См. также***

*MovieClip.attachMovie( )*, *duplicateMovieClip( )*.

**MovieClip._framesloaded свойство** количество кадров клипа или фильма, загруженных в проигрыватель

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._framesloaded

***Тип доступа*** Только для чтения

***Описание***

Целочисленное свойство _framesloaded указывает количество кадров *mc*, загруженных во Flash Player (от 0 до *mc*._totalframes). Обычно используется при создании предварительных загрузчиков, приостанавливающих воспроизведение до загрузки достаточного числа кадров. Для клипа фильма свойство _framesloaded всегда равно _totalframes (потому что клипы загружаются перед воспроизведением целиком), 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.

Код предварительного загрузчика обычно помещается непосредственно на главную временную диаграмму предварительно загружаемого фильма. Простым подходом является циркуляция между кадрами 1 и 2, пока не будет загружен фильм, после чего осуществляется переход в начальный кадр фильма. Например:

```
//КОД В КАДРЕ 1
if (_framesloaded > 0 && _framesloaded == _totalframes) {
  gotoAndPlay("beginMovie");
}

// КОД В КАДРЕ 2
gotoAndPlay(1);
```

Во Flash 5 и выше можно построить лучше переносимый предварительный загрузчик с использованием обработчика события клипа *enterFrame*. В фильме, который желательно предварительно загрузить, в том его кадре, где начинается предварительная загрузка, вызывается функция *stop( )*. Затем на временную диаграмму этого фильма помещается клип со следующим кодом:

```
onClipEvent (enterFrame) {
  loaded = _parent._framesloaded;
  if (loaded > 0 && loaded == _parent._totalframes && loading != "done") {
    _parent.gotoAndPlay("beginMovie");
    loading = "done";
  }
}
```

В этом примере клип следит за ходом процесса загрузки своего родителя и запускает воспроизведение родителя, когда загрузка закончена. Благодаря использованию клипа в качестве предварительного загрузчика удается обойтись без цикла на временной диаграмме фильма с предварительной

загрузкой. Предварительный загрузчик клипа можно даже превратить в интеллектуальный клип, что облегчило бы жизнь менее опытным разработчикам.

Обратите внимание, что в наших примерах предварительных загрузчиков помимо проверки `_framesloaded == _totalframes` проверялось также условие `_framesloaded > 0`. Это необходимо, поскольку после выгрузки фильма из клипа `_totalframes` этого клипа становится равным 0. Поэтому, если `_framesloaded` равно 0 (что может произойти при очень низкоскоростном соединении), то сравнение `_framesloaded == _totalframes` может возвратить `true`, даже если ни один кадр еще не загружен. Наша проверка не дает фильму перескочить вперед, пока не будет загружено соответствующее содержимое. В такой предосторожности нет необходимости при загрузке предварительных загрузчиков главных фильмов для файлов *.swf* на уровни, поскольку их свойство `_totalframes` никогда не равно нулю.

### *Пример*

В предварительных загрузчиках часто есть 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 обратите внимание, что масштабирование клипа происходит вокруг его точки регистрации (пропорционально в левую и правую стороны от точки регистрации клипа). Поэтому для изменения размера клипа только в одну сторону нужно поместить все его содержимое по одну сторону от точки регистрации в символе клипа. В следующем примере показано, как добавить индикатор загрузки и текстовое поле состояния в прежний код обработчика клипа:

```
// Переносимый предварительный загрузчик с индикатором состояния
onClipEvent (enterFrame) {
  // Определить, сколько кадров загружено
  loaded = _parent._framesloaded;
  // Если все кадры загружены...
  if (loaded > 0 && loaded == _parent._totalframes && loading != "done") {
    // ...воспроизвести фильм и отметить, что загрузка завершена
    _parent.gotoAndPlay("beginMovie");
    loading = "done";
  }
  // Определить процент загруженных байтов от их общего числа
  percentDone = Math.floor((_parent.getBytesLoaded()
                          / _parent.getBytesTotal()) * 100);
  // Вывести процент загруженных байтов в текстовом поле loadStatus
  loadStatus =  percentDone + "% complete";
  // Задать размер клипа loadBar
  loadBar._xscale = percentDone;
}
```

Для моделирования загрузки фильма с целью тестирования предварительного загрузчика используйте Bandwidth Profiler в режиме Test Movie.

***См. также***

*MovieClip.getBytesLoaded( )*, *MovieClip._totalframes*; разделы «data» в главе 10 и «Профайлер Bandwidth Profiler» в главе 19.

---

**MovieClip.getBounds( ) метод** определить рамку, ограничивающую клип или фильм

***Доступность*** Flash 5

***Краткий обзор*** *mc*.getBounds(*targetCoordinateSpace*)

***Аргументы***

*targetCoordinateSpace*

Строка, указывающая фильм или клип, в пространстве которого измеряются размеры *mc*. Поскольку ссылка на клип преобразуется в путь при использовании в строковом контексте, *targetCoordinateSpace* может быть и ссылкой на объект клипа, например, mc.getBounds(_root) вместо mc.getBounds("_root"). Если она не задана, то по умолчанию принимается равной *mc*.

***Возвращаемое значение***

Объект, свойства которого – xMin, xMax, yMin, yMax – описывают рамку, ограничивающую пространство, занимаемое *mc*. Эти четыре свойства объекта соответствуют 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* (т. е. ограничивающий прямоугольник *mc* – *bounding box*). Для получения значений, хранящихся в возвращаемом объекте, нужно обращаться к его свойствам, как показано в следующем примере.

Размеры ограничивающей рамки клипа могут измеряться относительно любого другого клипа или фильма. С помощью аргумента *targetCoordinateSpace* можно поставить вопрос: «Если бы *mc* располагался на холсте *targetCoordinateSpace*, какую область он бы занимал?» Ответ различен в зависимости от того, является ли *targetCoordinateSpace* главным фильмом или экземпляром клипа: начальной точкой пространства координат главного фильма является левый верхний угол рабочего стола, а начальной точкой пространства координат экземпляра является его точка регистрации, как отмечено в библиотечном символе клипа (показана перекрестием).

С помощью метода *getBounds( )* можно обнаруживать коллизию между фильмом или клипом и какой-либо другой точкой (хотя *MovieClip.hitTest( )* лучше подходит для этой цели). Его можно также использовать, чтобы определить прямоугольную область для размещения клипа, добавляемого в фильм с помощью *MovieClip.attachMovie( )*.

***Пример***

```
var clipBounds = this.getBounds();
```

```
var leftX   = clipBounds.xMin;
var rightX  = clipBounds.xMax;
var topY    = clipBounds.yMin;
var bottomY = clipBounds.yMax;
```

***См. также***

*MovieClip.hitTest( ).*

---

**MovieClip.getBytesLoaded( ) метод** получить количество байт, загруженных в проигрыватель

***Доступность*** Flash 5

***Краткий обзор*** *mc*.getBytesLoaded()

***Возвращаемое значение***

Целое число, представляющее количество байт *mc*, загрузка которых в проигрыватель завершена. (Для перевода в килобайты следует разделить это число на 1024.)

***Описание***

Метод *getBytesLoaded( )* сообщает о количестве байт фильма, загруженных в проигрыватель. Однако *getBytesLoaded( )* измеряет байты только кусками по целым кадрам. Поэтому если первый кадр имеет размер 200 байт, а второй – 3000 байт, то *getBytesLoaded( )* возвратит 200 или 3200, но не число между ними. Пока не будет загружен весь кадр, загружаемый в данное время, значение *getBytesLoaded( )* не изменится. Поэтому метод *getBytesLoaded( )* можно рассматривать как «байтовую версию» свойства _framesloaded.

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( )* эффективен только при использовании с главным фильмом или внешним файлом *.swf*, который загружается в экземпляр или уровень.

Как и _framesloaded, *getBytesLoaded( )* обычно используется при создании предварительных загрузчиков. 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).

***Пример***

```
// КОД В КАДРЕ 1
if (_framesloaded > 0 && _framesloaded == _totalframes) {
  gotoAndPlay("beginMovie");
} else {
  // Показать ход загрузки в текстовых полях.
  // Разделить на 1024 для преобразования в килобайты.
```

```
    loadProgressOutput = this.getBytesLoaded()/1024;
    loadTotalOutput = this.getBytesTotal()/1024;
}

// КОД В КАДРЕ 2
gotoAndPlay(1);
```

***См. также***

*MovieClip._framesloaded*, ***MovieClip.getBytesTotal( )***.

---

**MovieClip.getBytesTotal( ) метод** получить размер клипа или фильма в байтах

***Доступность*** Flash 5

***Краткий обзор*** *mc*.getBytesTotal()

***Возвращаемое значение***

Целое число, представляющее размер *mc* в байтах. Для перевода в килобайты следует разделить его на 1024.

***Описание***

Метод ***getBytesTotal( )*** сообщает размер экземпляра клипа или главного фильма в байтах. При вызове с главным фильмом ***getBytesTotal( )*** сообщает размер всего ***.swf***. Обычно используется вместе с ***getBytesLoaded( )*** при создании предварительных загрузчиков для главных фильмов и файлов ***.swf***, загружаемых в экземпляры или на уровни.

***См. также***

***MovieClip.getBytesLoaded( )***, *MovieClip._totalframes*.

---

**MovieClip.getURL( ) метод** загрузить документ в окно броузера

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции введен во Flash 2, усилен во Flash 4 добавлением аргумента *method*)

***Краткий обзор*** *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( )*. Формат метода полезен только при отправке переменных, когда *getURL( )* посылает переменные из *mc*, не обязательно являющегося текущей временной диаграммой.

***См. также***

Общие рекомендации по применению в описании глобальной функции *getURL( )*.

---

**MovieClip.globalToLocal( ) метод** преобразовать точку на главном рабочем столе в координаты клипа

***Доступность*** Flash 5

***Краткий обзор*** `mc.globalToLocal(point)`

***Аргументы***

*point* Ссылка на объект с двумя свойствами `x` и `y`, описывающими точку на главном рабочем столе проигрывателя (т. е. на `_root`). 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 рабочего стола в пространство координат *mc*. Обратите внимание, что *globalToLocal( )* не возвращает новый объект, а изменяет существующие значения `x` и `y` точки.

Для использования *globalToLocal( )* нужно сначала создать объект со свойствами `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 рабочего стола 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. Когда объект создан, и заданы свойства `x` и `y`, объект передается методу *globalToLocal( )*:

```
myClip.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`. Рассмотрение новых значений свойств объекта `myPoint` дает ответ на вопрос: «В каком месте точка *(x, y)* главного рабочего стола появляется в `myClip`?» Например:

```
xInClip = myPoint.x;
yInClip = myPoint.y;
```

### *Пример*

В следующем примере вычисляется смещение точки регистрации текущего клипа относительно левого верхнего угла главного рабочего стола:

```
pt = new Object();          // Создать общий объект, содержащий точку
pt.x = 0;                   // Левая граница главного рабочего стола
pt.y = 0;                   // Верхняя граница главного рабочего стола
this.globalToLocal(pt);     // Преобразовать pt в местные 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** — перескочить в указанный кадр и начать воспроизведение

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 2 и выше)

***Краткий обзор***
```
mc.gotoAndPlay(frameNumber)
mc.gotoAndPlay(frameLabel)
```

### *Аргументы*

*frameNumber* Положительное целое число, указывающее номер кадра, в который должна перейти воспроизводящая головка *mc* перед воспроизведением. Если *frameNumber* меньше 1 или больше количества кадров во временной диаграмме *mc*, воспроизводящая головка направляется в первый или последний кадр соответственно.

*frameLabel* Строка, указывающая метку кадра, в который должна перейти воспроизводящая головка *mc* перед воспроизведением. Если *frameLabel* не найдена, воспроизводящая головка направляется в первый кадр временной диаграммы *mc*.

### *Описание*

Метод *MovieClip.gotoAndPlay( )* служит альтернативой глобальной функции *gotoAndPlay( )*. Используйте формат метода для управления удаленными клипами и фильмами, задаваемыми *mc*.

Общие рекомендации по применению см. в описании глобальной функции *gotoAndPlay( )*.

### *Пример*

```
// Отправить клип part1 на метку intro, затем воспроизвести part1
part1.gotoAndPlay("intro");
```

**MovieClip.gotoAndStop( ) метод** перейти в указанный кадр и остановить воспроизводящую головку

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 2 и выше)

***Краткий обзор***
```
mc.gotoAndStop(frameNumber)
mc.gotoAndStop(frameLabel)
```

***Аргументы***

*frameNumber* Положительное целое число, указывающее номер кадра, в который должна перейти воспроизводящая головка *mc*. Если *frameNumber* меньше 1 или больше количества кадров в временной диаграмме *mc*, воспроизводящая головка направляется в первый или последний кадр соответственно.

*frameLabel* Строка, указывающая метку кадра, в который должна перейти воспроизводящая головка *mc* перед воспроизведением. Если *frameLabel* не найдена, головка направляется в первый кадр временной диаграммы *mc*.

***Описание***

Метод *MovieClip.gotoAndStop( )* служит альтернативой глобальной функции *gotoAndStop( )*. Используйте формат метода для управления удаленными клипами и главными фильмами, задаваемыми *mc*.

Общие рекомендации по применению см. в описании глобальной функции *gotoAndStop( )*.

***Пример***

```
// Направить клип mainMenu в кадр 6 и остановить там воспроизводящую головку
mainMenu.gotoAndStop(6);
```

---

**MovieClip._height свойство** высота клипа или фильма в пикселах

***Доступность*** Flash 4; усилено во Flash 5

***Краткий обзор*** `mc._height`

***Тип доступа*** Только для чтения во Flash 4; Чтение/запись во Flash 5

***Описание***

Свойство `_height`, имеющее тип числа с плавающей точкой, является неотрицательным числом, указывающим высоту *mc* в пикселах. Если *mc* не имеет 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, содержащий фигуру, изображение, кнопку, клип или другой элемент содержимого. Свойство `_height` отражает изменения, сделанные в клипе с помощью среды разработки или через `_xscale`.

Можно задавать значение свойства `_height`, чтобы изменить размер клипа по вертикали. Попытка присвоить `_height` отрицательное значение игно-

рируется. Установка свойства _height в 0 не делает его невидимым, а превращает в горизонтальную линию толщиной в один пиксел.

Значение _height для главного фильма (т. е. *root*._height или _level*n*._height) является не высотой рабочего стола, указанной в меню Modify → Movie → Dimensions в среде разработки, а высотой содержимого главного фильма. У рабочего стола нет явного свойства высоты; при необходимости высота рабочего стола задается вручную как переменная. Например, если высота рабочего стола равна 400, можно добавить такую переменную:

```
_root.stageHeight = 400;
```

Сделать это значение доступным на временной диаграмме любого клипа можно так:

```
Object.prototype.stageHeight = 400;
```

***Особенности применения***

Обратите внимание, что при установке высоты клипа пропорционально масштабируются линии, утрачивая первоначальный размер своих точек, установленный в панели Stroke. Однако размер точки линий, для которых в панели Stroke установлен размер Hairline, не меняется при изменении размеров клипа. Таким образом, чтобы штрихи не становились толстыми и не искажались при изменении масштаба клипа, следует использовать волосяные линии.

***Пример***

```
ball._height = 20;          // Установить высоту ball равной 20 пикселам
ball._height /= 2;          // Уменьшить высоту ball в 2 раза
```

***См. также***

*MovieClip*._width, *MovieClip*._yscale.

---

**MovieClip.hitTest( ) метод** — проверить принадлежность точки клипу или пересечение двух клипов

***Доступность*** Flash 5

***Краткий обзор***
*mc*.hitTest(*x*, *y*, *shapeFlag*)
*mc*.hitTest(*target*)

***Аргументы***

*x* — Горизонтальная координата проверяемой точки.

*y* — Вертикальная координата проверяемой точки.

*shapeFlag* — Необязательное булево значение, указывающее, нужно ли делать проверку относительно ограничивающей рамки *mc* (false) или фактических пикселов *mc* (true). Значение по умолчанию false. Обратите внимание, что *shapeFlag* используется только с аргументами *x* и *y*, а не с аргументом *target*. Он имеет смысл, только если у *mc* неправильные очертания или есть дырка, как в пончике; 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. Результатом является `true`, если выполняется любое из следующих условий:

- Точка *(x, y)* на главном рабочем столе 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 главном рабочем столе пересекается с каким-либо пикселом в ограничивающей рамке *mc*. Ограничивающей рамкой *mc* считается наименьший прямоугольник, в который входят все занятые пикселы *mc*.
- Какой-либо пиксел в ограничивающей рамке *target* пересекается с каким-либо пикселом в ограничивающей рамке *mc*.

Результат имеет значение `false` в любом из следующих случаев:

- Свойство *shapeFlag* имеет значение `true`, и точка *(x, y)* на главном рабочем столе пересекается с каким-либо занятым пикселом *mc*. Занятым считается пиксел, содержащий визуальный элемент – например, фигуру или текст.
- Точка *(x, y)* на главном рабочем столе не пересекается ни с одним пикселом в ограничивающей рамке *mc*. Ограничивающей рамкой *mc* считается наименьший прямоугольник, в который входят все занятые пикселы *mc*.
- Никакие пикселы ограничивающей рамки *target* не пересекаются ни с какими пикселами в ограничивающей рамке *mc*.

***Описание***

Метод *hitTest( )* используется для определения того, пересекается ли клип или конкретная точка с *mc*.

При проверке пересечения точки с *mc* в *hitTest( )* передаются координаты *x* и *y* проверяемой точки (относительно главного рабочего стола). Можно также передать необязательный аргумент *shapeFlag*, который определяет, что должно использоваться при проверке коллизии – фактические пикселы *mc* или ограничивающая рамка *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 то, входит ли она в ограничивающую рамку *mc*.

При проверке пересечения клипа фильма с *mc* в *hitTest( )* передается аргумент *target*, задающий проверяемый клип. При обнаружении коллизий между *target* и *mc* всегда используются ограничивающие рамки клипов:

*hitTest( )* не поддерживает попиксельной проверки пересечения двух клипов. Ручные процедуры обнаружения коллизий на уровне пикселов трудно разрабатывать, и они требуют интенсивного использования процессора. Во многих случаях, например, в простой космической игре, часто используется проверка относительно ограничивающего круга, а не точных пикселов.

### *Особенности применения*

Обратите внимание, что коллизия всегда проверяется относительно расположения *mc* на главном рабочем столе проигрывателя, поэтому при использовании в *hitTest( )* одной точки аргументы x и y всегда должны описывать точку в координатах рабочего стола. О том, как преобразовывать координаты клипа в координаты рабочего стола, см. в описании *MovieClip.localToGlobal( )*. При проверке коллизии между клипами координаты клипов в различных временных диаграммах автоматически преобразуются в глобальные координаты (т. е. координаты рабочего стола).

### *Пример*

В этом примере показывается, как вручную определить пересечение двух окружностей без использования *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 в удобном 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");
}
```

Вот проверка пересечения paddle и ball с помощью *hitTest( )*:

```
if (paddle.hitTest("ball")) {
  trace("The paddle hit the ball.");
}
```

Теперь проверим, находится ли курсор мыши над занятым пикселом внутри контура tractor. Обратите внимание, что координаты курсора даны относительно _root (главного рабочего стола). Разрывы в клипе можно обнаружить, если *shapeFlag* имеет значение true; например, если tractor использует пустое пространство для представления окошек в нем, то *hitTest( )* возвратит false при нахождении курсора над окном:

```
if (tractor.hitTest(_root._xmouse, _root._ymouse, true)) {
  trace("You're pointing to the tractor.");
}
```

***См. также***

*MovieClip.getBounds( )*, *MovieClip.localToGlobal( )*.

---

**MovieClip.loadMovie( ) метод** загрузить в проигрыватель внешний файл *.swf*

***Доступность*** Формат метода появился во Flash 5 (формат глобальной функции поддерживался Flash 4)

***Краткий обзор***
```
mc.loadMovie(URL)
mc.loadMovie(URL, method)
```

***Аргументы***

*URL* Строка, задающая местонахождение внешнего файла *.swf*, который должен быть загружен.

*method* Необязательная строка, задающая метод пересылки переменных во внешний сценарий. Должна быть литералом `"GET"` или `"POST"`; другие выражения недопустимы.

***Описание***

Метод *MovieClip.loadMovie( )* является альтернативой глобальной функции *loadMovie( )*. При вызове в качестве метода *MovieClip loadMovie( )* принимает параметр `target`; он загружает в `mc` файл *.swf*, находящийся по указанному `URL`. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции.

Рекомендации по использованию см. в описании глобальной функции *loadMovie( )*.

***См. также***

*loadMovie( )*; раздел «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13.

---

**MovieClip.loadVariables( ) метод** получить группу внешних переменных

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 4 и выше)

***Краткий обзор***
```
mc.loadVariables(URL)
mc.loadVariables(URL, method)
```

***Аргументы***

*URL* Строка, задающая путь к одному из двух источников переменных: сценарию, создающему на выходе переменные, или текстовому файлу, содержащему переменные.

*method* Необязательная строка, задающая метод отправки переменных из *mc* во внешний сценарий. Если этот аргумент задан, осуществляются как отправка, так и загрузка переменных. Если опущен, переменные только загружаются. Должна быть литералом `"GET"` или `"POST"`; другие выражения недопустимы.

***Описание***

Метод *MovieClip.loadVariables( )* является альтернативой глобальной функции *loadVariables( )*. При вызове в качестве метода *MovieClip loadVariables( )* не принимает параметра `target` – он загружает переменные из `URL` в `mc`. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции.

Рекомендации по использованию см. в описании глобальной функции *loadVariables( )*.

***См. также***

*loadVariables( )*; раздел «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13.

---

**MovieClip.localToGlobal( ) метод** преобразовать точку в клипе в координаты главного рабочего стола

***Доступность*** Flash 5

***Краткий обзор*** `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 главном рабочем столе проигрывателя. Обратите внимание, что *localToGlobal( )* не возвращает новый объект, а просто модифицирует существующие значения `x` и `y` в `point`.

Для использования *localToGlobal( )* нужно сначала создать так называемую *точку*, или объект *coordinates* со свойствами `x` и `y`. Для этого просто создается общий объект из класса *Object*, к которому добавляются 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` (показанной 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 точки регистрации `mc`. Когда объект создан и свойства `x` и `y` установлены, объект передается методу *localToGlobal( )*:

```
myClip.localToGlobal(myPoint);
```

После выполнения *localToGlobal( )* значения свойств `x` и `y` у `myPoint` оказываются преобразованными так, что представляют соответствующую точку на главном рабочем столе, отсчитанную от его левого верхнего угла. Рас-

смотрение новых значений свойств объекта myPoint дает ответ на вопрос: «В каком месте точка клипа *(x, y)* появляется на главном рабочем столе?» Например:

```
mainX = myPoint.x;
mainY = myPoint.y;
```

***Пример***

В следующем примере определяется расположение точки регистрации клипа относительно главного рабочего стола:

```
pt = new Object();
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 главного
                           // рабочего стола

trace("On the main Stage, the registration point of the current clip is at: ");
trace("x: " + pt.x + "y: " + pt.y);
```

Метод *localToGlobal( )* можно использовать для преобразования точки в координаты рабочего стола при использовании с методом *hitTest( )*, которому нужно передавать точки в пространстве координат рабочего стола. Его можно также использовать для сравнения точек в двух различных клипах с помощью общего пространства координат.

***См. также***

*MovieClip.globalToLocal( )*, *MovieClip.hitTest( )*.

---

**MovieClip._name свойство** — идентификатор экземпляра клипа в виде строки

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._name

***Тип доступа*** Чтение/запись

***Описание***

Свойство _name указывает имя экземпляра клипа фильма в виде строки. Поскольку первоначально оно отражает имя экземпляра, установленное в панели Instance при редактировании, свойство _name неприменимо к главным фильмам (к главному фильму проще всего обращаться по глобальному свойству _root).

***Пример***

С помощью _name можно определить, нужно ли производить какие-то действия над данным клипом, как мы это делали при генерации серии клипов со звездами в примере 10.2.

Свойство _name можно использовать 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( );*** раздел «Имена экземпляров» в главе 13.

---

**MovieClip.nextFrame( ) метод** — переместить воспроизводящую головку клипа или фильма на один кадр вперед и остановить ее

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 2 и выше)

***Краткий обзор*** *mc*.nextFrame( )

***Описание***

Метод *MovieClip.nextFrame( )* является альтернативой глобальной функции *nextFrame( )*. Используйте формат метода для управления удаленными клипами или главными фильмами, заданными *mc*.

Общие рекомендации по применению см. в описании глобальной функции *nextFrame( )*.

***Пример***

```
// Переместить slideshow на один кадр вперед и остановить воспроизводящую головку
slideshow.nextFrame();
```

***См. также***

*MovieClip.prevFrame( )*, *nextFrame( )*.

---

**MovieClip._parent свойство** — ссылка на клип или фильм, содержащий данный

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._parent

***Тип доступа*** Только для чтения

***Описание***

В свойстве _parent хранится ссылка на объект клипа, на временной диаграмме которого расположен *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 в реализации***

Хотя во Flash 5 можно присвоить свойству _parent другой клип, в этом мало смысла: меняется только ссылка, а физическая структура клипов остается прежней. Такое непреднамеренное поведение в будущем будет, вероятно, изменено.

***Пример***

Если *mc* находится на главной временной диаграмме фильма, воспроизвести главную временную диаграмму из *mc* можно с помощью следующего кода:

```
_parent.play();
```

Можно также задавать свойства родительской временной диаграммы, например:

```
_parent._alpha = 50;
```

Свойство _parent можно также использовать последовательно, т. е. обращаться к родителю родителя, например:

```
_parent._parent.play();  // Воспроизвести клип, находящийся
                         // на два клипа выше текущего
```

***См. также***

_root; раздел «Ссылка на вложенные экземпляры» в главе 13.

---

**MovieClip.play( ) метод** — начать последовательный вывод кадров клипа или фильма

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 2 и выше)

***Краткий обзор*** *mc*.play()

***Описание***

Метод *MovieClip.play( )* является альтернативой глобальной функции *play( )*. Используйте формат метода для управления удаленными клипами или главными фильмами, заданными *mc*.

Общие рекомендации по применению см. в описании глобальной функции *play( )*.

***Пример***

```
// Начать воспроизведение клипа intro
intro.play();
```

***См. также***

*MovieClip.stop( )*, *play( )*.

---

**MovieClip.prevFrame( ) метод** — переместить воспроизводящую головку клипа или фильма на один кадр назад и остановить ее

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 2 и выше)

***Краткий обзор*** *mc*.prevFrame()

***Описание***

Метод *MovieClip.prevFrame( )* является альтернативой глобальной функции *prevFrame( )*. Используйте формат метода для управления удаленными клипами или главными фильмами, заданными *mc*.

Общие рекомендации по применению см. в описании глобальной функции *prevFrame( )*.

***Пример***

```
// Rewind slideshow one frame and stop it
slideshow.prevFrame();
```

***См. также***

*MovieClip.nextFrame( )*, *prevFrame( )*.

---

**MovieClip.removeMovieClip( ) метод** — удалить клип фильма из Flash Player

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 4 и выше)

***Краткий обзор*** *mc*.removeMovieClip()

***Описание***

*MovieClip.removeMovieClip( )* является альтернативой глобальной функции *removeMovieClip( )*. При вызове в качестве метода *MovieClip*, *removeMovieClip( )* не принимает параметра *target*; он удаляет *mc*. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции.

Рекомендации по использованию см. в описании глобальной функции *removeMovieClip( )*.

***См. также***

*duplicateMovieClip( ), MovieClip.attachMovie( ), MovieClip.duplicateMovieClip( )*; раздел «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13.

---

**MovieClip._rotation свойство** — угол поворота клипа или фильма в градусах

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._rotation

***Тип доступа*** Чтение/запись

***Описание***

Свойство _rotation, значение которого является числом с плавающей точкой, задает количество градусов, на которое *mc* поворачивается относительно первоначального направления (если *mc* клип, его *первоначальным направлением* является направление символа в библиотеке). В _rotation отражаются как действия в среде разработки, так и программные измене-

ния. 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* положительно) получается один и тот же результат. Например, нет разницы между поворотом клипа на +299.4 градуса и на −60.6 градусов. Аналогично, если *n* отрицательно, нет разницы между *n* градусами и *n+360* градусами. Например, поворот клипа на −90 градусов делает то же самое, что поворот на +270 градусов.

Если `_rotation` присваивается значение за пределами диапазона −180...180, 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 в реализации***

Во Flash 4 Player задание значения `_rotation` для клипа немного уменьшает масштаб этого клипа. После осуществления большого числа установок `_rotation` клип заметно сжимается. Чтобы учесть эту ошибку, задавайте `_xscale` и `_yscale` клипа равными 100 при каждой установке `_rotation`.

***Пример***

Размещение следующего кода в клипе влечет его поворот на 5 градусов по часовой стрелке при каждом выводе кадра:

```
onClipEvent (enterFrame) {
  _rotation += 5;
}
```

---

**MovieClip.startDrag( ) метод** — заставить фильм или клип фильма следовать за курсором мыши

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 4 и выше)

***Краткий обзор***

```
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 перемещена относительно своего первоначального положения (`false`).

*left* Число, задающее x-координату, левее которой нельзя перетаскивать точку регистрации *mc*.

| | |
|---|---|
| *top* | Число, задающее y-координату, выше которой нельзя перетаскивать точку регистрации *mc*. |
| *right* | Число, задающее x-координату, правее которой нельзя перетаскивать точку регистрации *mc*. |
| *bottom* | Число, задающее y-координату, ниже которой нельзя перетаскивать точку регистрации *mc*. |

***Описание***

Метод *MovieClip.startDrag( )* является альтернативой глобальной функции *startDrag( )*. При вызове в качестве метода *MovieClip startDrag( )* не принимает параметра *target*: он буксирует *mc*. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции.

Рекомендации по использованию см. в описании глобальной функции *startDrag( )*.

***Ошибки в реализации***

Обратите внимание, что правильным порядком координат ограничительного прямоугольника является следующий: левая, верхняя, правая, нижняя, но во Flash 5 ActionScript Dictionary они перечислены для *MovieClip.startDrag( )* в неправильном порядке.

***Пример***

```
// Код кнопки для перетаскивания текущего клипа или фильма
on (press) {
  this.startDrag(true);
}

on (release) {
  stopDrag();
}
```

***См. также***

*MovieClip.stopDrag( )*, *startDrag( )*, *stopDrag( )*; раздел «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13.

---

**MovieClip.stop( ) метод** сделать паузу в клипе или фильме

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 2 и выше)

***Краткий обзор*** *mc*.stop()

***Описание***

Метод *MovieClip.stop( )* является альтернативой глобальной функции *stop( )*. Используйте формат метода для управления удаленными клипами или главными фильмами, заданными *mc*.

Общие рекомендации по применению см. в описании глобальной функции *stop( )*.

***Пример***

```
// Остановить воспроизведение spinner
spinner.stop();
```

***См. также***

*MovieClip.play( )*, *stop( )*.

---

**MovieClip.stopDrag( ) метод** закончить текущую операцию буксировки

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 4 и выше)

***Краткий обзор*** *mc*.stopDrag()

***Описание***

Метод *MovieClip.stopDrag( )* является альтернативой глобальной функции *stopDrag( )*. Однако необходимости в использовании формата метода нет: *stopDrag( )* прекращает любую текущую операцию буксировки, будь она вызвана через клип или через глобальную функцию.

***См. также***

*MovieClip.startDrag( )*, *stopDrag( )*.

---

**MovieClip.swapDepths( ) метод** изменить графический слой экземпляра в стеке экземпляров

***Доступность*** Flash 5

***Краткий обзор***

*mc*.swapDepths(*target*)
*mc*.swapDepths(*depth*)

***Аргументы***

*target* Строка, указывающая путь к клипу, который нужно поменять местами с *mc*. Поскольку ссылки на клипы преобразуются в пути при использовании в строковом контексте, *target* может также являться ссылкой на объект клипа, например, *mc*.swapDepths(window2) вместо *mc*.swapDepths("window2").

*depth* Целое число, задающее слой в стеке клипов родителя *mc*. Уровень –1 находится под уровнем 0, 1 впереди 0, 2 впереди 1, и так далее. Более подробное описание см. в разделе «Порядок расположения фильмов и экземпляров в стеке» главы 13. Отрицательные уровни действуют, но официально не поддерживаются в ActionScript, поэтому для гарантии совместимости с будущими версиями следует использовать уровни 0 и выше.

***Описание***

Все экземпляры клипов в документе *.swf* располагаются в стеке, который управляет зрительным размещением клипов по слоям в проигрывателе. Стек имеет структуру колоды карт: клипы с более высоким положением в

стеке оказываются впереди клипов с более низким положением. Положение экземпляра в стеке устанавливается при создании клипа (в редакторе Flash или посредством *attachMovie( )* или *duplicateMovieClip( )*). С помощью *swapDepths( )* можно изменять положение экземпляра в стеке.

Метод *swapDepths( )* имеет два формата. При использовании аргумента `target` метод *swapDepths( )* обменивает позиции в стеке `mc` и `target` при условии, что родительский клип для `mc` и `target` один и тот же (они находятся на одной временной диаграмме). При использовании аргумента `depth` метод *swapDepths( )* помещает `mc` в новую позицию в стеке родителя `mc`. Если эта позиция занята, 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 см. в разделе «Порядок расположения фильмов и экземпляров в стеке» главы 13.

***Ошибки в реализации***

Обмен местами между дублированным или прикрепленным экземпляром с одной стороны, и созданным вручную экземпляром с другой при использовании аргумента `depth` может вызвать во Flash 5 Player проблемы с перерисовкой. Осторожно обращайтесь с аргументом `depth`, и всегда тщательно тестируйте код, обменивающий уровни.

***См. также***

раздел «Порядок расположения фильмов и экземпляров в стеке» в главе 13.

---

**MovieClip._target свойство** — путь к клипу или фильму с использованием слэш-синтаксиса

***Доступность*** Flash 4 и выше

***Краткий обзор*** `mc._target`

***Тип доступа*** Только для чтения

***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, свойство `_target` для `ball`'s имеет значение `"/ball"`. Для клипа `stripe` внутри `ball` свойство `_target` будет иметь значение `"/ball/stripe"`.

Для получения строки, содержащей путь к клипу в точечной нотации, воспользуйтесь функцией *targetPath( )*.

***См. также***

*targetPath( )*.

---

**MovieClip._totalframes свойство** — суммарное количество кадров на временной диаграмме клипа или фильма

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._totalframes

***Тип доступа*** Только для чтения

***Описание***

Целочисленное свойство _totalframes представляет количество кадров на временной диаграмме *mc*, где *mc* является клипом фильма или главным фильмом. Для нового клипа или главного фильма _totalframes всегда равно 1. Но если содержимое клипа выгружается через *unloadMovie( )*, _totalframes становится равным 0. Оно может быть нулем очень короткое время, когда текущий клип выгружен, а новый клип загружается посредством операции *loadMovie( )* через очень медленное соединение. Свойство _totalframes чаще всего используется вместе с _frameloaded для создания предварительных загрузчиков, как показано в описании MovieClip._framesloaded.

***См. также***

*MovieClip*._framesloaded.

---

**MovieClip.unloadMovie( ) метод** — удалить фильм или клип фильма из проигрывателя

***Доступность*** Формат метода введен во Flash 5 (формат глобальной функции поддерживается Flash 3 и выше)

***Краткий обзор*** *mc*.unloadMovie()

Метод *MovieClip.unloadMovie( )* является альтернативой глобальной функции *unloadMovie( )*. При вызове в качестве метода *MovieClip*, *unloadMovie( )* не принимает параметра *target*: он выгружает *mc*. Синтаксис метода меньше подвержен возможным ошибкам пользователя, чем синтаксис соответствующей глобальной функции.

Рекомендации по использованию см. в описании глобальной функции *unloadMovie( )*.

***Пример***

```
// Удаляет загруженный документ с уровня 1
_level1.unloadMovie();
```

***См. также***

*MovieClip.loadMovie( )*, *unloadMovie( )*; раздел «Порядок расположения фильмов и экземпляров в стеке» в главе 13.

---

**MovieClip._url свойство** — сетевой адрес, с которого был загружен клип или фильм

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._url

***Тип доступа*** Только для чтения

***Описание***

Свойство _url представляет URL (универсальный локатор ресурсов), задающий адрес в Интернете или на диске, с которого было загружено содержи-

мое *mc*, в виде строки. Свойство _url всегда является абсолютным, а не относительным 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 используется для создания простых систем защиты, предотвращающих воспроизведение фильмов в нежелательном месте.

***Пример***

Значение _url в фильме, загруженном с 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 нахождения фильма в желаемом месте (если он не там, выводится кадр с сообщением об ошибке):

```
if (_url != "http://www.moock.org/gwen/meetgwen.swf") {
  trace ("This movie is not running from its intended location.";
  gotoAndStop("accessDenied");
}
```

***См. также***

*MovieClip.loadMovie( )*.

---

**MovieClip.valueOf( ) метод** путь к клипу или фильму в виде строки

***Доступность*** Flash 5

***Краткий обзор*** *mc*.valueOf( )

***Возвращаемое значение***

Строка, содержащая полный путь к *mc*, записанная с использованием точечного синтаксиса. Например:

```
"_level1.ball"
"_level0"
"_level0.shoppingcart.product1"
```

***Описание***

Метод *valueOf( )* возвращает строку, содержащую абсолютный путь к *mc* в точечной нотации. Метод *valueOf( )* автоматически вызывается, когда объект *MovieClip* используется там, где требуется строка. Например, trace(*mc*) выведет такой же результат, как и trace(*mc*. valueOf()). Поэтому необходимость в явном вызове *valueOf( )* возникает редко.

***См. также***

*Object.valueOf( )*, *targetPath( )*.

**MovieClip._visible свойство** обозначает, показывается клип или фильм, или скрыт

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._visible

***Тип доступа*** Чтение/запись

***Описание***

Булево свойства _visible указывает на то, что *mc* в данный момент показывается (true) или скрыт (false). Свойство _visible используется как средство быстро скрыть из видимости клип или фильм. Обратите внимание, что скрытыми фильмами можно по-прежнему управлять через ActionScript, и они по-прежнему могут воспроизводиться, останавливаться, получать события и в других отношениях вести себя, как обычно. Скрытые клипы просто не выводятся на экран.

Начальным значением свойства _visible для всех клипов является true, даже для полностью прозрачных клипов или находящихся за пределами рабочего стола. Свойство _visible изменяется только при намеренной модификации его сценарием; его нужно рассматривать как программный способ показывать и скрывать клипы, а не отражение всех обстоятельств, таких как положение на рабочем столе и прозрачность, которые влияют на видимость клипа.

Для того чтобы сделать клип невидимым, предпочтительнее воспользоваться свойством _visible, чем делать клип полностью прозрачным или удалять его с рабочего стола, потому что Flash Player не пытается рисовать графику в клипах, у которых _visible установлено в false, что увеличивает производительность вывода изображений.

***Пример***

Следующий код кнопки скрывает текущий клип при нажатии на кнопку и показывает его вновь, когда кнопка отпущена:

```
on (press) {
  this._visible = false;
}
on (release) {
  this._visible = true;
}
```

***См. также***

*MovieClip*._alpha.

---

**MovieClip._width свойство** ширина клипа или фильма в пикселах

***Доступность*** Flash 4; усилено во Flash 5

***Краткий обзор*** *mc*._width

***Тип доступа*** Только для чтения во Flash 4; чтение/запись во Flash 5

### *Описание*

Свойство _width, имеющее тип числа с плавающей точкой, является неотрицательным числом, указывающим высоту *mc* в пикселах. Если *mc* не имеет содержимого, _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, содержащий фигуру, изображение, кнопку, клип или другой элемент содержания. Свойство _width отражает изменения, сделанные в клипе в среде разработки или через _xscale.

Можно задать значение свойства _width, чтобы изменить размер клипа по горизонтали. Попытка присвоить _width отрицательное значение игнорируется. Установка свойства _width в 0 не делает его невидимым, а превращает в вертикальную линию толщиной в один пиксел.

Значение _width для главного фильма (т. е. *root*._width или _level*n*._width) является не высотой рабочего стола, указанной в меню Modify → Movie → Dimensions в среде разработки, а шириной содержимого главного фильма. У рабочего стола нет явного свойства ширины; при необходимости ширина рабочего стола задается вручную как переменная. Например, если ширина рабочего стола равна 550, можно добавить такую переменную:

```
_root.stageWidth = 550;
```

Сделать это значение доступным на временной диаграмме любого клипа можно так:

```
Object.prototype.stageWidth = 550;
```

### *Особенности применения*

Обратите внимание, что при установке ширины клипа пропорционально масштабируются линии, утрачивая первоначальный размер точки, установленный в панели Stroke. Однако размер точки линий, для которых в панели Stroke установлен размер Hairline, не меняется при изменении размеров клипа. Таким образом, чтобы штрихи не становились толстыми и не искажались при изменении масштаба клипа, следует использовать волосяные линии.

### *Пример*

```
ball._width = 20;          // Установить ширину ball равной 20 пикселам
ball._width *= 2;          // Удвоить ширину ball
```

### *См. также*

*MovieClip*._height, *MovieClip*._xscale.

---

**MovieClip._x свойство** горизонтальное положение клипа или фильма в пикселах

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._x

***Тип доступа*** Чтение/запись

### *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 *рабочего стола*. Например, _x, равное 20, указывает, что точка регистрации mc находится на 29 пикселов правее левого края рабочего стола; −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; −20 обозначает 20 пикселов влево.
- Если mc является главным фильмом, то _x представляет смещение по горизонтали всего документа *.swf* относительно левого края рабочего стола. Например, _x, равное 200, указывает, что содержимое рабочего стола смещено на 200 вправо от позиции во время редактирования; −200 обозначает 200 пикселов влево.

Свойство _x (как и все горизонтальные координаты во Flash) возрастает в правую сторону и уменьшается в левую. Дробные значения _x аппроксимируются во Flash с помощью сглаживания.

Если mc содержится в экземпляре, который масштабирован и/или повернут, пространство координат, в котором он находится, тоже масштабировано и/или повернуто. Например, если родитель mc имеет масштаб 200% и повернут на 90 градусов 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 кода вызывает его перемещение вправо на 5 пикселов в каждом новом кадре (в предположении, что его пространство координат не изменилось в результате преобразований, произведенных над его родителем):

```
onClipEvent (enterFrame) {
  _x += 5;
}
```

Расположение клипов с помощью _x и _y является базовой задачей визуального программирования на ActionScript. В следующем примере показаны обработчики событий клипов, которые вызывают перемещение клипа за курсором мыши с фиксированной скоростью (из сетевого хранилища кода можно получить много других примеров перемещений):

```
// Заставить клип перемещаться за курсором мыши
```

```
onClipEvent(load) {
  this.speed = 10;

  // Перемещает клип в сторону точки (leaderX, leaderY)
  function follow (clip, leaderX, leaderY) {
    // Перемещаться, только если не достигнута точка назначения
    if (clip._x != leaderX || clip._y != leaderY) {
      // Определить расстояние между clip и leader
      var deltaX = clip._x - leaderX;
      var deltaY = clip._y - leaderY;
      var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));

      // Разложить speed по осям X и Y
      var moveX = clip.speed * (deltaX / dist);
      var moveY = clip.speed * (deltaY / dist);

      // Если у клипа достаточно скорости, чтобы перескочить точку
      // назначения,то просто перейти в точку назначения. Иначе
      // переместиться в соответствии  со скоростью клипа.
      if (clip.speed >= dist) {
        clip._x = leaderX;
        clip._y = leaderY;
      } else {
        clip._x -= moveX;
        clip._y -= moveY;
      }
    }
  }
}

onClipEvent(enterframe) {
  follow(this, _root._xmouse, _root._ymouse);
}
```

***См. также***

*MovieClip.globalToLocal( )*, *MovieClip.localToGlobal( )*, `MovieClip._y`.

---

**MovieClip._xmouse свойство** — горизонтальное расположение курсора мыши

***Доступность*** Flash 5

***Краткий обзор*** `mc._xmouse`

***Тип доступа*** Только для чтения

***Описание***

Свойство `_xmouse`, представляемое числом с плавающей точкой, указывает горизонтальное расположение курсора мыши относительно пространства координат *mc*. Если *mc* является главным фильмом, `_xmouse` измеряется от левого края рабочего стола. Если *mc* является экземпляром, `_xmouse` измеряется от точки регистрации экземпляра. Для получения единообразной координаты `_xmouse`, которая всегда измеряется относительно рабочего стола, используйте `_root._xmouse`.

***Пример***

Размещение в клипе следующего кода заставляет его отражать горизонтальное положение курсора мыши относительно рабочего стола (перемещаясь влево и вправо по прямой линии):

```
onClipEvent (enterFrame) {
  _x = _root._xmouse;
}
```

***См. также***

*MovieClip*._ymouse.

---

**MovieClip._xscale свойство** ширина клипа или фильма в виде процента

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._xscale

***Тип доступа*** Чтение/запись

***Описание***

Свойство _xscale, представляемое числом с плавающей точкой, указывает ширину *mc* по отношению к исходной ширине в процентном отношении. Если *mc* является экземпляром, то его «исходная ширина» является шириной символа экземпляра в библиотеке. Если *mc* является главным фильмом, его «исходной шириной» является ширина фильма во время редактирования.

Когда текущая ширина *mc* совпадает с исходной шириной, _xscale равняется 100. При _xscale 200 ширина *mc* по сравнению с исходной удваивается. При _xscale 50 ширина *mc* вдвое меньше исходной.

Свойство _xscale масштабирует клип относительно точки регистрации (пропорционально в левую и правую стороны от точки регистрации клипа). Чтобы размер клипа изменялся только в одну сторону, разместите все его содержимое по одну сторону от точки регистрации в символе клипа (этот прием полезен при создании горизонтального указателя в предварительном загрузчике). Если присвоить свойству клипа _xscale отрицательное значение, он переворачивается по горизонтали, как если бы через точку регистрации проходило вертикальное зеркало (т. е. становится зеркальным отражением самого себя), после чего отрицательное значение рассматривается как положительное. Для горизонтального переворота клипа без изменения его размеров задайте свойство _xscale равным –100.

***Пример***

```
// Удвоить ширину ball (не изменяя высоты)
ball._xscale *= 2;

// Создать зеркальное отражение ball
ball._xscale = -100;
```

***См. также***

MovieClip._yscale.

**MovieClip._y свойство** вертикальное положение клипа или фильма в пикселах

***Доступность*** Flash 4 и выше

***Краткий обзор*** *mc*._y

***Тип доступа*** Чтение/запись

***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, _y измеряется относительно верхнего края *рабочего стола*. Например, _y, равное 20, указывает, что точка регистрации *mc* находится на 20 пикселов ниже верхнего края рабочего стола; −20 обозначает 20 пикселов вверх.
- Если *mc* располагается на временной диаграмме другого 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 экземпляр; −20 обозначает 20 пикселов вверх.
- Если *mc* является главным фильмом, то _y представляет смещение по вертикали всего документа *.swf* относительно верхнего края рабочего стола. Например, _y, равное 200, указывает, что содержимое рабочего стола смещено на 200 вниз от позиции во время редактирования; −200 обозначает 200 пикселов вверх.

Свойство _y (как и все вертикальные координаты во Flash) возрастает вниз и уменьшается вверх − противоположно декартовой системе координат. Дробные значения _y аппроксимируются во Flash с помощью сглаживания.

Если *mc* содержится в экземпляре, который масштабирован и/или повернут, пространство координат, в котором он находится, тоже масштабировано и/или повернуто. Например, если родитель *mc* имеет масштаб 200% и повернут на 90 градусов по часовой стрелке, то _y будет возрастать влево, а не вниз, и 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***

Размещение в клипе следующего кода вызывает его перемещение вниз на 5 пикселов в каждом новом кадре (в предположении, что его пространство координат не изменилось в результате преобразований, произведенных над его родителем):

```
onClipEvent (enterFrame) {
  _y += 5;
}
```

***См. также***

*MovieClip.globalToLocal( )*, *MovieClip.localToGlobal( )*, `MovieClip._x`.

---

**MovieClip._ymouse свойство** вертикальное расположение курсора мыши

***Доступность*** Flash 5

***Краткий обзор*** `mc._ymouse`

***Тип доступа*** Только для чтения

***Описание***

Свойство `_ymouse`, представляемое числом с плавающей точкой, указывает горизонтальное расположение курсора мыши относительно пространства координат `mc`. Если `mc` является главным фильмом, `_ymouse` измеряется от верхнего края рабочего стола. Если `mc` является экземпляром, `_ymouse` измеряется от точки регистрации экземпляра. Для получения единообразной координаты `_ymouse`, которая всегда измеряется относительно рабочего стола, используйте `_root._ymouse`.

***Пример***

Размещение в клипе следующего кода заставляет его отражать вертикальное положение курсора мыши относительно рабочего стола (перемещаясь вверх и вниз по прямой линии):

```
onClipEvent (enterFrame) {
  _y = _root._ymouse;
}
```

***См. также***

`MovieClip._xmouse`.

---

**MovieClip._yscale свойство** высота клипа или фильма как процент

***Доступность*** Flash 4 и выше

***Краткий обзор*** `mc._yscale`

***Тип доступа*** Чтение/запись

***Описание***

Свойство `_yscale`, представляемое числом с плавающей точкой, указывает высоту `mc` по отношению к исходной высоте в процентном отношении. Если `mc` является экземпляром, то его «исходная высота» является высотой символа экземпляра в библиотеке. Если `mc` является главным фильмом, его «исходной высотой» является высота фильма во время редактирования.

Когда текущая высота `mc` совпадает с исходной высотой, `_yscale` равняется 100. При `_yscale` 200 высота `mc` удвоена по сравнению с исходной. При `_yscale` 50 высота `mc` вдвое меньше исходной.

Свойство `_yscale` масштабирует клип относительно точки регистрации (пропорционально вверх и вниз от точки регистрации клипа). Чтобы размер клипа изменялся только вверх и вниз, разместите все его содержимое

по одну сторону от точки регистрации в символе клипа (этот прием полезен при создании вертикального указателя в предварительном загрузчике). Если присвоить свойству клипа _yscale отрицательное значение, он переворачивается по вертикали, как если бы через точку регистрации проходило горизонтальное зеркало (т. е. становится зеркальным отражением самого себя по вертикали), после чего отрицательное значение рассматривается как положительное. Для горизонтального переворота клипа без изменения его размеров установите свойство _yscale равным –100.

***Пример***

```
// Удвоить высоту ball (не изменяя ширины)
ball._yscale *= 2;
// Перевернуть ball в вертикальном направлении
ball._yscale = -100;
```

***См. также***

*MovieClip*._xscale.

---

**NaN глобальное свойство** константа, представляющая неверные числовые данные (Not-a-Number)

***Доступность*** Flash 5

***Краткий обзор*** NaN

***Тип доступа*** Только для чтения

***Описание***

NaN является особой числовой константой, используемой для представления неверных числовых данных (NaN служит акронимом для «Not-a-Number»). Числовые операции, которые не могут быть разрешены в 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, хотя с ним нельзя проводить вычисления:

```
typeof NaN;    // Дает значение"number"
```

***Особенности применения***

Значение NaN редко непосредственно используется в исходном коде, но является для операции способом сообщить об ошибке. Поскольку сравнение NaN с самим собой не дает равенства, обнаружить его можно только с помощью глобальной функции *isNaN( )*. NaN служит сокращенной записью Number.NaN.

***См. также***

*isNaN( )*, Number.NaN; раздел «Особые значения числового типа данных» в главе 4.

**newline константа** вставка переноса строки

***Доступность*** Flash 4 и выше

***Краткий обзор*** `newline`

***Возвращаемое значение***

Символ перевода строки.

***Описание***

Константа `newline` представляет стандартный символ переноса строки (ASCII 10). Она служит синонимом для escape-последовательности `"\n"` и используется для принудительного переноса строки в блоке текста (обычно при выводе в переменной текстового поля).

***Особенности применения***

Во Flash 4 `newline` была функцией, но во Flash 5 стала константой, и синтаксис ее напоминает синтаксис свойства или переменной. Обратите внимание, что после `newline` не ставятся скобки.

***Пример***

```
myOutput = "hello" + newline + "world";
```

***См. также***

раздел «Escape-последовательности» в главе 4.

---

**nextFrame( ) глобальная функция** переместить воспроизводящую головку фильма или клипа вперед на один кадр и остановить ее

***Доступность*** Flash 2 и выше

***Краткий обзор*** `nextFrame()`

***Описание***

Функция *nextFrame( )* перемещает воспроизводящую головку текущей временной диаграммы вперед на один кадр и останавливает ее там. «Текущей временной диаграммой» является временная диаграмма, из которой вызвана функция *nextFrame( )*. Функция *nextFrame( )* – синоним для *gotoAndStop( _currentFrame + 1 );*. При вызове из последнего кадра временной диаграммы *nextFrame( )* просто останавливает в этом кадре воспроизводящую головку, если только за текущей сценой не следует другая, и тогда *nextFrame( )* перемещает воспроизводящую головку в кадр 1 следующей сцены.

***См. также***

*gotoAndStop( )*, *MovieClip.nextFrame( )*, *nextScene( )*, *prevFrame( )*.

---

**nextScene( ) глобальная функция** переместить воспроизводящую головку в кадр 1 следующей сцены

***Доступность*** Flash 2 и выше

***Краткий обзор*** `nextScene()`

***Описание***

Функция *nextScene( )* перемещает главную воспроизводящую головку фильма в кадр 1 следующей сцены и останавливает временную диаграмму главного фильма. «Текущей сценой» считается та, из которой вызывается функция *nextScene( )*. Чтобы иметь какой-либо результат, она должна вызываться из главной временной диаграммы фильма – это значит, что *nextScene( )* не работает внутри клипа или обработчика *onClipEvent( )*. При вызове из последней сцены фильма *nextScene( )* отправляет воспроизводящую головку в кадр 1 этой сцены и останавливает воспроизведение фильма.

***См. также***

*nextFrame( )*, *prevScene( )*.

---

**Number( ) глобальная функция** преобразовать величину в тип данных *Number*

***Доступность*** Flash 5

***Краткий обзор*** `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. Часто надобности в использовании функции *Number( )* нет: ActionScript при необходимости автоматически преобразует значения к типу *number*.

Не путайте глобальную функцию *Number( )* с конструктором класса *Number*. Первая из них является функцией, которая преобразует величину в тип *number*, тогда как последний является классом, служащим оболочкой для элементарного числового элемента данных в объекте, который может иметь свойства и методы.

***Особенности применения***

Обратите внимание, что функция *Number( )* часто появляется в файлах *.fla* Flash 4, которые преобразованы в формат Flash 5. Сведения о действиях с типами данных при конвертировании файлов Flash 4 в файлы Flash 5 см. в разделе «Преобразование типов данных Flash 4 во Flash 5» в главе 3.

***См. также***

The *Number* class, *parseFloat( )*, *parseInt( )*; раздел «Явное преобразование типов данных» в главе 3.

---

**Number класс** класс-оболочка для элементарных числовых данных

***Доступность*** Flash 5

***Конструктор*** `new Number(value)`

***Аргументы***

*value* Выражение, которое должно быть вычислено и, при необходимости, преобразовано в числовое значение, а затем заключено в объект *Number*.

***Свойства класса***

Доступ к перечисленным ниже свойствам осуществляется непосредственно как к свойствам класса *Number*, в виде `Number.propertyName`. Для обращения к ним не нужно создавать новый экземпляр объекта *Number* (т. е. не нужен конструктор). Для некоторых из этих свойств, таких как `NaN`, даже не требуется обозначение в виде `Number.propertyName`. Можно использовать просто `NaN` как сокращение для `Number.NaN`.

*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 «не число», указывающее неверный числовой элемент данных.

*NEGATIVE_INFINITY* Любое число, меньшее `-MAX_VALUE`.

*POSITIVE_INFINITY* Любое число, большее `MAX_VALUE`.

***Методы***

*toString( )* Преобразовать число в строку.

***Описание***

Класс *Number* имеет два назначения:

- Он предоставляет доступ ко встроенным свойствам, представляющим специальные числовые значения – `MIN_VALUE`, `MIN_VALUE`, `NaN`, `NEGATIVE_INFINITY` и `POSITIVE_INFINITY`, которые можно использовать для проверки действительности числовых данных.
- Его можно использовать для преобразования чисел из одной системы исчисления в другую – например, чисел десятичной системы счисления (десятичных) и шестнадцатеричных чисел. См. метод *Number.toString( )*.

***Особенности применения***

Не требуется создавать новый объект *Number*, если нужен просто доступ к определенным в нем числовым свойствам. На самом деле, проще исполь-

зовать эквиваленты в виде глобальных свойств, когда они есть (NaN, Infinity и -Infinity). Честно говоря, свойства класса *Number* вообще редко требуются.

С другой стороны, метод класса *Number toString( )* используется с созданным экземпляром объекта *Number*. Однако интерпретатор берет на себя создание объекта *Number*, когда вызывается метод для элементарного числового значения. Например:

```
x = 102;
x.toString(16);  // x автоматически преобразуется в объект Number,
                 // чтобы выполнить эту операцию.
```

Это можно делать, чтобы, скажем, с помощью *toString( )* осуществлять преобразования между разными системами счисления. Опять-таки, такая задача возникает достаточно редко, и вряд ли вам придется много использовать класс *Number*.

***См. также***

The *Math* object; раздел «Тип данных Number» в главе 3.

---

**Number.MAX_VALUE свойство** — самое большое положительное число, которое можно представить в ActionScript

***Доступность*** Flash 5

***Краткий обзор*** Number.MAX_VALUE

***Тип доступа*** Только для чтения

***Описание***

Свойство MAX_VALUE хранит самое большое положительное число, которое можно представить в ActionScript (1.7976931348623157e+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***

Производится поиск минимального значения, содержащегося в массиве. Переменной minVal присваивается начальное значение MAX_VALUE, поэтому любое последующее значение будет меньше:

```
var myArray = [-10, 12, 99]
var minVal = Number.MAX_VALUE
for (thisValue in myArray) {
  if (myArray[thisValue] < minVal) {
    minVal = myArray[thisValue];
  }
}
trace ("The minimum value is " + minVal);
```

***См. также***

`Number.MIN_VALUE`; `Number.POSITIVE_INFINITY`; раздел «Особые значения числового типа данных» в главе 4.

---

**Number.MIN_VALUE свойство** самое маленькое положительное число, которое можно представить в ActionScript

***Доступность*** Flash 5

***Краткий обзор*** `Number.MIN_VALUE`

***Тип доступа*** Только для чтения

***Описание***

Свойство `MIN_VALUE` хранит самое маленькое положительное число, которое можно представить в ActionScript, 5e-324 (не путать с `-Number.MAX_VALUE`, самым маленьким отрицательным числом, допускаемым ActionScript).

***См. также***

`Number.MAX_VALUE`; раздел «Особые значения числового типа данных» в главе 4.

---

**Number.NaN свойство** константа, представляющая неверные числовые данные (не-число)

***Доступность*** Flash 5

***Краткий обзор*** `Number.NaN`

***Тип доступа*** Только для чтения

***Описание***

В свойстве `NaN` хранится специальное значение «неверное число» с типом данных *number*. Оно используется для представления результата нелогичного математического вычисления (например, 0/0) или попытки преобразования данных, которое не дает действительного числа.

`Number.NaN` обычно используется в более удобной форме глобального свойства, `NaN`.

***Особенности применения***

Значение `NaN` редко непосредственно используется в исходном коде, но служит способом для операции сообщить об ошибке. Обнаружить его можно только с помощью глобальной функции *isNaN( )*.

***См. также***

*isNaN( )*, `NaN`; раздел «Особые значения числового типа данных» в главе 4.

---

**Number.NEGATIVE_INFINITY свойство** константа, представляющая любое число, которое меньше чем `-Number.MAX_VALUE`

***Доступность*** Flash 5

***Краткий обзор*** `Number.NEGATIVE_INFINITY`

***Тип доступа*** Только для чтения

***Описание***

Свойство `NEGATIVE_INFINITY` хранит особое числовое значение, используемое для представления чисел, меньших `-Number.MAX_VALUE` (самое отрицательное число, представимое в ActionScript). Это называется *переполнением* и обычно вызывается математической ошибкой.

`Number.NEGATIVE_INFINITY` обычно используется в более удобной форме глобального свойства, `-Infinity`.

***См. также***

`-Infinity`, *isFinite( )*, `Number.MAX_VALUE`; раздел «Особые значения числового типа данных» в главе 4.

---

**Number.POSITIVE_INFINITY свойство** константа, представляющая любое число, большее чем `Number.MAX_VALUE`

***Доступность*** Flash 5

***Краткий обзор*** `Number.POSITIVE_INFINITY`

***Тип доступа*** Только для чтения

***Описание***

Свойство `POSITIVE_INFINITY` хранит специальное числовое значение, используемое для представления значений, больших `Number.MAX_VALUE` (наибольшее число, представимое в ActionScript). Оно также известно как *переполнение* и обычно вызывается какой-либо математической ошибкой.

`Number.POSITIVE_INFINITY` обычно используется в более удобной форме глобального свойства `Infinity`.

***Пример***

```
if (score == Number.POSITIVE_INFINITY) {
  trace ("You've achieved the highest possible score.");
}
```

***См. также***

`Infinity`, *isFinite( )*, `Number.MAX_VALUE`; раздел «Особые значения числового типа данных» в главе 4.

---

**Number.toString( ) метод** преобразовать число в строку

***Доступность*** Flash 5

***Краткий обзор*** *numberObject*`.toString(`*radix*`)`

***Аргументы***

*radix* Целое число между 2 и 36, задающее основание системы счисления, используемой для представления *numberOb-*

*ject* в строковом формате. Этот аргумент необязателен. 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. Аргумент *radix* может использоваться для преобразования чисел в различные системы счисления (например, двоичные, восьмеричные, десятичные и шестнадцатеричные). Буквы A–Z используются для представления цифр со значениями 10–35, соответственно, хотя обычно используются только буквы от A до F (для представления шестнадцатеричных цифр от 10 до 15).

Для использования *Number.toString( )* с элементарным числовым значением следует заключить его в круглые скобки, например:

```
(204).toString(16);
```

***Пример***

```
x = new Number(255);
trace(x.toString());        // Выводит: "255" (т. е. десятичное)
trace(x.toString(16));      // Выводит: "ff" (т. е. шестнадцатеричное)
trace(x.toString(2));       // Выводит: "11111111" (т. е. двоичное)

// Преобразовать шестнадцатеричный литерал в десятичный
trace((0xFFFFFF).toString(10));  // Displays: "16777215"
```

***См. также***

*Number ( )*, *Object.toString( )*, *parseInt( )*; см. раздел «Целочисленные литералы» в главе 4.

---

**Object класс** основа для всех остальных классов и родовых объектов

***Доступность*** Flash 5

***Конструктор*** `new Object( )`

***Свойства***

| | |
|---|---|
| *constructor* | Ссылка на функцию конструктора класса, создающую объект. |
| *__proto__* | Ссылка на свойство `prototype` функции конструктора класса. |

***Методы***

| | |
|---|---|
| *toString( )* | Преобразовать значение объекта в строку. |
| *valueOf( )* | Извлечь значение объекта элементарного типа, если оно существует. |

***Описание***

Класс *Object* является базовым в объектной модели ActionScript. *Object* используется в двух главных целях: (а) как конструктор для создания но-

вых родовых объектов; и (b) 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 *Object* (хотя некоторые классы переопределяют эти свойства).

Чтобы создать родовой объект класса Object непосредственно в коде без использования конструктора, можно использовать литерал объекта, как можно использовать литерал строки или литерал массива. Литерал объекта состоит из ряда пар имя/значение для свойств, разделяемых запятыми и 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 «Лексическая структура». Значениями могут быть любые допустимые выражения. Например:

```
// Литерал объекта с двумя числовыми свойствами
myObject = { x: 30, y: 23 };
// Установить значение свойства x с помощью сложного выражения
myOtherObject = { x: Math.floor(Math.random() * 50 + 1) };
```

Поскольку литералы объектов всегда создают родовые анонимные объекты, они используются только в случае временной нужды в данных в формате объекта, например, при вызове *Sound.setTransform( )*, *Color.setTransform( )* или *MovieClip.localToGlobal( )*.

***См. также***

главу 12 «Объекты и классы».

---

**Object.constructor свойство** ссылка на функцию конструктора класса, используемую для создания объекта

***Доступность*** Flash 5

***Краткий обзор*** *someObject*.constructor

***Тип доступа*** Чтение/запись (переписывать свойство объекта constructor не рекомендуется, так как в результате изменяется естественная структура наследования классов).

***Описание***

В свойстве constructor хранится ссылка на функцию конструктора, использовавшуюся для создания *someObject*. Например, свойство constructor объекта *Date* содержит ссылку на функцию конструктора *Date*:

```
now = new Date();
now.constructor == Date;       // Дает значение true
```

***См. также***

раздел «Свойство constructor» в главе 12.

**Object.__proto__ свойство** ссылка на свойство prototype конструктора объекта

***Доступность*** Flash 5

***Краткий обзор*** *someObject*.__proto__

***Тип доступа*** Чтение/запись (переписывать свойство объекта __proto__ не рекомендуется, так как в результате изменяется естественная структура наследования классов.)

***Описание***

Свойство __proto__ хранит ссылку на автоматически назначаемое свойство prototype функции конструктора *someObject*, используемое для передачи свойств вниз по иерархии классов. Свойство объекта __proto__ в основном используется интерпретатором внутренне для поиска унаследованных объектом свойств, но можно использовать его также для определения класса объекта, как показано в примерах 12.6 и 12.9. Обратите внимание на то, что __proto__ начинается и оканчивается *двумя* символами подчеркивания.

***См. также***

раздел «Свойство __proto__» в главе 12.

---

**Object.toString( ) метод** значение объекта в виде строки

***Доступность*** Flash 5

***Краткий обзор*** *someObject*.toString()

***Возвращаемое значение***

Внутренне определенная строка, которая описывает объект или представляет его еще каким-либо способом.

***Описание***

Метод *toString( )* возвращает описание *someObject* в виде строки. По умолчанию *someObject*.toString() возвращает выражение:

```
"[object " + class + "]"
```

где *class* является внутренне определенным именем класса, к которому принадлежит *someObject*. Интерпретатор ActionScript автоматически вызывает метод *toString( )*, когда *someObject* используется в строковом контексте. Например:

```
x = new Object();
trace(x);       // Выводит: "[object Object]"
```

Большинство классов переписывает метод *toString( )*, используемый *Object* по умолчанию, чтобы предоставить более содержательную информацию о каждом члене класса. Например, метод *Date.toString( )* возвращает дату и время, а метод *Array.toString( )* возвращает разделяемый запятыми список элементов массива. То же можно делать и при создании собственных классов.

***Пример***

В этом примере показано, как создать собственный метод *toString( )* для класса *Ball*, который был создан в главе 12:

```
// Добавить пользовательский метод toString()
// Создать конструктор Ball
function Ball (radius, color, xPosition, yPosition) {
  this.radius = radius;
  this.color = color;
  this.xPosition = xPosition;
  this.yPosition = yPosition;
}

// Присвоить литерал функции Ball методу toString() прототипа класса
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);  // Displays: "A ball with the radius 6"
```

***См. также***

*Array.toString( )*, *Date.toString( )*, *Number.toString( )*, *Object.valueOf( )*.

---

**Object.valueOf( ) метод** элементарное значение объекта

***Доступность*** Flash 5

***Краткий обзор*** *someObject*.valueOf()

***Возвращаемое значение***

Внутренне определенное элементарное значение *someObject*, если с *someObject* связано элементарное значение. Если элементарного значения с *someObject* не связано, то возвращается сам *someObject*.

***Описание***

Метод *valueOf( )* возвращает элементарный элемент данных, связанный с объектом, если такая связь существует. Этот метод чаще всего используется с объектами классов *Number*, *String* и *Boolean*, для всех из которых есть связанные с ними элементарные данные. Метод *MovieClip.valueOf( )* возвращает строку, представляющую путь к клипу.

Заметьте, что необходимость в явном вызове *valueOf( )* возникает редко: он автоматически вызывается интерпретатором при использовании *someObject* там, где предполагается элементарный тип данных.

***См. также***

*Boolean.valueOf( )*, *MovieClip.valueOf( )*, *Number.valueOf( )*, *Object.toString( )*, *String.valueOf( )*.

**parseFloat( ) глобальная функция** извлечь из строки число с плавающей точкой

***Доступность*** Flash 5

***Краткий обзор*** `parseFloat(`*`stringExpression`*`)`

***Аргументы***

*stringExpression* Строка, из которой должно быть извлечено число с плавающей точкой.

***Возвращаемое значение***

Извлеченное число с плавающей точкой. Или, если извлечение оказалось неудачным, специальное числовое значение `NaN`.

***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; в противном случае, возвращается `NaN`. Строка должна иметь следующий формат:

- Необязательный ведущий пробельный символ
- Необязательный символ знака + или –
- Хотя бы одна из цифр 0–9 и необязательная десятичная точка
- Необязательный показатель степени, начинающийся с `e` или `E`, за которыми следует целочисленный показатель степени.

***Особенности применения***

Поскольку данные, вводимые пользователем в текстовые поля, всегда принадлежат типу данных *string*, функция *parseFloat( )* часто используется для извлечения числовых данных из текста, вводимого пользователем. Обратите внимание на то, что *parseFloat( )* может извлекать числа из строк, которые содержат как числа, так и нечисловые символы, в то время как *Number( )* этого не может.

***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( )*; раздел «Явное преобразование типа данных» в главе 3 и раздел «Целые числа и числа с плавающей точкой» в главе 4.

---

**parseInt( ) глобальная функция** извлечь из строки целое число или преобразовать число к десятичному

***Доступность*** Flash 5

***Краткий обзор*** `parseInt(`*`stringExpression`*`)`
`parseInt(`*`stringExpression, radix`*`)`

***Аргументы***

*stringExpression* Строка, из которой должно быть извлечено целое число.

*radix* Необязательное целое число между 2 и 36, задающее основание системы счисления извлекаемого целого числа. Если этот аргумент не задан, то основание, используемое по умолчанию, зависит от содержимого *stringExpression* (как описано ниже).

***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 действует только со строками, содержащими допустимое строковое представление целого числа с использованием указанного основания системы счисления. Строка должна иметь следующий формат:

- Необязательный ведущий пробельный символ
- Необязательный + или –
- Одна или более цифр, допустимых в системе счисления с указанным основанием

Замыкающие символы, которые не удовлетворяют предшествующему числовому формату, игнорируются.

Число, полученное из переданной строки, начинается с первого символа строки, отличного от пробела, и оканчивается символом перед первым символом, не являющимся допустимой цифрой в заданном *radix*. Например, в десятичной системе 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* являются допустимыми числами, поэтому следующее выражение возвращает число 47:

```
parseInt("2F", 16);  // шестнадцатеричное 2F дает 47 по основанию 10
```

Аргумент *radix* функции of *parseInt( )* указывает основание числа, которое существует в строке. Иными словами, с помощью аргумента *radix* можно приказать интерпретатору обработать строку как число с основанием 16 или обработать строку как число с основанием 2.

*parseInt( )* также интерпретирует префикс `0x`, указывающий на шестнадцатеричное число (как если бы был задан *radix* 16), и ведущий 0, указывающий на восьмеричное число (как если бы был указан *radix* 8):

```
parseInt("0xFF");   // Анализируется как 16-е, дает 255
```

```
parseInt("FF", 16); // Анализируется как 16-е, дает 255
parseInt("0377");   // Анализируется как 8-е, дает 255 = (3*64)+(7*8)+(7*1)
parseInt("377", 8);  // Анализируется как 8-е, 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( )* используется 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( )*; разделы «Явное преобразование типа данных» в главе 3 и «Целые числа и числа с плавающей точкой» в главе 4.

---

**play( ) глобальная функция** — начать последовательное воспроизведение кадров фильма

***Доступность*** Flash 2 и выше

***Краткий обзор*** `play()`

***Описание***

При вызове функции *play( )* начинается последовательное воспроизведение кадров текущего главного фильма или клипа. «Текущим» называется фильм или клип, в котором находится предложение с вызовом функции *play( )*. Кадры воспроизводятся с частотой, измеряемой в кадрах в секунду (FPS) и задаваемой для фильма в целом в свойствах фильма Movie Properties (меню Modify → Movie → Frame Rate).

Начавшееся воспроизведение фильма или клипа продолжается, пока его специально не остановит вызов другой функции. Все клипы фильмов зацикливаются (снова начинают воспроизведение с кадра 1), когда воспроизводящая головка 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 используется команда Publish, атрибут `LOOP` устанавливается при выборе меню File → Publish Settings → HTML → Playback → Loop.)

***См. также***

*gotoAndPlay( )*, *MovieClip.play( )*, *stop( ).*

---

**prevFrame( ) глобальная функция** переместить воспроизводящую головку фильма на один кадр назад и остановить ее

***Доступность*** Flash 2 и выше

***Краткий обзор*** `prevFrame()`

***Описание***

Функция *prevFrame( )* перемещает воспроизводящую головку текущей временной диаграммы назад на один кадр и останавливает ее там. «Текущей временной диаграммой» является та, из которой вызвана функция *prevFrame( )*. Функция *prevFrame( )* является синонимом для *gotoAndStop( _currentFrame - 1);*. При вызове из первого кадра временной диаграммы *prevFrame( )* просто останавливает в этом кадре воспроизводящую головку, если только текущей сцене не предшествует другая, и тогда *prevFrame( )* перемещает воспроизводящую головку в последний кадр предыдущей сцены.

***См. также***

*gotoAndStop( )*, *MovieClip.prevFrame( )*, *nextFrame( )*, *prevScene( ).*

---

**prevScene( ) глобальная функция** переместить воспроизводящую головку в кадр 1 предыдущей сцены

***Доступность*** Flash 2 и выше

***Краткий обзор*** `prevScene()`

***Описание***

Функция *prevScene( )* перемещает главную воспроизводящую головку фильма в кадр 1 сцены, предшествующей текущей, и останавливает главную временную диаграмму фильма. «Текущей» является сцена, 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 фильма или обработчика события *onClipEvent( )*. При вызове из первой сцены фильма *prevScene( )* перемещает воспроизводящую головку в кадр 1 этой сцены и останавливает воспроизведение фильма.

***См. также***

*nextScene( )*, *prevFrame( ).*

---

**print( ) глобальная функция** вывод на печать кадров фильма или клипа с использованием векторной графики

***Доступность*** Flash 5

***Краткий обзор*** `print(target, boundingBox)`

***Аргументы***

*target* Строка или ссылка, указывающая путь к клипу фильма или уровню документа, которые должны быть напечатаны (ссылки преобразуются в пути при использовании в строковом контексте).

*boundingBox* Строка, указывающая способ обрезания кадров *target* при выводе на печать. Обрезание определяется ограничивающей рамкой, которая представляет всю печатную 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, чтобы соответствовать размеру содержимого каждого кадра. Поэтому содержимое каждого печатаемого кадра масштабируется, чтобы заполнить всю печатную страницу.

*«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.

*«bmovie»*

Рамка границ для всех печатаемых кадров устанавливается по размеру одного выделенного кадра в клипе *target*. Содержимое печатаемых кадров обрезается по рамке границ этого выделенного кадра. Чтобы выделить кадр как рамку границ, ему присваивается метка `#b`.

***Описание***

Печать фильма Flash с помощью встроенной функции печати броузера происходит не единообразно и часто дает отпечатки низкого качества. С помощью функции *print( )* можно печатать содержимое 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 их в соответствии с аргументом *boundingBox*. Для печати избранных кадров им присваивается метка `#P`.

Функция *print( )* посылает на PostScript-принтеры непосредственно векторную графику, а на принтеры без поддержки PostScript – графику, преобразованную в растровые изображения. Так как *print( )* использует векторы, с ее помощью нельзя печатать фильмы с преобразованиями прозрачности или цвета. Для печати фильмов с цветовыми эффектами используется функция *printAsBitmap( )*.

***Особенности применения***

Во Flash 4 r20 и выше возможности различных функций *print( )* Flash 5 доступны в виде модифицированного действия *getURL( )* Action. За подробностями обращайтесь к Macromedia Flash Printing SDK, имеющемуся по адресу:

*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 со следующим кодом Flash выводит на печать все кадры на временной диаграмме кнопки, обрезанные по рамке границ кадра с меткой `#b` и с размером на всю страницу:

```
on (release) {
  print(this, "bmovie");
}
```

***См. также***

*getURL( )*, *printAsBitmap( )*, *printAsBitmapNum( )*, *printNum( )*.

---

**printAsBitmap( ) глобальная функция** — печатает кадры фильма или клипа как растровые изображения

***Доступность*** Flash 5

***Краткий обзор*** `printAsBitmap(`*target*, *boundingBox*`)`

***Аргументы***

*target* — Строка или ссылка, указывающая путь к клипу фильма или уровню документа, которые должны быть напечатаны (ссылки преобразуются в пути при использовании в строковом контексте).

*boundingBox* — Строка, указывающая способ обрезания кадров *target* при выводе на печать, как описано выше для *print( )*.

***Описание***

Функция *printAsBitmap( )* по своему действию идентична *print( )*, за исключением того, что выводит не векторы, а растрированное содержимое. В результате она может успешно печатать фильмы с преобразованием цвета, но дает результаты более низкого качества для векторной графики.

***Особенности применения***

См. замечания по применению функции *print( )*.

***См. также***

*print( )*, *printAsBitmapNum( )*, *printNum( )*.

---

**printAsBitmapNum( ) глобальная функция** печать кадров уровня документа как растровых изображений

***Доступность*** Flash 5

***Краткий обзор*** `printAsBitmapNum(level, boundingBox)`

***Аргументы***

*level* Неотрицательное целое число или выражение, дающее неотрицательное целое число Задает уровень документа, который должен быть напечатан.

*boundingBox* Строка, указывающая способ обрезания кадров *target* при выводе на печать, как описано выше для *print( )*.

***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 фильмов. Обычно она используется, когда нужно динамически назначить выводимый на печать уровень фильма, например:

```
var x = 3;
printAsBitmapNum(x, "bmax");
```

Того же самого можно достичь с помощью конкатенации строк и использования обычной функции *printAsBitmap( )*:

```
printAsBitmap("_level" + x, "bmax");
```

***Особенности применения***

См. замечания по применению функции *print( )*.

***См. также***

*print( )*, *printAsBitmap( )*, *printNum( )*.

---

**printNum( ) глобальная функция** печать кадров уровня документа с использованием векторов

***Доступность*** Flash 5

***Краткий обзор*** `printNum(level, boundingBox)`

***Аргументы***

*level* Неотрицательное целое число или выражение, дающее неотрицательное целое число, указывающее уровень документа, который должен быть выведен на печать.

*boundingBox* Строка, указывающая способ обрезания кадров *target* при выводе на печать, как описано выше для *print( )*.

***Описание***

Функция *printNum( )* почти идентична *print( )*, но требует задания целевого уровня *level* для операции печати в виде числа, а не строки. Это означает, что *printNum( )* может печатать только уровни документов, а не клипы фильмов. Обычно она используется, когда нужно динамически назначить выводимый на печать уровень фильма, например:

```
var x = 3;
printNum(x, "bmax");
```

Того же самого можно достичь с помощью конкатенации строк и использования обычной функции *print( )*:

```
print("_level" + x, "bmax");
```

***Особенности применения***

См. замечания по применению функции *print( )*.

***См. также***

*print( )*, *printAsBitmap( )*, *printAsBitmapNum( )*.

---

**_quality глобальное свойство** качество вывода в проигрывателе

***Доступность*** Flash 5

***Краткий обзор*** _quality

***Тип доступа*** Чтение/запись

***Описание***

Свойство _quality содержит строку, определяющую качество вывода Flash Player следующим образом:

*«LOW»* Низкое качество. Не производится сглаживания (antialiasing) ни для растровой, ни для векторной графики.

*«MEDIUM»* Среднее качество. Умеренное сглаживание векторной графики.

*«HIGH»* Высокое качество. Векторная графика сглаживается в значительной степени. Растровая графика сглаживается, когда не происходит анимации.

*«BEST»* Высшее качество. Всегда производится сглаживание как растровой, так и векторной графики.

Низкое качество вывода означает, что для рисования кадра или фильма требуется меньше вычислений, и это приводит к повышению скорости воспроизведения. Обычно в большинстве фильмов _quality устанавливается в "HIGH".

***Пример***

В данном примере устанавливается наивысшее возможное качество вывода фильма (что также влечет самое медленное воспроизведение):

```
_quality = "BEST";
```

***См. также***

_highQuality, *toggleHighQuality( )*.

---

**random( ) глобальная функция** генерировать случайное число

***Доступность*** Flash 4; устарела в 5 благодаря появлению *Math.random( )*

***Краткий обзор*** random(*number*)

***Аргументы***

*number* Положительное целое число.

***Возвращаемое значение***

Случайное целое число, большее или равное 0 и меньшее *number*.

***Описание***

Устаревшая функция *random( )* использовалась во Flash 4 для генерации случайных чисел. Эта функция является устаревшей и присутствует только в целях обратной совместимости. Во Flash 5 и выше предпочтительнее пользоваться методом *Math.random( )*. Обратите внимание, что *random( )* генерировала целые числа от 0 до number −1, в то время как *Math.random( )* генерирует числа с плавающей точкой от 0.0 до .999.

***См. также***

*Math.random( )*.

---

**removeMovieClip( ) глобальная функция** удалить клип фильма из Flash Player

***Доступность*** Flash 4; усилена во Flash 5 возможностью применения к экземплярам, создаваемым с помощью *attachMovie( )*

***Краткий обзор*** removeMovieClip(*target*)

***Аргументы***

*target* Строка или ссылка, задающие путь к экземпляру клипа, который должен быть удален из проигрывателя (ссылки преобразуются в пути, если используются в строковом контексте).

***Описание***

Функция *removeMovieClip( )* удаляет указанный клип фильма из проигрывателя, не оставляя следов содержимого или оболочки клипа. Последующие ссылки на клип или его переменные и свойства возвращают undefined.

Функцию *removeMovieClip( )* можно использовать только с экземплярами клипов, созданными первоначально посредством *duplicateMovieClip( )* или *attachMovie( )*. На клипы фильмов, созданные в среде разработки, влияния не оказывает.

***См. также***

*attachMovie( )*, *duplicateMovieClip( )*, *MovieClip( ).removeMovieClip( )*; раздел «Удаление экземпляров клипов и главных фильмов» в главе 13.

---

**_root глобальное свойство** ссылка на главную временную диаграмму фильма на текущем уровне

***Доступность*** Flash 5 (то же, что "/" в фильмах Flash 4)

***Краткий обзор*** `_root`

***Тип доступа*** Только для чтения

***Описание***

Свойство `_root` хранит ссылку на главную временную диаграмму текущего уровня документа. Свойство `_root` используется 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
```

Свойством `_root` можно пользоваться и при ссылках на вложенные клипы, 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.

***См. также***

`_level`***n***`;` `_parent;` разделы «Ссылка на главный фильм с помощью _root и _leveln» в главе 13 и «Доступ к переменным других временных диаграмм» в главе 2.

---

**scroll свойство** текущая верхняя строка, выводимая в текстовом поле

***Доступность*** Flash 4 и выше

***Краткий обзор*** *`textfield`*`.scroll`

***Возвращаемое значение***

Положительное целое число, представляющее номер самой верхней видимой строки текстового поля.

***Описание***

Свойство `scroll` текстового поля можно как читать, так и задавать. При извлечении значения свойства `scroll` текстового поля получается номер строки, выводимой в данный момент в качестве первой строки видимой области поля. При задании значения `scroll` происходит прокрутка текстового поля, чтобы сделать переданное число номером верхней строки в видимой области поля. Обычно свойство `scroll` используется вместе с `maxscroll`

для создания интерфейсов с прокруткой текста, как описывается в разделе «Типичный код для прокрутки текста» главы 18.

***Особенности применения***

Хотя `scroll` числится во Flash как функция, в сущности, это свойство переменной текстового поля. Обратите внимание, что при вызове `scroll` не используются круглые скобки.

***Ошибки в реализации***

В сборке Flash Player Build 5.0 r30 при встраивании шрифта текстового поля использование `scroll` может повлечь вывод текста за пределами видимой области поля. Иногда текст может не удаляться при прокрутке поля. Для решения этой проблемы используйте маску над слоем текстового поля. Ошибка была исправлена в Build 5.0 r41.

***Пример***

```
// Sets x to the index of the top line displayed in myField
var x = myField.scroll;
// Scrolls the text in myField down one
myField.scroll++;
```

***См. также***

`maxscroll`; раздел «Свойство scroll» в главе 18.

---

**Selection объект** управление выделением в текстовых полях

***Доступность*** Flash 5

***Краткий обзор*** `Selection.methodName()`

***Методы***

| | |
|---|---|
| *getBeginIndex( )* | Получить индекс первого выбранного символа. |
| *getCaretIndex( )* | Получить индекс точки ввода. |
| *getEndIndex( )* | Получить индекс последнего выбранного символа. |
| *getFocus( )* | Указать текстовое поле, в котором в данное время находится точка ввода. |
| *setFocus( )* | Поместить точку ввода в указанное текстовое поле. |
| *setSelection( )* | Выделить символы в текстовом поле, имеющем в данный момент фокус. |

***Описание***

Объект *Selection* используется для управления взаимодействием пользователя с текстовыми полями и захвата участков текстового поля. На практике методам объекта *Selection* предшествует ключевое слово `Selection`; они всегда действуют над текстовым полем, имеющем фокус, которое может быть только одно в каждый данный момент. Методы объекта *Selection* могут устанавливать точку ввода и выбирать или извлекать значение содержимого текстового поля. Например, в экране регистрации пользователя можно предложить пользователю ввести свое имя, переместив курсор в

текстовое поле ввода имени. Либо можно выделить ошибку в форме, выделив спорный текст. Можно также настроить порядок табуляции ряда текстовых полей, как показано в описании метода *Selection.setFocus( )*.

Позиции символов в текстовом поле относятся к индексам, нумеруемым с нуля, когда первый символ имеет индекс 0, второй имеет индекс 1, и так далее. Индексирование символов подробно описывается в разделе «Индекс символов» в главе 4.

Во Flash 5 нельзя вырезать, копировать или вставлять текст программным образом. Вторая кнопка мыши (щелчок правой кнопкой в Windows, Ctrl-щелчок в Macintosh) обеспечивает доступ к командам «вырезать», «копировать» и «вставить».

***Особенности применения***

При щелчке кнопка передачи автоматически получает фокус от любого текстового поля, имевшего фокус перед этим. Для перехвата выделения перед потерей фокуса используется событие кнопки rollover. Например:

```
on (rollOver) {
  focusedField = Selection.getFocus();
}
```

***См. также***

`_focusrect`, *Selection.setFocus( )*; главу 18 «Экранные текстовые поля».

---

**Selection.getBeginIndex( ) метод** получить индекс первого выбранного в текстовом поле символа

***Доступность*** Flash 5

***Краткий обзор*** `Selection.getBeginIndex()`

***Возвращаемое значение***

Индекс первого символа в текущем выделении (выделенном блоке текста). Если никакой текст не имеет фокуса ввода с клавиатуры, функция возвращает −1. Если текст содержит фокус, но никакие символы не выбраны, возвращается значение *Selection.getCaretIndex( )*.

***Описание***

Метод *getBeginIndex( )* определяет начало выделения. Чтобы определить диапазон символов, выбранных в данное время, нужно использовать две функции: *getBeginIndex( )* и *getEndIndex( )*.

***Пример***

В следующем примере создается строка, представляющая выделенные в данный момент символы, которая выводится в окно Output:

```
var firstChar = Selection.getBeginIndex();
var lastChar  = Selection.getEndIndex();
var currentSelection = eval(Selection.getFocus()).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( )*.

---

**Selection.getCaretIndex( ) метод** получить индекс точки ввода

***Доступность*** Flash 5

***Краткий обзор*** `Selection.getCaretIndex()`

***Возвращаемое значение***

Индекс точки ввода в текущем текстовом поле. Если фокуса ввода не имеет ни одно текстовое поле, возвращается –1. Если текстовое поле с фокусом пусто, возвращается 0.

***Описание***

Метод *getCaretIndex( )* показывает точку ввода (т. е. местонахождение курсора) в текстовом поле. Когда текстовое поле получает фокус ввода с клавиатуры, курсор выглядит как буква I. Для установки положения точки ввода используйте *setSelection( )*.

***Пример***

Поскольку *getCaretIndex( )* возвращает –1, когда ни ни на одном текстовом поле не стоит фокус, можно узнать, есть ли фокус у какого-либо поля, проверив *getCaretIndex( )* на равенство –1, например:

```
if (Selection.getCaretIndex() == -1) {
  trace("No field has focus");
}
```

***См. также***

*Selection.setSelection( )*.

---

**Selection.getEndIndex( ) метод** получить индекс последнего выделенного символа

***Доступность*** Flash 5

***Краткий обзор*** `Selection.getEndIndex()`

***Возвращаемое значение***

Индекс символа, располагающегося за последним символом в текущем выделении (подсвеченном блоке текста). Если фокуса не стоит ни в одном тексте, возвращается –1. Если текст находится в фокусе, но не выделены никакие символы, возвращается значение *Selection.getCaretIndex( )*.

***Описание***

Метод *getEndIndex( )* указывает на конец выделения. Чтобы определить диапазон символов, которые выбраны в данный момент, нужно использовать 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** — определить текстовое поле, в котором в данный момент находится курсор

***Доступность*** Flash 5

***Краткий обзор*** `Selection.getFocus()`

***Возвращаемое значение***

Строка, указывающая полный путь к переменной текстового поля, имеющего фокус ввода с клавиатуры (т. е. того, в котором в данный момент находится курсор), например, `"_level1.myTextField"`. Если ни у одного текстового поля нет фокуса ввода с клавиатуры, возвращается `null`.

***Описание***

Метод *getFocus( )* определяет текстовое поле, в котором в настоящее время находится фокус (если такое есть), возвращая строку пути к нему. Преобразовать эту строку в ссылку на переменную можно с помощью *eval( )*. Например, в следующем коде определяется число символов в текстовом поле, имеющем фокус. Имя поля получается в результате вызова *getFocus( )*, и с помощью *eval( )* оно преобразуется в переменную

```
var numChars = eval(Selection.getFocus()).length;
```

***Пример***

Так как *getFocus( )* возвращает `null`, если ни одно текстовое поле не выбрано, то можно определить, есть ли фокус у какого-либо поля, проверив *getFocus( )* на равенство `null`, например:

```
if (Selection.getFocus() == null) {
  trace("No field has focus");
}
```

***См. также***

*Selection.setFocus( )*, *eval( )*, *Selection.getCaretIndex( )*.

---

**Selection.setFocus( ) метод** — установить фокус клавиатуры в определенное текстовое поле

***Доступность*** Flash 5

***Краткий обзор*** `Selection.setFocus(`*variableName*`)`

***Аргументы***

*variableName* — Строка, представляющая путь (например, `"_root.myTextField"` или `"userName"`) к переменной текстового поля, которая должна получить фокус.

### *Возвращаемое значение*

Булева величина, показывающая, была ли попытка установить фокус удачной (`true`) или безуспешной (`false`). Попытка установки фокуса может быть неудачной, только если переменная, указанная в *`variableName`*, не найдена, или она не является переменной текстового поля.

### *Описание*

Метод *setFocus( )* устанавливает фокус ввода с клавиатуры на текстовое поле. Он помещает в это текстовое поле курсор, обычно для того, 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 сделать исправления. С помощью *setFocus( )* можно также задавать пользовательский порядок перемещения по полям формы с помощью клавиши табуляции, как показывается в следующем примере.

### *Особенности применения*

Обратите внимание, что при установке фокуса в текстовое поле его содержимое автоматически выделяется. Чтобы установить фокус, не выделяя символы, воспользуйтесь таким кодом:

```
// Сначала установить фокус в myField
Selection.setFocus("myField");

// Теперь поместить точку ввода в начало myField
Selection.setSelection(0, 0);

// Или в конец myField
Selection.setSelection(myField.length, myField.length);
```

При просмотре фильма в броузере текстовое поле, имеющее фокус, принимает текст только тогда, когда сам фильм Flash получает фокус (т. е. если пользователь щелкнул по фильму в какой-то точке во время воспроизведения). Сделайте так, чтобы фильм получил фокус, прежде чем просить пользователя ввести что-либо в текстовые поля. Один из способов – поместить в начале фильма кнопку с надписью «Щелкнуть здесь, чтобы начать».

### *Пример*

В следующем примере показано, как назначить свой порядок для полей в заполняемой форме. Соответствующий файл *.fla* можно загрузить из сетевого хранилища кода (дополнительные сведения о перехвате клавиши Tab можно найти в разделе «Обработка специальных клавиш» главы 10 «События и обработчики событий»:

```
// Назначение порядка перехода по клавише табуляции
// КОД В КЛИПЕ ФИЛЬМА, СОДЕРЖАЩЕГО 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.
  tabOrder = ["field1", "field3", "field2", "field4"];
}
onClipEvent (keyUp) {
  // Если нажата клавиша Tab...
  if (Key.getCode() == Key.TAB) {
    // ...Если ни у какого поля нет фокуса...
    if (Selection.getFocus() == null) {
      // ...то установить фокус на первое поле, указанное в массиве
      Selection.setFocus(path + "." + tabOrder[0]);
    } else {
      // В противном случае 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)) {
        // Shift нажата, поэтому перейти к предыдущему полю, если
        // текущее поле не первое, иначе перейти в последнее поле .
       nextFocus = currentFocus-1 == -1 ? tabOrder.length-1 : currentFocus-1;
      } else {
        // Shift не нажата, поэтому перейти к следующему полю, если
        // текущее поле не последнее: тогда перейти в первое.
        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***

*Selection.getFocus( )*, *Selection.setSelection( )*.

---

**Selection.setSelection( ) метод** — выделить символы в текстовом поле, имеющем фокус, или установить точку ввода

***Доступность*** Flash 5

***Краткий обзор*** `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 (подсвечивает) 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 используется для подсвечивания введенных пользователем данных, с которыми есть проблемы.

***Особенности применения***

Хотя у объекта *Selection* нет метода *«setCaretIndex»* («установить индекс возврата каретки»), с помощью метода *setSelection( )* можно установить точку ввода в определенное место текстового поля. Для этого нужно 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( )*.

---

**setсвойство( ) глобальная функция** — присвоить значение свойству клипа фильма

***Доступность*** Flash 4 и выше

***Краткий обзор*** `setProperty(movieClip, property, value)`

***Аргументы***

*movieClip* Выражение, возвращающее строку, указывающую путь к клипу фильма. Во Flash 5 это может быть и ссылкой на клип фильма, поскольку ссылки на клипы преобразуются в пути при использовании в строковом контексте.

*property* Имя встроенного свойства, которому должно быть присвоено *value*. Должно быть идентификатором, а не строкой (т. е. `_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* (встроенные свойства перечислены в описании класса *MovieClip*). Ее нельзя использовать для установки свойств, определенных пользователем. Во Flash 4 *setProperty( )* была единственным способом присвоения значений свойствам клипа; начиная с Flash 5 наиболее предпочтительными способами задания 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» в главе 13; приложение С.

---

**Sound класс** управление звуком фильма

***Доступность*** Flash 5

***Конструктор***

```
new Sound()
new Sound(target)
```

***Аргументы***

*target* Строка, задающая путь к клипу или уровню документа, звуком которого нужно управлять. Может также являться ссылкой на клип или уровень документа (ссылки преобразуются в пути при использовании в строковом контексте).

***Методы***

*attachSound( )* Связать звук из библиотеки с объектом *Sound*.

*getPan( )* Извлечь текущую установку панорамирования.

*getTransform( )* Определить текущее распределение каналов по левому и правому динамикам (т. е. баланс).

| | |
|---|---|
| *getVolume( )* | Извлечь текущую установку громкости. |
| *setPan( )* | Установить панорамирование звука по левому и правому каналам. |
| *setTransform( )* | Распределить каналы по левому и правому динамикам (т. е. установить баланс). |
| *setVolume( )* | Установить громкость. |
| *start( )* | Начать воспроизведение текущего звука. |
| *stop( )* | Заглушить все звуки или указанный прикрепленный звук. |

***Описание***

Объекты класса *Sound* используются для управления текущими звуками фильма или звуками, программно добавленными в фильм. У объектов *Sound* есть несколько различных применений. Они могут управлять:

- Всеми звуками Flash Player
- Всеми звуками конкретного экземпляра клипа или главного фильма (в том числе всеми звуками во вложенных клипах)
- Отдельным программно прикрепленным звуком

Чтобы создать объект *Sound*, управляющий всеми звуками в Player (в том числе звуками в файлах *.swf* на уровнях документов), используйте конструктор *Sound* без параметров. Например:

```
myGlobalSound = new Sound();
```

Чтобы создать объект *Sound*, управляющий звуками конкретного клипа или главного фильма, задайте параметр *target*, указывающий клип или фильм, которыми нужно управлять. Обратите внимание, что при этом также контролируются звуки в клипах, находящихся внутри *target*. Например:

```
spaceshipSound = new Sound("spaceship");  // Control sounds in spaceship clip
mainSound      = new Sound("_root");      // Control sounds on main timeline
```

Чтобы 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( )*.

***См. также***

*stopAllSounds( )*; _soundbuftime.

---

**Sound.attachSound( ) метод** связать звук из библиотеки с объектом Sound

***Доступность*** Flash 5

***Краткий обзор*** *soundObject*.attachSound(*linkageIdentifier*)

***Аргументы***

*linkageIdentifier* Имя прикрепляемого звука, заданное в библиотеке в пункте Options → Linkage.

***Описание***

Метод *attachSound( )* на этапе выполнения добавляет в фильм новый звук, находящийся под управлением *soundObject*. Когда звук прикреплен, его можно отдельно запускать и останавливать, вызывая методы *start( )* и *stop( )* над *soundObject*.

Чтобы прикрепить звук к *soundObject*, его нужно экспортировать из библиотеки фильма. Для экспорта звука нужно выполнить следующие шаги:

1. Выбрать в библиотеке экспортируемый звук.
2. Выбрать меню Options → Linkage. Появляется диалоговое окно Symbol Linkage Properties.
3. Выбрать Export This Symbol.
4. В окне Identifier ввести уникальное имя, идентифицирующее звук.

Обратите внимание, что *все* экспортированные звуки загружаются в первом кадре содержащего их фильма (а 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 можно достичь, если поместить их во внешние файлы *.swf* и импортировать с помощью *loadMovie( )* по мере необходимости.

***Особенности применения***

В каждый конкретный момент к объекту *Sound* может быть прикреплен только один звук. Прикрепление нового звука к объекту *Sound* заменяет звук, прикреплявшийся к этому объекту ранее. Обратите внимание, что *attachSound( )* не будет работать в фильмах, загруженных в клип или на уровень документа посредством *loadMovie( )*, если прикрепляемого звука нет в библиотеке документа, являющегося областью видимости объекта Sound. Для глобальных звуковых объектов, создаваемых без параметра *target*, областью видимости является _level0.

***Пример***

В следующем примере звук с идентификатором phaser добавляется к объекту *Sound* с именем *phaserSound*. Затем звук phaser запускается и останавливается:

```
phaserSound = new Sound();
phaserSound.attachSound("phaser");

// Запустить звук phaser
phaserSound.start();

// Остановить только звук phaser
phaserSound.stop("phaser");
```

***См. также***

*Sound.start( )*, *Sound.stop( )*.

**Sound.getPan( ) метод** получить последнее заданное значение панорамы

***Доступность*** Flash 5

***Краткий обзор*** *soundObject*.getPan()

***Возвращаемое значение***

Число, представляющее собой 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 это значение равно 0 (левый и правый канал включены в равной пропорции).

***Описание***

Изменяя панораму звука, можно создавать иллюзию движущегося источника звука. Метод *getPan( )* используется для определения текущего распределения звука по левому и правому каналам, которым управляет *soundObject*. Обычно *getPan( )* используется вместе с *setPan( )* для настройки текущей панорамы звука.

***Пример***

Следующий код изменяет панораму звука на 20:

```
mySound = new Sound();
mySound.setPan(mySound.getPan() - 20);
```

***См. также***

*Sound.getTransform( )*, *Sound.setPan( )*.

---

**Sound.getTransform( ) метод** определить текущее распределение каналов звука для левого и правого динамиков

***Доступность*** Flash 5

***Краткий обзор*** *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 по левому и правому динамикам. Возвращаемый объект имеет свойства `ll`, `lr`, `rl` и `rr`, описываемые в методе *Sound.setTransform( )*.

***См. также***

*Sound.getPan( )*, *Sound.setTransform( )*.

**Sound.getVolume( ) метод** получить текущую установку громкости

***Доступность*** Flash 5

***Краткий обзор*** *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( )* используется для определения текущей громкости звука, управляемой *soundObject*. Обычно *getVolume( )* используется вместе с *setVolume( )* для настройки текущей громкости звука.

***Пример***

В этом примере громкость звука уменьшается на 20 единиц:

```
mySound = new Sound();
mySound.setVolume(mySound.getVolume() - 20);
```

***См. также***

*Sound.setVolume( )*.

---

**Sound.setPan( ) метод** установить баланс звука между левым и правым каналами

***Доступность*** Flash 5

***Краткий обзор*** *soundObject*.setPan(*pan*)

***Аргументы***

*pan* Число от −100 (левый) до 100 (правый), указывающее распределение между левым и правым динамиками для звуков, управляемых *soundObject*. Если передаваемое значение *pan* больше 100, фактически присваиваемым значением будет 200 - *pan*. Если передаваемое значение *pan* меньше −100, 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 одного динамика в другой (что называют *панорамированием – panning*).

Чтобы воспроизводить звуки, управляемые *soundObject*, только в левом динамике, используется значение *pan*, равное −100. Чтобы воспроизводить звуки, управляемые *soundObject*, только в правом динамике, используется значение *pan*, равное 100. Для равной балансировки обоих каналов используется значение *pan*, равное 0.

Обратите внимание, что *setPan( )* действует на все звуки, управляемые *soundObject*. Если *soundObject* является глобальным звуком, *setPan( )* действует на все звуки фильма. Если *soundObject* привязан к клипу или к главной временной диаграмме, *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*.

***Пример***

Следующие обработчики событий клипов вызывают беспрерывное изменение баланса звуков клипа между левым и правым динамиками:

```
onClipEvent (load) {
  panEffect = new Sound(this);
  panDirection = "right";
  panIncrement = 50;
}

onClipEvent(enterFrame) {
  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);
    }
  }
}
```

Следующие обработчики событий клипов заставляют звуки клипов реагировать на мышь. Предполагается, что ширина и высота рабочего стола равны 550 и 400 пикселам. Баланс звука смещается влево или вправо при горизонтальном перемещении мыши, а при вертикальном перемещении мыши увеличивается или уменьшается громкость:

```
onClipEvent (load) {
  // Создать новый объект Sound и прикрепить к нему звук bgMusic
  mySound = new Sound(this);
  mySound.attachSound("bgMusic");
  mySound.start(0, 999);                // Воспроизводить звук в цикле
}

onClipEvent (enterFrame) {
```

```
    // Получить горизонтальное положение мыши и установить соответствующий баланс
    mouseX = (_root._xmouse / 550) * 200;
    mySound.setPan(mouseX - 100);
    // Получить вертикальное положение мыши и установить соответствующую громкость
    mouseY = (_root._ymouse / 400) * 300;
    mySound.setVolume(300 - mouseY);
  }
```

***См. также***

*Sound.getPan( ).*

---

**Sound.setTransform( ) метод** — распределить левый и правый каналы между левым и правым динамиками

***Доступность*** Flash 5

***Краткий обзор*** `soundObject.setTransform(`*`transformObject`*`)`

***Аргументы***

*transformObject* Определяемый пользователем объект, задающий новые настройки каналов в виде ряда свойств.

***Описание***

Метод *setTransform( )* предоставляет точный контроль над выводом звуковых каналов на левый и правый динамики. В принципе, *setTransform( )* имеет сходство с *setPan( )*, но обеспечивает более детальное управление стереозвучанием.

Стереозвук является комбинацией двух отдельных звуков, левого канала и правого канала, которые обычно отдельно посылаются на левый и правый динамики. Однако с помощью *setTransform( )* можно задать, какая часть каждого канала воспроизводится через каждый динамик. Например, можно сказать: «Играть половину левого канала в левом динамике, нулевую часть левого канала в правом динамике и весь правый канал в обоих динамиках». Либо потребовать: «Играть полностью левый и правый каналы в левом динамике».

Для использования *setTransform( )* необходимо сначала создать объект с рядом предопределенных свойств. Свойства описывают распределение левого и правого каналов стереозвука между левым и правым динамиками, как описано в табл. 12.

*Таблица 12. Свойства transformObject*

| Имя свойства | Значение свойства | Описание свойства |
|---|---|---|
| `ll` | от 0 до 100 | Процент левого канала, воспроизводимый в левом динамике |
| `lr` | от 0 до 100 | Процент правого канала, воспроизводимый в левом динамике |
| `rl` | от 0 до 100 | Процент левого канала, воспроизводимый в правом динамике |

*Таблица 12 (продолжение)*

| Имя свойства | Значение свойства | Описание свойства |
|---|---|---|
| rr | от 0 до 100 | Процент правого канала, воспроизводимый в правом динамике |

Объект, созданный со свойствами, которые описаны в табл. 12, передается методу *setTransform( )* объекта *Sound*. Значения свойств *transformObject* становятся новыми процентными отношениями выходных мощностей каналов для звуков, которые управляются через *soundObject*.

Чтобы получить текущие значения процентных отношений для некоторого объекта *Sound*, используется метод *Sound.getTransform( )*.

***Пример***

```
// Создать новый объект Sound
mySound = new Sound();

// Создать новый родовой объект для использования с setTransform()
transformer = new Object();

// Установить свойства объекта преобразования
transformer.ll = 0;      // Отсутствие левого канала в левом динамике
transformer.lr = 0;      // Отсутствие правого канала в левом динамике
transformer.rl = 0;      // Отсутствие левого канала в правом динамике
transformer.rr = 100;    // Весь правый канал в правом динамике

// Применить новое распределение каналов путем передачи объекта
// преобразования методу setTransform()
mySound.setTransform(transformer);
```

***См. также***

*Sound.getTransform( ), Sound.setPan( )*.

---

**Sound.setVolume( ) метод** — установить громкость звуков, управляемых объектом *Sound*

***Доступность*** Flash 5

***Краткий обзор*** *soundObject*.setVolume(*volume*)

***Аргументы***

*volume* — Число, характеризующее громкость звука, управляемого *soundObject*, причем 0 означает отсутствие звука. 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. Значением громкости по умолчанию является 100.

***Описание***

Метод *setVolume( )* делает звуки, управляемые *soundObject*, громче или тише. Чтобы вообще убрать звук, нужно установить громкость в 0. Чтобы

сделать звук громче, нужно увеличить значение *volume*. Значения в диапазоне 100–200 обычно дают очень большую громкость, а максимального значения нет.

Обратите внимание, что *setVolume( )* действует на все звуки, управляемые *soundObject*. Если *soundObject* является глобальным звуком, *setVolume( )* действует на все звуки фильма. Если *soundObject* привязан к клипу или главной временной диаграмме, *setVolume( )*действует на все звуки в этом клипе или временной диаграмме.

Действие *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:

```
var mySound = new Sound();
mySound.setVolume (65);
```

В следующем примере показано, как делать кнопки для настройки уровня громкости фильма:

```
// КОД НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ ФИЛЬМА
var globalSound = new Sound();
var maxVolume = 200;
var minVolume = 0;
var volumeIncrement = 20;

// КОД НА КНОПКЕ УВЕЛИЧЕНИЯ ГРОМКОСТИ НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
on (release) {
  globalSound.setVolume(Math.min(globalSound.getVolume() + volumeIncrement,
                                 maxVolume));
}

// КОД НА КНОПКЕ УМЕНЬШЕНИЯ ГРОМКОСТИ НА ГЛАВНОЙ ВРЕМЕННОЙ ДИАГРАММЕ
on (release) {
  globalSound.setVolume(Math.max(globalSound.getVolume() - volumeIncrement,
                                 minVolume));
}
```

***См. также***

*Sound.getVolume( ).*

---

**Sound.start( ) метод** начать воспроизведение прикрепленного звука

***Доступность*** Flash 5

***Краткий обзор*** *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 *точ-*

*кой входа – entry point*). Например, *secondOffset* со значением 1 начинает воспроизведение на одну секунду после фактического начала звука, как оно определено в библиотеке. Значением по умолчанию является 0. Точку выхода (т. е. время прекращения воспроизведения звука) задать нельзя. Звук воспроизводится до конца или до остановки вручную.

*loops* Положительное целое число, определяющее количество воспроизведений звука, прикрепленного к *soundObject*. Чтобы воспроизвести звук один раз, указывается 1 (значение по умолчанию), чтобы воспроизвести звук два раза подряд, задается 2, и так далее. Часть звука, начиная с *secondOffset*, повторяется до конца столько раз, сколько указано в *loops*.

***Описание***

Метод *start( )* используется для воспроизведения программно определяемых звуков, добавленных к *soundObject* посредством *attachSound( )*. Метод *start( )* не воспроизводит все звуки клипа или фильма – он воспроизводит только звук, последним добавленный в *soundObject* с помощью *attachSound( )*.

Для воспроизведения только части звука, добавленного в *soundObject*, задается аргумент *secondOffset*. Для многократного воспроизведения звука, прикрепленного к *soundObject*, используется аргумент *loops*.

***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, начиная с 0.5 секунды после начала;
// по умолчанию loops равно 1
boink.start(.5);

// Воспроизвести boink три раза, с начала и до конца
boink.start(0, 3);
```

***См. также***

*Sound.stop( )*.

---

**Sound.stop( ) метод** прекращает воспроизведение всех звуков или заданного прикрепленного звука

***Доступность*** Flash 5

***Краткий обзор***

```
soundObject.stop()
soundObject.stop(linkageIdentifier)
```

***Arguments***

*linkageIdentifier* Имя любого звука, прикрепленного к любому объекту *Sound* с таким же *target*, как *soundObject*. Идентификаторы связи задаются в библиотеке через меню Options → Linkage.

***Описание***

При вызове без *linkageIdentifier* метод *stop( )* прекращает все звуки, управляемые *soundObject*; если *soundObject* является глобальным звуком, *stop( )* прекращает воспроизведение всех звуков в фильме; если *soundObject* был создан с параметром *target*, метод *stop( )* прекращает воспроизведение всех звуков в *target*.

При вызове с *linkageIdentifier* метод *stop( )* прекращает воспроизведение только конкретного звука, заданного *linkageIdentifier*. 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* был создан без задания *target* (т. е. 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.start( )*.

**_soundbuftime глобальное свойство** длительность потокового звука для предварительной загрузки в секундах

***Доступность*** Flash 4 и выше

***Краткий обзор*** `_soundbuftime`

***Тип доступа*** Чтение/запись

***Описание***

Свойство `_soundbuftime` является целым числом, задающим длительность в секундах потокового звука, который должен быть загружен до начала воспроизведения. По умолчанию значение этого свойства составляет 5 секунд.

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 длительного значения может приводить к чрезмерной задержке начала при низкоскоростном соединении. Поскольку поток в сети может оказаться медленным или кратковременно прерываться, задание короткого `_soundbuftime` может приводить к пропускам в звуке (если недостаточно звуковых данных загружено в буфер). 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 действует хорошо, но в отдельных случаях для нахождения наилучшей настройки могут потребоваться эксперименты. Объем буфера потока может меняться во время воспроизведения, но это свойство является глобальным и не может задаваться отдельно для разных звуков.

***Пример***

```
_soundbuftime = 10;  // Буферизовать 10 секунд аудиоданных
```

---

**startDrag( ) глобальная функция** заставить фильм или клип следовать за курсором мыши

***Доступность*** Flash 4 и выше

***Краткий обзор***

```
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 (ссылки на клипы преобразуются в пути при использовании в строковом контексте).

*lockCenter* Булева величина, определяющая, должна ли точка регистрации *target* помещаться в центре указателя мыши (`true`) или перемещаться относительно исходного положения (`false`).

| | |
|---|---|
| *left* | Число, задающее минимальную x-координату, левее которой не может перемещаться точка регистрации *target*. |
| *top* | Число, задающее минимальную y-координату, выше которой не может перемещаться точка регистрации *target*. |
| *right* | Число, задающее максимальную x-координату, правее которой не может перемещаться точка регистрации *target*. |
| *bottom* | Число, задающее максимальную y- координату, ниже которой не может перемещаться точка регистрации *target*. |

***Описание***

Функция *startDrag( )* вызывает зрительное перемещение *target* за курсором мыши по проигрывателю (называемое также буксировкой (*dragging*) клипа). Перемещение буксируемого клипа можно ограничить рамкой, координаты которой передаются в качестве аргументов функции *startDrag( )*. Координаты ограничивающей рамки задаются относительно холста, на котором расположен *target*. Если холстом является рабочий стол главного фильма, то точка (0, 0) является левым верхним углом рабочего стола. Если холстом является клип фильма, то (0, 0) обозначает точку регистрации холста клипа. Обратите внимание, что координатная система Flash переворачивает декартову ось Y: значения y *возрастают* в направлении к нижней части экрана и *убывают* в направлении верхней части экрана. Отрицательные значения y находятся *выше* начала координат (т. е. над осью X).

Буксировку можно прекратить в любой момент с помощью функции *stopDrag( )*. В каждый конкретный момент 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 рабочего стола
startDrag("ball", true, 0, 0, 225, 200);
```

***См. также***

*Movieclip.startDrag( )*, *stopDrag( )*.

---

**stop( ) глобальная функция** остановить воспроизведение фильма на текущем кадре

***Доступность*** Flash 2 и выше

***Краткий обзор***

```
stop()
```

***Описание***

Функция *stop( )* – это простая, но базовая функция, останавливающая воспроизведение фильма или клипа. Это глобальный аналог метода *Movie-*

*Clip.stop( )*. Она часто используется для ожидания, когда пользователь, скажем, сделает выбор в графическом меню.

***См. также***

*MovieClip.stop( )*, *play( )*.

---

**stopAllSounds( ) глобальная функция** выключить звук в фильме

***Доступность*** Flash 3 и выше

***Краткий обзор*** `stopAllSounds()`

***Описание***

Функция *stopAllSounds( )* заглушает все звуки, воспроизводящиеся в данное время в фильме, независимо от глубины вложенности в клипы. Она применяется ко всем звукам фильма, в том числе к программно генерируемым объектам *Sound*. Более точно управлять остановкой, запуском и громкостью звука можно с помощью класса *Sound*.

Обратите внимание, что *stopAllSounds( )* оказывает только временное воздействие. Любой звук, который начинается после вызова *stopAllSounds( )*, будет воспроизводиться нормально. Способ навсегда прекратить воспроизведение звуков в фильме отсутствует.

***См. также***

*Sound.setVolume( )*, *Sound.stop( )*.

---

**stopDrag( ) глобальная функция** завершить текущую операцию буксировки

***Доступность*** Flash 4 и выше

***Краткий обзор*** `stopDrag()`

***Описание***

Функция *startDrag( )* вызывает перемещение клипа за курсором мыши по рабочему столу. Операция *stopDrag( )* прекращает перемещение клипа вслед за курсором мыши. Поскольку в каждый конкретный момент можно буксировать только один клип или фильм, для *stopDrag( )* не требуется аргумент *target* – эта функция просто прекращает любую текущую операцию буксировки.

Вместе с *startDrag( )* функция *stopDrag( )* используется для создания во Flash простых интерфейсов «drag-and-drop», которые демонстрируются в разделе «Interface Widgets» (элементы интерфейса) сетевого хранилища кода.

***Пример***

Следующий код вызывает буксировку клипа, когда кнопка нажата, и отпускает его, когда кнопка отпущена:

```
on (press) {
  startDrag("", true);
```

```
}

on (release) {
  stopDrag();
}
```

***См. также***

*MovieClip.stopDrag( )*, *startDrag( )*, *String.toLowerCase( )*; раздел «Функция toUpperCase()» в главе 4.

---

**String( ) глобальная функция** — преобразовать значение к типу данных *String*

***Доступность*** Flash 5

***Краткий обзор*** `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. Обычно необходимости в использовании функции *String( )* нет: если нужно, 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.

***Особенности применения***

Обратите внимание на возможное появление функции *String( )* в файлах *.fla* Flash 4, которые преобразованы в формат Flash 5. Сведения об обращении с типами данных при преобразовании файлов Flash 4 в файлы Flash 5 можно найти в разделе «Преобразование типов данных Flash 4 во Flash 5» главы 3.

***См. также***

The *String* class; раздел «Явное преобразование типов» в главе 3.

---

**String класс** — класс-оболочка для элементарного строкового типа

***Доступность*** Flash 5

***Конструктор*** `new String(value)`

***Аргументы***

| | |
|---|---|
| *value* | Выражение, которое должно быть вычислено 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( )* | Генерировать строку из одной или нескольких кодовых позиций Latin 1/Shift-JIS. |

***Методы***

Следующие методы объекта вызываются через экземпляр класса *String*:

| | |
|---|---|
| *charAt( )* | Получить символ в заданной позиции строки. |
| *charCodeAt( )* | Получить кодовую позицию символа в заданной позиции строки. |
| *concat( )* | Объединить один или более элементов в единую строку. |
| *indexOf( )* | Найти первое вхождение указанной подстроки в строку. |
| *lastIndexOf( )* | Найти последнее вхождение указанной подстроки в строку. |
| *slice( )* | Извлечь из строки подстроку, исходя из положительных или отрицательных позиций символов. |
| *split( )* | Преобразовать строку в массив. |
| *substr( )* | Извлечь из строки подстроку, исходя из начальной позиции и длины. |
| *substring( )* | Извлечь из строки подстроку, исходя только из положительных позиций символов. |
| *toLowerCase( )* | Возвратить версию строки, записанную буквами нижнего регистра. |
| *toUpperCase( )* | Возвратить версию строки, записанную буквами верхнего регистра. |

***Описание***

Класс *String* решает несколько задач:

- Он 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.
- Он может использоваться для преобразования в строку элемента данных любого типа.
- Он используется для доступа к методу класса *fromCharCode( )* при создании новой строки, исходя из заданных кодовых позиций Latin 1 или Shift-JIS.

- Он может использоваться для создания объекта *String*, который содержит элементарное строковое значение безымянного внутреннего свойства; однако в этом редко может возникнуть необходимость.

***Особенности применения***

На практике конструктор класса *String* используется, в основном, для преобразования в строки других типов данных. Подробнее см. в описании глобальной функции *String( )*.

***См. также***

раздел «Строковый тип данных» в главе 4 «Элементарные типы данных».

---

**String.charAt( ) метод** получить символ в указанной позиции строки

***Доступность*** Flash 5

***Краткий обзор*** *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( )*; раздел «Функция charAt» главы 4.

**String.charCodeAt( ) метод** возвратить кодовую позицию символа на заданной позиции строки

***Доступность*** Flash 5

***Краткий обзор*** *string*.charCodeAt(*index*)

***Аргументы***

*index* Целочисленная позиция символа в строке, которая должна находиться в диапазоне от 0 (первый символ) до *string*.length-1 (последний символ).

***Возвращаемое значение***

Целое число, представляющее кодовую позицию в Latin 1 или Shift-JIS (см. приложение B) символа в позиции *index* строки *string*.

***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( )*; приложение B, раздел «Функция charCodeAt» главы 4.

---

**String.concat( ) метод** объединить одно или несколько значений в единую строку

***Доступность*** Flash 5

***Краткий обзор*** *string*.concat(*value1*, *value2*, ...*valuen*)

***Аргументы***

*value1,...valuen* Значения, которые должны быть преобразованы в строки (если это необходимо) и конкатенированы со *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 оказывается предпочтительнее для ясности кода, так как оператор + может использоваться и для сложения чисел. Подробнее об операторе *concat( )* см. в главе 4.

***Особенности применения***

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***

+ оператор в главе 5, раздел «Функция concat( )» в главе 4.

---

**String.fromCharCode( ) метод класса** генерировать строку из одной или нескольких кодовых позиций

***Доступность*** Flash 5

***Краткий обзор*** `String.fromCharCode(`*code_point1, code_point2,...code_pointn*`)`

***Аргументы***

*code_point1,...code_pointn*

Последовательность из одного или нескольких десятичных целых чисел, соответствующих кодовым позициям символов в Latin 1 или Shift-JIS, показанным в приложении B «Набор символов Latin 1 и коды клавиш».

***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.

***Пример***

```
// Создает символ copyright, за которым идет год (2001)
copyNotice = String.fromCharCode(169) + " 2001";
```

***См. также***

*String.charCodeAt( )*; приложение B, раздел «Функция fromCharCode( )» в главе 4.

---

**String.indexOf( ) метод** найти первое вхождение подстроки в строку

***Доступность*** Flash 5

***Краткий обзор*** *string*`.indexOf(`*substring*`)`
*string*`.indexOf(`*substring, startIndex*`)`

***Аргументы***

*substring* Строка, содержащая символ или символы, которые нужно найти.

*startIndex* Необязательное целое число, задающее позицию в *string*, с которой должен начинаться поиск *substring*. Должно находиться в диапазоне от 0 (первый символ) до *string*.length-1 (последний символ). По умолчанию равно 0.

***Возвращаемое значение***

Позиция первого вхождения *substring* в *string* (начиная с *startIndex*). Возвращает -1, если *substring* не найдена в *string* ни в позиции *startIndex*, ни далее.

***Описание***

Метод *indexOf( )* используется для поиска в строке символа или проверки того, что строка содержит определенную подстроку.

***Пример***

```
// Проверить, что адрес электронной почты содержит символ @
var email = "derekaol.com";
if (email.indexOf("@") == -1) {
  trace ("This isn't a valid email address");
}

// Проверить, что адрес электронной почты содержит символ @ и относится
// к домену domain aol.com
var email = "derek@aol.com";
var atPos = email.indexOf("@");
if (atPos != -1 && email.indexOf("aol.com") == atPos + 1) {
  gotoAndStop("AOLuserOffer");
}
```

В следующем коде показана общая функция, которая ищет в строке ключевое слово 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( )*; раздел «Функция indexOf()» в главе 4.

---

**String.lastIndexOf( ) метод** — найти последнее вхождение подстроки в строку

***Доступность*** Flash 5

***Краткий обзор*** `string.lastIndexOf(substring)`
`string.lastIndexOf(substring, startIndex)`

***Аргументы***

*substring* Строка, содержащая символ или символы, которые нужно найти.

*startIndex* Необязательное целое число, задающее позицию в `string`, с которой должен начинаться 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 равно `string.length-1`.

***Возвращаемое значение***

Позиция первого вхождения `substring` в `string` до `startIndex`. Возвращает –1, если `substring` не найдена в `string` в позиции до `startIndex`.

***Описание***

Метод *lastIndexOf( )* используется для поиска последнего вхождения подстроки в строку или проверки того, что строка содержит определенную подстроку.

***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( )*; раздел «Функция lastIndexOf()» в главе 4.

---

**String.length свойство** — количество символов в строке

***Доступность*** Flash 5

***Краткий обзор*** `string.length`

***Тип доступа*** Только для чтения

***Описание***

Свойство `length` возвращает количество символов в *string*. Обратите внимание, что нулевой символ (ASCII 0) не обозначает конца строки, как в некоторых языках, 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( )*; раздел «Свойство length» в главе 4.

---

**String.slice( ) метод** — извлечь из строки подстроку, определяемую положительными или отрицательными позициями символов

***Доступность*** Flash 5

***Краткий обзор*** *string*.slice(*startIndex*, *endIndex*)

***Аргументы***

*startIndex* — Целое число, задающее позицию первого символа, извлекаемого из *string*. Если *startIndex* отрицателен, позиция отсчитывается от конца строки, причем −1 является последним символом, −2 является предпоследним символом, и так далее. (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 −1 является последним символом, −2 является предпоследним символом, 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( )* является одним из трех методов, которые можно использовать для извлечения подстроки из строки (другие два – *substring( )* и *substr( )*). Метод *slice( )* позволяет указывать отрицательные значения начального и конечного индексов, что дает возможность извлекать подстроку, отсчитывая от конца строки.

***Особенности применения***

Заметьте, что *slice( )* не модифицирует *string* – он 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( )*; разделы «Функция slice()» и «Сочетание исследования строки с извлечением подстроки» в главе 4.

---

**String.split( ) метод** — преобразовать строку в последовательность элементов массива

***Доступность*** Flash 5

***Краткий обзор*** *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, разделенные *delimiter*.

***Описание***

Метод *split( )* разбивает строку на подстроки, присваивает эти подстроки элементам массива и возвращает этот массив. Смежные положения *delimiter* без промежуточных символов 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 undefined):

```
0: terry
1: doug
2:
3:
4: jon
```

Метод *split( )* обычно используется 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. Пример кода см. в разделе «Вызов функций ActionScript из ссылок HTML» главы 18 «Экранные текстовые поля». Обычно в качестве разделителей используются запятая или символ табуляции.

### *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. раздел «data» в главе 10 «События и обработчики событий»), 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 при обработке больших участков текста выполнение *split( )* может занять много времени. Если скорость работы оказывается неудовлетворительной, попробуйте разбить данные на меньшие порции или воспользоваться XML. См. *XML*.

### *Ошибки в реализации*

Во Flash 5 при использовании в качестве разделителя пустой строки вся *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 целиком *string* первому элементу возвращаемого массива.

### *См. также*

*Array.join( )*; раздел «Функция split()» в главе 4.

**String.substr( ) метод** — извлечь из строки подстроку, заданную начальной позицией и длиной

***Доступность*** Flash 5

***Краткий обзор*** *string*.substr(*startIndex*, *length*)

***Аргументы***

*startIndex* — Целое число, задающее позицию первого символа, извлекаемого из *string*. Если *startIndex* отрицателен, позиция отсчитывается от конца строки, причем −1 является последним символом, −2 является предпоследним символом, и так далее. (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( )* является одним из трех методов, которые можно использовать для извлечения подстроки из строки (два других – *slice( )* и *substring( )*). Метод *substr( )* извлекает строку, исходя из количества символов, заданного *length*, а не по двум индексам символов.

***Особенности применения***

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( )*; разделы «Функция substr()» и «Сочетание исследования строки с извлечением подстроки» в главе 4.

**String.substring( ) метод** извлечь из строки подстроку, определяемую положительными позициями символов

***Доступность*** Flash 5

***Краткий обзор*** *string*.substring(*startIndex*, *endIndex*)

***Аргументы***

*startIndex* Положительное целое число, задающее позицию первого символа, извлекаемого из *string*. Вместо отрицательного числа используется 0.

*endIndex* Положительное целое число, задающее позицию символа *за* последним символом, извлекаемым из *string*. По умолчанию имеет значение *string*.length. Вместо отрицательного числа используется 0.

***Возвращаемое значение***

Подстрока *string*, начинающаяся с *startIndex* и заканчивающаяся *endIndex*-1, где *startIndex* и *endIndex* отсчитываются с нуля.

***Описание***

Метод *substring( )* является одним из трех методов, которые можно использовать для извлечения подстроки из строки (другими являются *slice( )* и *substr( )*). Функция *substring( )* идентична *slice( )*, но не допускает отрицательные значения *startIndex* и *endIndex*, и автоматически меняет местами эти два индекса, если *endIndex* меньше *startIndex*.

***Особенности применения***

Заметьте, что *substring( )* не модифицирует *string* – он 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 следующем примере приведена функция, которую можно повторно использовать для поиска и замены всех вхождений подстроки в строку:

```
// 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( )*; разделы «Сочетание исследования строки с извлечением подстроки» и «Функция substr()» в главе 4.

---

**String.toLowerCase( ) метод** — создать версию строки, записанную буквами нижнего регистра

***Доступность*** Flash 5

***Краткий обзор*** *string*.toLowerCase()

***Возвращаемое значение***

Новая строка – эквивалент *string* на нижнем регистре. Символы, не имеющие эквивалента в нижнем регистре, остаются без изменения.

***Описание***

Метод *toLowerCase( )* создает новую версию *string* на нижнем регистре; его можно использовать при форматировании или нечувствительном к регистру сравнении символов. Метод *toLowerCase( )* преобразует только символы из диапазона A–Z (он не преобразует символы с диакритическими знаками, такими как акценты или умляуты).

***Особенности применения***

Заметьте, что *toLowerCase( )* не модифицирует *string* – он 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( )*; раздел «Функция toLowerCase()» в главе 4.

---

**String.toUpperCase( ) метод** создать версию строки, записанную буквами верхнего регистра

***Доступность*** Flash 5

***Краткий обзор*** `string.toUpperCase()`

***Возвращаемое значение***

Новая строка – 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. Метод *toUpperCase( )* преобразует только символы из диапазона a–z (он не преобразует символы с диакритическими знаками, такими как акценты или умляуты).

***Особенности применения***

Заметьте, что *toUpperCase( )* не модифицирует *string* – он 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( )*; раздел «Функция toUpperCase()» в главе 4.

---

**targetPath( ) глобальная функция** абсолютный путь к фильму или клипу

***Доступность*** Flash 5

***Краткий обзор*** `targetPath (movieClip)`

***Аргументы***

*movieClip* ссылка на объект клипа.

***Возвращаемое значение***

Строка, представляющая путь к *movieClip* в абсолютном виде с использованием точечной нотации (например, `"_level0.myMovie"`).

***Описание***

Функция *targetPath( )* возвращает ссылку на клип фильма в виде строки, описывающей абсолютный путь к клипу (идентичной значению, возвращаемому *MovieClip.valueOf( )*). Функция *targetPath( )* иногда применяет-

ся при написании учитывающего положение кода, работающего с клипом относительно временной диаграммы, на которой он находится.

***Ошибки в реализации***

Обратите внимание, что пример кода для *targetPath( )*, приведенный во Flash 5 ActionScript Dictionary, показывает неправильное применение *targetPath( )*; в противоположность тому, что подразумевалось в примере, это не синоним для *tellTarget( )*.

***Пример***

Если `square` является клипом фильма, содержащимся в `shapes`, находящемся на главной временной диаграмме уровня 0, то изнутри клипа `shapes` предложение

```
targetPath(square);
```

возвратит:

```
"_level0.shapes.square"
```

***См. также***

*`MovieClip`*.`_target`, *MovieClip.valueOf( )*; раздел «Функция targetPath()» в главе 13.

---

**tellTarget( ) глобальная функция** — выполнить предложения, находящиеся в области видимости удаленного клипа фильма

***Доступность*** Flash 3 и Flash 4; устарела во Flash 5 благодаря объектно-ориентированному синтаксису предложения *with*

***Краткий обзор***

```
tellTarget (target) {
        statements
}
```

***Аргументы***

*target* — Строка или ссылка, указывающие путь к фильму или экземпляру клипа (ссылки преобразуются в пути при использовании в строковом контексте).

*statements* — Предложения, которые должны быть выполнены в области видимости *target*.

***Описание***

Во Flash 3 и Flash 4 *tellTarget( )* была главным средством связи между двумя клипами фильмов (т. е. управления одним клипом из другого). Она использовалась для вызова таких функций, как *play( )*, *stop( )* и *gotoAndStop( )* с удаленными клипами фильмов. Во Flash 4, где в ActionScript были добавлены переменные, с помощью *tellTarget( )* можно было также получать и задавать значения переменных удаленных клипов. Во Flash 5 такие действия лучше осуществлять с помощью оператора точки, ., и оператора обращения к массиву, `[]`. Другой альтернативой функции *tellTarget( )* является предложение *with*, описанное в главе 6 «Предложения».

***Особенности применения***

Функцию *tellTarget( )* лучше было бы описать как предложение, поскольку для нее требуется блок подпредложений. Вопрос этот, однако, чисто академический, поскольку *tellTarget( )* устарела.

***Пример***

```
tellTarget ("ball") {
  gotoAndStop("redStripes");
  _x += 300;
}
```

***См. также***

разделы «Предложение with» в главе 6, «Использовать ли Tell Target?» и ««Объектность» клипов» в главе 13.

---

**toggleHighQuality( ) глобальная функция** изменить качество вывода проигрывателя

***Доступность*** Flash 2 и выше; устарела во Flash 5 благодаря появлению `_quality`

***Краткий обзор*** `toggleHighQuality()`

***Описание***

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: она просто переключает две возможные установки – «High» и «Low». Это может вызвать проблемы, поскольку качество не устанавливается явно в известное состояние, и при этом не допускается иных установок качества, кроме двух заданных.

Во Flash 5 *toggleHighQuality( )* устарела благодаря появлению глобального свойства `_quality`, которое поддерживает качество вывода Low, Medium, High и Best.

***См. также***

`_highquality`, `_quality`.

---

**trace( ) глобальная функция** вывести значение в окно Output

***Доступность*** Flash 4 и выше

***Краткий обзор*** `trace(`*value*`)`

***Аргументы***

*value* Выражение, которое должно быть вычислено и затем выведено в окно Output. Если вычисленное значение *value* не является строкой, то оно преобразуется в строку перед отправкой в окно Output согласно правилам, описанным в табл. 3.2.

### *Описание*

Функция *trace( )* является инструментом отладки, используемым только в отладочном режиме Test Movie среды разработки Flash. Скромная по природе, *trace( )* фактически является одним из базовых компонентов программирования на ActionScript: она позволяет проверить значение переменной или выражения в любой точке во время воспроизведения фильма.

### *Особенности применения*

К сожалению, *trace( )* может оказаться весьма медлительной. Отключить трассировку можно с помощью отключения в меню File→Publish Settings→Flash опции Omit Trace Actions.

### *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*

главу 19 «Отладка».

---

**unescape( ) глобальная функция** раскодировать строку с escape-последовательностями

***Доступность*** Flash 5

***Краткий обзор*** `unescape(stringExpression)`

### *Аргументы*

*stringExpression* Строка (или выражение, разрешаемое в строку), ранее закодированное с помощью *escape( )*.

### *Возвращаемое значение*

Новая строка, являющаяся раскодированной версией `string`.

### *Описание*

Функция *unescape( )* возвращает новую строку, основанную на `string`. Новая строка содержит эквивалентные символы в кодировке Latin 1 для каждой двузначной шестнадцатеричной escape-последовательности, имеющей в `string` префикс %. Функции *escape( )* и *unescape( )* используются при кодировании и раскодировании строк для более надежной передачи по сети. Однако использовать *unescape( )* вручную приходится редко, поскольку Flash автоматически преобразует текст в кодировке URL при импорте его через *loadVariables( )*.

### *Пример*

```
var msg = "hello!";
// Установить msgCoded равной "hello%21"
msgCoded = escape(msg);
// Установить msgDecoded равной"hello!"
var msgDecoded = unescape(msgCoded);
```

***См. также***

*escape( )*; приложение B «Набор символов Latin 1 и коды клавиш».

---

**unloadMovie( ) глобальная функция** удалить фильм или клип фильма из проигрывателя

***Доступность*** Flash 4 и выше (функция Flash 5 *unloadMovie( )* соответствует функции Flash 4 *Unload Movie* с заданием пути к цели)

***Краткий обзор*** `unloadMovie(target)`

***Аргументы***

*target* Строка или ссылка, указывающие путь к клипу фильма или уровню документа, которые нужно удалить из проигрывателя (ссылки преобразуются в пути при использовании в строковом контексте).

***Описание***

Функция *unloadMovie( )* чаще всего используется для удаления фильмов с уровней документа в проигрывателе. Например, если фильм загружен на уровень 1 проигрывателя, удалить его из проигрывателя можно так:

```
unloadMovie("_level1");
```

Функция *unloadMovie( )* может использоваться с экземплярами клипов, и тогда она удаляет содержимое экземпляра, а не сам экземпляр. Экземпляр остается на рабочем столе в виде пустой оболочки, в которую можно загружать последующие фильмы. Поэтому один клип может выступать как контейнер динамического содержимого, управляемого последовательными вызовами *loadMovie( )* и *unloadMovie( )*.

***См. также***

*loadMovie( )*, *MovieClip.unloadMovie( )*, *unloadMovieNum( )*; разделы «Удаление экземпляров клипов и главных фильмов» и «Проблемы, связанные с перекрытием методов и глобальных функций» в главе 13.

---

**unloadMovieNum( ) глобальная функция** удалить фильм из уровня документа

***Доступность*** Flash 3 и выше (Функция Flash 5 *unloadMovieNum( )* соответствует функции Flash 3 *unload Movie*, которая действовала только с пронумерованными уровнями)

***Краткий обзор*** `unloadMovieNum(level)`

***Аргументы***

*level* Неотрицательное целое число или выражение, дающее неотрицательное целое число, которое указывает уровень документа, подлежащий выгрузке.

***Описание***

Функция *unloadMovieNum( )* почти идентична *unloadMovie( )*, за исключением того, что требует задания конечного уровня *level* как числа, а не как строки. Это означает, что *unloadMovieNum( )* может удалять только фильмы на уровнях документа, а не клипы. Обычно она применяется, если нужно динамически назначить уровень удаляемого фильма, например:

```
var x = 3;
unloadMovieNum(x);
```

Тот же результат может быть получен с помощью конкатенации строк и использования обычной функции *unloadMovie( )*:

```
unloadMovie("_level" + x);
```

***См. также***

*loadMovieNum( )*, *Movieclip.unloadMovie( )*, *unloadMovie( )*; раздел «Удаление экземпляров клипов и главных фильмов» в главе 13.

---

**updateAfterEvent( ) глобальная функция** — вывести содержимое рабочего стола в промежутке между кадрами

***Доступность*** Flash 5

***Краткий обзор*** `updateAfterEvent()`

***Описание***

Обработчики событий клипов, связанных с вводом пользователя (*mouseMove*, *mouseDown*, *mouseUp*, *keyDown* и *keyUp*), часто появляются в промежутке между прорисовкой кадров во Flash Player. Чтобы заставить изменения в изображении, происходящие в обработчике события клипа, связанного с пользовательским вводом, отразиться на экране, в любом из этих обработчиков вызывается *updateAfterEvent( )*. Обратите внимание, однако, что *updateAfterEvent( )* не является произвольным средством обновления экрана: она действует только в событиях клипов, связанных с вводом пользователя. Вне обработчиков *onClipEvent( )* она не оказывает никакого действия.

***Пример***

Следующий сценарий, прикрепленный к клипу, вызывает перемещение клипа за указателем мыши и обновляет экран при каждом перемещении указателя. Так как экран обновляется при каждом перемещении указателя, клип двигается за указателем мыши очень плавно:

```
onClipEvent (mouseMove) {
  _x = _root._xmouse;
  _y = _root._ymouse;
  updateAfterEvent();
}
```

***Ошибки в реализации***

Обратите внимание, что во Flash 5 ActionScript Dictionary неверно написано, будто функция *updateAfterEvent( )* принимает в качестве аргумента событие клипа. На самом деле, она не принимает никаких аргументов.

***См. также***

раздел «Обновление экрана с помощью updateAfterEvent» в главе 10.

---

**$version глобальное свойство** версия Flash Player

***Доступность*** Flash 4 Build 11 и выше; устарела во Flash 5 благодаря появлению *get Version*

***Краткий обзор*** `_root.$version`

***Тип доступа*** Только для чтения

***Описание***

Свойство `$version` содержит ту же строку, что возвращается глобальной функцией *getVersion( )* (операционную систему и информацию о версии проигрывателя). Свойство `$version` было введено посреди жизненного цикла Flash 4 Player и заменено функцией *getVersion( )*. Формально это не глобальное свойство, а свойство главной временной диаграммы: из временной диаграммы любого другого клипа оно должно вызываться как `_root.$version`.

***См. также***

*getVersion( )*.

---

**XML класс** основанная на DOM поддержка данных XML

***Доступность*** Flash 5

***Конструктор***

```
new XML()
new XML(source)
```

***Аргументы***

*source* Необязательная строка, содержащая корректные данные XML (или HTML), которые должны быть разобраны 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 (пробельные узлы).

| | |
|---|---|
| *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. |
| *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. |
| *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 в сценарий. |
| *sendAndLoad( )* | Отправить исходный код XML в сценарий и получить обратно исходный код XML. |
| *toString( )* | Преобразовать объект *XML* в строку. |

***Обработчики событий***

| | |
|---|---|
| *onData( )* | Обработчик, выполняемый при завершении загрузки источника XML. |
| *onLoad( )* | Обработчик, выполняемый, когда внешние данные XML разобраны в иерархию объектов. |

***Описание***

Объекты класса *XML* используются для обработки содержимого документа XML (или HTML) в объектно-ориентированном стиле и отправки данных в формате XML во Flash и из него. С помощью свойств и методов объекта *XML* можно создавать документ со структурой XML (или читать уже существующий), а также эффективно читать, изменять или удалять информацию в этом документе.

Исходный код документа XML состоит в основном из 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: 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. 1 показан этот простой фрагмент XML <BOOK>, мысленно представленный в виде иерархии объектов XML.

Рассмотрим слева направо структуру и семантику этого примера иерархии объектов XML. Начнем с главного объекта *XML*, показанного на рис. 1 как DOCUMENT, который автоматически создается конструктором *XML* и служит контейнером для иерархии объектов XML.

Перемещаясь по иерархии на один ряд вправо, приходим к BOOK, являющемуся первым элементом в данном фрагменте исходного кода XML и, в дан-

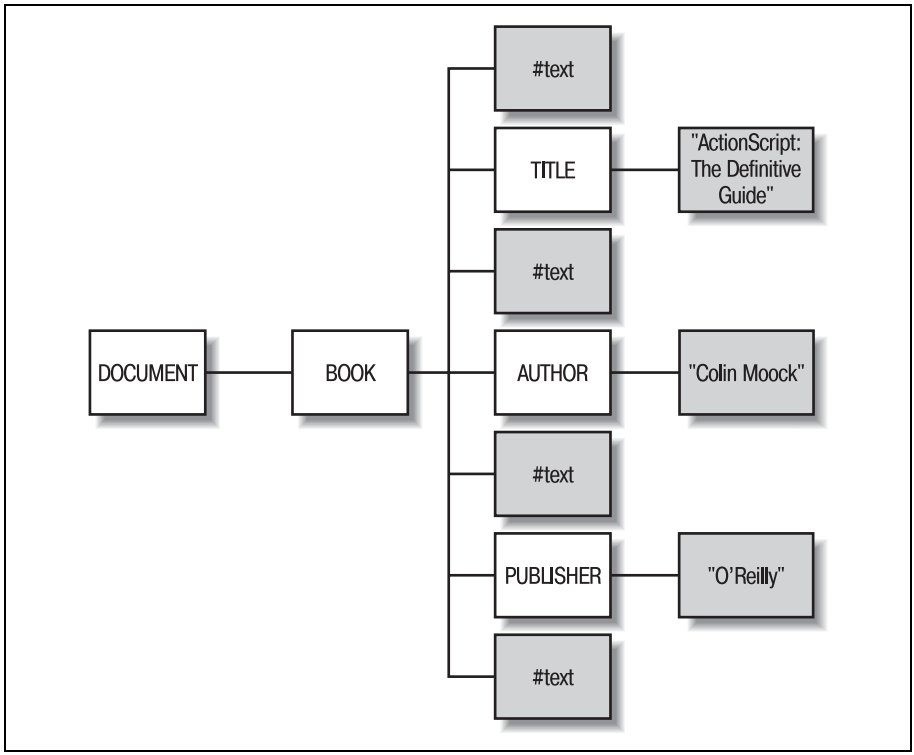


***Рис. 1.*** *Пример иерархии узлов XML*

ном случае, также первым *узлом объекта* в DOCUMENT. Узел BOOK является *корнем* этой структуры данных XML – у каждого корректного документа XML должен быть такой охватывающий все элемент, как BOOK, в котором содержатся все другие элементы. Ветви иерархии объектов XML добавляются к дереву в результате анализа исходного кода XML или вызова методов, добавляющих узлы, над объектами иерархии.

Когда один узел содержится в другом, он называется *дочерним* для содержащего его узла, а тот называется *родительским*. В данном примере BOOK является первым дочерним элементом DOCUMENT, а DOCUMENT является родителем BOOK.

Перемещаясь по рисунку вправо, обнаруживаем, что у BOOK семь дочерних элементов, в том числе четыре узла #text, которых не было видно в исходном документе XML. Каждый пробельный символ между элементами в исходном коде XML представляется объектом в иерархии объектов XML. Если внимательно посмотреть, то можно обнаружить пробельные символы – возврат каретки и символ табуляции – между BOOK и TITLE в предшествующем фрагменте XML. Этот пробел представляется на рис. 1 в виде узла #text. Аналогичные пробельные узлы есть после узлов TITLE, AUTHOR и PUBLISHER.

Дочерние элементы BOOK являются друг для друга братьями (т. е. находятся в иерархии на одном уровне). Например, следующим одноуровневым узлом (*next sibling*) для AUTHOR является #text, и предыдущим одноуровневым узлом (*previous sibling*) для AUTHOR является #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 удалять из иерархии объектов (примеры кода, удаляющего пробельные символы, см. ниже)
- Обнаруживать их и обходить стороной в коде (способы перехода через узлы предлагаются свойствами nextSibling и previousSibling)
- Просто удалять форматирующие пробельные символы в исходном тексте XML, что обеспечит отсутствие пробельных узлов
- Задавать свойство объекта *XML* ignoreWhite равным true, прежде чем анализировать источник XML (доступно начиная с Build 41 Flash 5 Player)

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 является текстовым узлом, соответствующим 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.

Рассмотрим, как построить документ XML в виде иерархии объектов узлов. Для создания нового пустого объекта *XML* используется 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 нужно получить текст «hello world!» в myDocument. Если думать на языке объектов, то можно предположить, что текст P извлекается как свойство myDocument, например: myDocument.P. На самом деле, это не действует: для ссылок на узлы вместо ссылок по имени используются встроенные свойства класса *XML*, например, firstChild и childNodes. Например, к узлу P можно обратиться так:

```
myDocument.firstChild       // Обращается к P
myDocument.childNodes[0]    // Тоже обращается к P
```

Так как 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 обращаться к текстовому узлу (т. е. первому потомку P) можно так:

```
myDocument.firstChild.firstChild  // Обращается к текстовому узлу под P
```

Для получения значения узла используется свойство nodeValue. Например, можно вывести значение «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";
```

Для полного удаления узла P, добавления нового узла или перемещения текста «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 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 ниже.

ActionScript строит обработку данных XML на основе стандарта модели объектов документа (Document Object Model – DOM), опубликованной World Wide Web Consortium (W3C). Полное описание того, как DOM представляет данные XML в виде иерархии объектов, можно найти по адресу:

*http://www.w3.org/DOM*

Подробности независимых от языка спецификаций базовой DOM можно найти по адресу:

*http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html*

(обратите особое внимание на раздел «Interface Node» (узел интерфейса) в 1.2, *Fundamental Interfaces*). О подробностях реализации DOM в ECMA-262 сказано по адресу:

*http://www.w3.org/TR/REC-DOM-Level-1/ecma-script-language-binding.html*

***Пример***

Как уже отмечалось, пробельные символы между любыми двумя элементами в исходном коде XML представляются текстовым узлом в соответствующей иерархии объектов *XML*. До выпуска Build 41 Flash 5 Player нежелательные пробельные узлы нужно было вручную вырезать из иерархии объектов XML. Задача удаления узлов определенного типа часто возникает при обработке XML и служит хорошим примером *обхода дерева* (т. е. прохода через каждый узел в иерархии). Рассмотрим два приема удаления пробельных узлов из документа.

В первом примере используется классический стек FIFO (First In First Out, «первым пришел – первым ушел») для помещения всех узлов дерева в массив для дальнейшей обработки. Функция *stripWhitespaceTraverse( )* порождает массив элементов узлов с помощью *theNode*, передаваемого ей в качестве аргумента. Затем выполняется цикл, в котором первый узел в массиве удаляется и 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 (т. е. только пробельные), удаляются:

```
// Удаление пробельных узлов с помощью стека FIFO
// Удаление всех пробельных узлов, являющихся потомками theNode,
// во время обхода дерева
function stripWhitespaceTraverse (theNode) {
  // Создать список узлов, подлежащий обработке
  var nodeList = new Array();
  // Породить список из узла, переданного функции
  nodeList[0] = theNode;

  // Обрабатывать потомков, пока больше не останется необработанных
  while (nodeList.length > 0) {
    // Удалить первый узел из списка и присвоить его currentNode
    currentNode = nodeList.shift();

    // 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. Если так, проверить, содержит ли он
      // только пробельные символы. nodeType 3 указывает на текстовый узел
         if (currentNode.nodeType == 3) {
        var i = 0;
        var emptyNode = true;
        for (i = 0; i < currentNode.nodeValue.length; i++) {
          // Полезным символом является все, что превосходит 32
          // (пробел, табуляция, перевод строки и т.п. находятся ниже).
          if (currentNode.nodeValue.charCodeAt(i) > 32) {
            emptyNode = false;
            break;
          }
        }
      }
```

```
      // Если полезных символов не найдено, удалить узел
      if (emptyNode) {
        currentNode.removeNode();
      }
    }
  }
}
```

**Прием, продемонстрированный в предшествующем примере, обычно является очень эффективным. Однако во Flash 5 Player метод *Array.concat()* выполняется довольно медленно. Поэтому быстрее пробелы удаляются с помощью приема, показанного 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 пр. находятся ниже)
        if (theNode.childNodes[i].nodeValue.charCodeAt(j) > 32) {
          emptyNode = false;
          break;
        }
      }

      // Если полезных символов не найдено, 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; раздел «Поддержка HTML» в главе 18.**

**XML.appendChild( ) метод** добавить к узлу новый дочерний узел, или переместить существующий

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.appendChild(*childNode*)

***Аргументы***

*childNode* Существующий объект узла XML.

***Описание***

Метод *appendChild( )* добавляет заданный *childNode* к *theNode* в качестве последнего дочернего узла для него. 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.

Чтобы добавить новый дочерний узел к уже существующему, нужно сначала создать новый дочерний узел с помощью *createElement( )*, *createTextNode( )* или *cloneNode( )*, либо в результате синтаксического разбора исходного кода XML в объект *XML*. Например, в следующем коде создаются 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>"
```

Чтобы переместить узел внутри документа, задайте в *childNode* 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]);
```

Чтобы переместить узел из одного документа в другой, *childNode* должен быть ссылкой на узел в первом (исходном) документе, а *theNode* должен быть 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.createElement( ), XML.createTextNode( ), XML.cloneNodee( ), XML.insertBefore( ).*

---

**XML.attributes свойство** — объект, в свойствах которого хранятся атрибуты элемента

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.attributes.*attributeIdentifier*
*theNode*.attributes[*attributeNameInQuotes*]

***Тип доступа*** Чтение/запись

***Описание***

Свойство attributes хранит имена и значения атрибутов, определенных для *theNode*. 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, 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);

// Установить атрибут ALIGN
myDoc.firstChild.attributes.ALIGN = "LEFT";
```

Свойство attributes само является объектом. К объекту attributes можно добавлять новые свойства, тем самым добавляя новые атрибуты в *theNode*, например:

```
// Добавить атрибут CLASS к тегу P
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). 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, attributes является пустым объектом, не имеющим свойств, и предыдущий пример покажет ноль атрибутов.

***См. также***

*XML*.nodeType.

---

**XML.childNodes свойство** массив ссылок на дочерние для данного узлы

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.childNodes[*n*]

***Тип доступа*** Только для чтения

***Описание***

Свойство childNodes является массивом, элементы которого содержат ссылки на непосредственных потомков *theNode*. Оно используется для доступа к узлам в иерархии XML. Например, создадим иерархию объектов следующим образом:

```
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 (являющимся потомками 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 с помощью следующего кода:

```
// 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 firstChild. Следующие ссылки совпадают:

```
myDoc.childNodes[0];
myDoc.firstChild;
```

Чтобы обойти все дочерние узлы для данного, можно использовать предложение *for*, например:

```
for (var i = 0; i < theNode.childNodes.length; i++) {
  trace("child " + i + " is " + theNode.childNodes[i].nodeName);
}
```

Однако в этом примере обходится только первый уровень иерархии *theNode*. Примеры для *XML*.nextSibling показывают, как обратиться ко всем узлам под *theNode*. Если у *theNode* нет дочерних узлов, *theNode*.childNodes.length равно 0.

***Особенности применения***

Помните, что пустые текстовые узлы, которые представляют пробельные символы, используемые 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: The Definitive Guide</TITLE>
  <AUTHOR SALUTATION="Mr">Colin Moock</AUTHOR>
  <PUBLISHER>O'reilly</PUBLISHER>
</BOOK>
```

Следовательно, первым дочерним узлом BOOK будет пустой текстовый узел; вторым дочерним узлом является TITLE.

***См. также***

*XML.firstChild, XML.hasChildNodes( ), XML.lastChild, XML.nextSibling, XML.previousSibling.*

---

**XML.cloneNode( ) метод** создать копию узла

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.cloneNode(*deep*)

***Аргументы***

*deep* — Булево значение, указывающее, должны ли дочерние узлы *theNode* рекурсивно участвовать в операции клонирования. В случае `true` клонируется вся иерархия, начиная с *theNode*. В случае `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 *theNode*, включая все атрибуты *theNode* и значения, если *theNode* является узлом элемента. Если *deep* имеет значение `true`, в возвращаемой копии содержится вся иерархия узлов, являющихся потомками *theNode*.

Метод *cloneNode( )* часто используется для создания нового узла, исходя из существующего шаблона (что избавляет от необходимости вручную создавать структуру нового узла). После клонирования узла он 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 *deep* надо присвоить значение `true`, чтобы сохранить при клонировании текстовое содержание элемента. Помните, что *cloneNode( )* не вставляет возвращаемый им элемент в документ узла – это нужно сделать самостоятельно с помощью *appendChild( )* или *insertBefore( )*.

***См. также***

*XML.appendChild( )*, *XML.createElement( )*, *XML.createTextNode( )*, *XML.insertBefore( )*.

---

**XML.contentType свойство** тип содержимого MIME для данных XML, пересылаемых с помощью *XML.send( )* и *XML.sendAndLoad( )*

***Доступность*** Flash 5 Build 41 и более поздние

***Краткий обзор*** *XMLdoc*.contentType

***Тип доступа*** Чтение/запись

***Описание***

Свойство contentType является типом MIME, который отправляется серверу при вызове *XML.send( )* или *XML. sendAndLoad( )*. По умолчанию имеет значение application/x-www-urlform-encoded. Свойство contentType может быть модифицировано для конкретных объектов *XML*, либо можно модифицировать XML.prototype.contentType, что оказывает воздействие на все объекты *XML*.

Свойство contentType впервые появилось в Build 41 Flash 5 Player – до этого возможности задания типа MIME не было. Проверка действия contentType производится путем сравнения его с undefined или с помощью функции *getVersion( )*, определяющей номер версии и сборки проигрывателя.

***Особенности применения***

Важные замечания об установке типа MIME см. в описании *XML.send( )*.

***См. также***

*XML.send( )*, *XML.sendAndLoad( )*.

---

**XML.createElement( ) метод** создать новый узел элемента

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.createElement(*tagName*)

***Аргументы***

*tagName* Строка с учетом регистра, задающая имя создаваемого элемента. Например, в теге <P ALIGN="RIGHT"> именем тега является P.

***Возвращаемое значение***

Новый объект узла элемента, не имеющий родителя и потомков.

***Описание***

Метод *createElement( )* является главным способом создания новых узлов элементов для включения в иерархию объектов документа *XML*. Обратите внимание, что *createElement( )* не вставляет возвращаемый им элемент в *XMLdoc* – это нужно сделать самостоятельно с помощью *appendChild( )* или *insertBefore( )*. Например, вот как можно создать новый элемент P и вставить его в документ:

```
myDoc = new XML();
newP = myDoc.createElement("P");
myDoc.appendChild(newP);
```

Эти шаги можно объединить следующим образом:

```
myDoc.appendChild(myDoc.createElement("P"));
```

*XMLdoc* должен быть экземпляром класса *XML*, а не класса *XMLnode*.

Метод *createElement( )* нельзя использовать для создания текстовых узлов – для этого нужно пользоваться *createTextNode( )*.

***См. также***

*XML.appendChild( )*, *XML.cloneNode( )*, *XML.createTextNode( )*, *XML.insertBefore( )*.

---

**XML.createTextNode( ) метод** создать новый текстовый узел

***Доступность*** Flash 5

***Краткий обзор*** *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( )* служит основным средством создания новых текстовых узлов для включения в иерархию объектов документа XML. Обратите внимание на то, что *createTextNode( )* не включает в *XMLdoc* возвращаемый им элемент – это нужно сделать самостоятельно с помощью *appendChild( )* или *insertBefore( )*. Вот, например, как 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>"
```

*XMLdoc* должен быть экземпляром класса *XML*, а не класса *XMLnode*.

Текстовые узлы обычно хранятся как дочерние для узлов элементов, создаваемых с помощью *createElement( )*.

***См. также***

*XML.appendChild( )*, *XML.cloneNode( )*, *XML.createElement( )*, *XML.insertBefore( )*.

---

**XML.docTypeDecl свойство** тег DOCTYPE документа

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.docTypeDecl

***Тип доступа*** Чтение/запись

***Описание***

Строка свойства docTypeDecl задает тег DOCTYPE узла *XMLdoc*, если таковой существует. В противном случае, docTypeDecl имеет значение undefined. *XMLdoc* должен быть при этом узлом верхнего уровня в иерархии объектов *XML* (т. е. экземпляром класса *XML*, а не класса *XMLnode*).

Свойство DOCTYPE документа XML задает имя и местонахождение DTD, используемого для проверки действительности документа. ActionScript не осуществляет проверки действительности документов XML, а только анализирует их. Тег DOCTYPE используется для создания документов XML, действительность которых можно проверить внешним образом, 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** ссылка на первого потомка узла

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.firstChild

***Тип доступа*** Только для чтения

***Описание***

Свойство firstChild служит синонимом для childNodes[0]. Оно возвращает ссылку на первый объект узла, являющийся потомком *theNode*. Если у *theNode* нет дочерних узлов, firstChild возвращает null.

В следующем фрагменте исходного кода XML свойство firstChild узла MESSAGE является текстовым узлом с nodeValue «hey»:

```
<!-- Fragment 1 -->
<MESSAGE>hey</MESSAGE>
```

Ниже свойством firstChild узла HOTEL является узел ROOM:

```
<!-- Fragment 2 -->
<HOTEL><ROOM><SIZE>Double</SIZE></ROOM></HOTEL>
```

Когда *theNode* является вершиной иерархии узлов (т. е. ссылается на объект документа *XML*), 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 символы.

Однако если в фрагменте XML не содержится объявления XML и DOCTYPE, 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;
```

Использование nodeValue для доступа к тексту, содержащемуся в теге USER, является хорошим стилем, хотя фактически не обязательно. При использовании объекта текстового узла в строковом контексте для этого узла автоматически вызывается метод *toString( )* и возвращается текст, содержащийся в узле.

***См. также***

*XML*.childNodes, *XML*.lastChild, *XML*.nextSibling, *XML*.previousSibling.

---

**XML.hasChildNodes( ) метод** проверить, есть ли у узла потомки

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.hasChildNodes()

***Возвращаемое значение***

Булева величина: true, если у *theNode* есть дочерние элементы; false, если их нет.

***Описание***

Метод *hasChildNodes( )* показывает, исходит ли из данного узла какая-либо иерархия узлов. Это синонимично выражению сравнения:

```
theNode.childNodes.length > 0
```

Если *theNode* не содержит подузлов, *hasChildNodes( )* возвращает false.

***Пример***

Можно использовать *hasChildNodes( )* для того, чтобы определить, следует ли действовать над данным узлом при обработке узлов. Например, вот как удалять узлы под первым дочерним узлом документа, пока у первого дочернего узла не останется больше дочерних:

```
while (myDoc.firstChild.hasChildNodes()) {
  myDoc.firstChild.firstChild.removeNode();
}
```

***См. также***

*XML*.childNodes.

**XML.ignoreWhite свойство** определяет, нужно ли игнорировать пробельные узлы во время синтаксического анализа XML

***Доступность*** Flash 5 Build 41 и более поздние

***Краткий обзор*** *XMLdoc*.ignoreWhite

***Тип доступа*** Чтение/запись

***Описание***

Свойство ignoreWhite хранит булево значение, которое определяет, нужно ли в процессе синтаксического анализа отбрасывать текстовые узлы, содержащие только пробельные символы. Значением по умолчанию является false (не отбрасывать пробельные узлы). Это глобальная установка, которая применяется к документу XML в целом, а не к определенному узлу. Это означает, что экземпляры класса *XMLnode* не поддерживают *ignoreWhite*.

***Пример***

Чтобы отбрасывать при синтаксическом анализе пробельные символы в отдельном документе XML, используется код:

```
myXML.ignoreWhite = true;
```

Чтобы отбрасывать при синтаксическом анализе пробельные символы во всех документах XML, используется код:

```
XML.prototype.ignoreWhite = true;
```

Свойство ignoreWhite должно быть задано до первой попытки синтаксического разбора XML (обычно в операции *load( )* или *sendAndLoad( )*).

***См. также***

примеры кода удаления пробельных символов вручную в описании класса *XML*.

---

**XML.insertBefore( ) метод** задать для узла новый предыдущий узел того же уровня

***Доступность*** Flash 5

***Краткий обзор*** *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*. Метод *insertBefore( )* аналогичен *appendChild( )*, но позволяет точно задать положение нового узла в существующей иерархии объектов XML.

***Пример***

```
// Создать документ
myDoc = new XML('<P>paragraph 2</P>');

// Создать узел P и текстовый узел
newP = myDoc.createElement("P");
newText = myDoc.createTextNode("paragraph 1");

// Добавить текстовый узел в узел P
newP.appendChild(newText);

// Вставить новый узел P (включая 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** — ссылка на последний дочерний узел для данного

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.lastChild

***Тип доступа*** Только для чтения

***Описание***

Свойство lastChild служит синонимом для childNodes[childNodes.length-1]. Оно возвращает ссылку на объект последнего узла, являющегося дочерним для *theNode*. Если у *theNode* нет дочерних узлов, lastChild возвращает null.

В следующем фрагменте исходного кода XML, lastChild узла MESSAGE является узел 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** — импортировать исходный код XML из внешнего документа

***Доступность*** Flash 5

***Краткий обзор*** *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 содержание *XMLdoc* заменяется вновь загруженным содержанием XML.

*XMLdoc* должен быть экземпляром класса *XML*, а не класса *XMLnode*.

***Особенности применения***

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 по завершении загрузки. Подробности см. в описании *XML*.loaded и *XML.onLoad( )*. Чтобы узнать об успешном завершении синтаксического анализа загруженных данных, следует проверить свойство документа status.

На *XML.load( )* распространяются связанные с безопасностью ограничения по домену, описанные для функции *loadVariables( )* в табл. 8.

***Пример***

```
myDoc = new XML();
myDoc.load("myData.xml");
```

***См. также***

*XML*.loaded, *XML.onLoad( )*, *XML.sendAndLoad( )*, *XML*.status.

---

**XML.loaded свойство** статус операций *load( )* или *sendAndLoad( )*

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.loaded

***Тип доступа*** Только для чтения

***Описание***

Свойство loaded возвращает булево значение, указывающее на завершение ранее вызванных операций *load( )* или *sendAndLoad( )* над *XMLdoc*. Ему присваивается false в самом начале операций XML *load( )* или *sendAndLoad( )*. Если загрузка произведена успешно, то loaded присваивается true. Если с *XMLdoc* не производилось ни одной такой операции, loaded имеет значение undefined.

Если loaded имеет значение false, загрузка и разбор данных XML все еще продолжается, и попытки доступа к иерархии объектов в *XMLdoc* окажутся неудачными. Если loaded имеет значениеis true, то данные XML загружены, проанализированы и записаны в иерархии объектов *XMLdoc*. Обратите,

однако, внимание, что синтаксический анализ загруженных данных XML может оказаться неудачным (это можно выяснить из *XMLdoc*.status).

*XMLdoc* должен быть экземпляром класса *XML*, а не класса *XMLnode*.

***Пример***

В следующем примере показан базовый предварительный загрузчик XML, который ждет окончания загрузки данных XML, прежде чем вывести их (предварительные загрузчики XML можно также создавать на основе обработчика события *XML.onLoad( )*):

```
// 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 окнчания 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.nextSibling свойство** ссылка на узел, следующий за данным

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.nextSibling

***Тип доступа*** Только для чтения

***Описание***

Свойство nextSibling возвращает объект узла, следующего за *theNode* на текущем уровне иерархии объектов XML. 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 используется для обхода иерархии объектов XML. Например, чтобы просмотреть все дочерние элементы *theNode* в порядке их появления, можно воспользоваться кодом:

```
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 — имя текущего узла

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.nodeName

***Тип доступа*** Чтение/запись

### *Описание*

Строковое свойство nodeName отражает имя *theNode*. Поскольку ActionScript поддерживает только два типа узлов (узлы элементов и текстовые узлы), nodeName может иметь только два возможных значения:

- Если *theNode* является узлом элемента, nodeName является строкой, совпадающей 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, которая определяет nodeName для текстового узла как строку "#text". При желании можно пользоваться совместимым с DOM свойством nodeType.

### *Пример*

С помощью nodeName можно проверять, является ли текущий узел тем типом элемента, который разыскивается. Например, в следующем коде извлекается все содержимое тегов H1 на первом уровне документа XML (в этом примере ищутся только теги с именем *H1*, кроме тегов с именем *h1* на нижнем регистре):

```
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 == "H1") {
    trace(myDoc.childNodes[i].firstChild.nodeValue);
  }
}
```

***См. также***

*XML*.nodeType, *XML*.nodeValue.

---

**XML.nodeType свойство** тип текущего узла

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.nodeType

***Тип доступа*** Только для чтения

***Описание***

Свойство nodeType является целым числом, представляющим тип *theNode*. Поскольку в ActionScript поддерживаются только два типа узлов – узлы элементов и текстовые узлы, то nodeName может принимать только два возможных значения: 1, если узел является узлом элемента, и 3, если узел является текстовым. Эти значения не являются произвольными – они соответствуют спецификации DOM. Для справки, прочие типы узлов в DOM перечислены в табл. 13.

*Таблица 13. 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 помимо узлов *element* и *text* так называемые узлы *attribute*, *document* и *document_ type*, 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 непосредственного доступа к самому *document_type* нет.

Узлы элементов соответствуют тегам XML или HTML. Например, во фрагменте XML <P>what is your favorite color?</P> тег P будет представлен в иерархии объектов *XML* как узел элемента (nodeType 1). Тег, содержащийся в теге в исходном коде XML – например, текст «what is your favorite color?» – будет представлен как текстовый узел (nodeType 3).

### *Пример*

В зависимости от типа узла (свойства nodeType) можно осуществлять над ним условные операции. Вот, например, код, удаляющий 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 является текстовым...
  if (theNode.childNodes[i].nodeType == 3) {
    // Проверить, есть ли в этом узле значимые символы
    var j = 0;
    var emptyNode = true;
    for (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;
      }
    }
    // Значимые символы не найдены, поэтому удалить узел
    theNode.childNodes[i].removeNode();
  }
}
```

### *См. также*

XML класс, *XML*.nodeName, *XML*.nodeValue.

---

**XML.nodeValue свойство** значение текущего узла

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.nodeValue

***Тип доступа*** Чтение/запись

### *Описание*

Свойство nodeValue отражает строковое значение *theNode*. Поскольку в ActionScript поддерживается только два типа узлов (узлы элементов и текстовые узлы), у nodeValue есть только два возможных значения:

- Если *theNode* является узлом элемента, nodeValue имеет значение null.
- Если *theNode* является текстовым узлом, значение nodeValue – текст, содержащийся в узле.

На практике nodeValue обычно используется только с текстовыми узлами. Чтобы присвоить новый текст существующему текстовому узлу, можно так использовать nodeValue:

```
// 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";
```

Хотя значение текстового узла можно получить явно с помощью nodeValue, метод *toString( )* неявно возвращает значение узла, когда тот используется в строковом контексте. Поэтому следующий код выводит текст текстового узла в окне Output:

```
trace(myDoc.firstChild.firstChild);
```

***См. также***

*XML*.nodeName, *XML*.nodeType.

---

**XML.onData( ) обработчик события** выполняется, когда загрузка внешнего исходного кода XML завершена, но синтаксический анализ еще не произведен

***Доступность*** Flash 5 (не документирован)

***Краткий обзор*** *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*, вызванная *load( )* или *sendAndLoad( )*. По умолчанию *onData( )* работает в следующем режиме:

- Если загруженный исходный код имеет значение undefined, вызывается *XMLdoc.onLoad( )* с параметром success, которому присвоено значение false.
- В противном случае, исходный код разбирается в *XMLdoc*, свойству XMLdoc.loaded присваивается значение true, и вызывается *XMLdoc.onLoad( )* с параметром success, которому присвоено значение 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 применения встроенного анализатора 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");
```

***См. также***

*XML.onLoad( ).*

---

**XML.onLoad( ) обработчик события** выполняется, когда внешние данные XML загружены и обработаны

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.onLoad(*success*)

***Аргументы***

*success* Булево значение, указывающее, была ли загрузка успешной (`true`) или нудачной (`false`).

***Описание***

Обработчик *onLoad( )* объекта *XMLdoc* автоматически выполняется, когда внешний файл XML загружен в *XMLdoc* с помощью методов *load( )* или *sendAndLoad( )*. По умолчанию обработчик *onLoad( )* объекта документа XML является пустой функцией. Для использования *onLoad( )* ему присваивается *обработчик обратного вызова* (*callback handler*), т. е. пользовательская функция. Например:

```
myDoc = new XML();
myDoc.onLoad = handleLoad;
function handleLoad (success) {
  // Здесь XML обрабатывается, как нужно...
}
```

События *onLoad( )* сообщают о том, что можно безбоязненно обрабатывать *XMLdoc*. Если сгенерировано *onLoad( )*, то известно, что загрузка и синтаксический анализ внешних данных XML завершены, и обращаться к этому загруженному содержимому безопасно. Поэтому обработчик *onLoad( )* устраняет необходимость написания кода предварительной загрузки, который должен ждать поступления данных после вызова функции XML *load( )*. Например, 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 *display( )*. В противном случае, с помощью функции *display( )* выводится сообщение об ошибке. (Функции *displayProduct( )* и *displayError( )* являются пользовательскими функциями, которые предположительно написаны для вывода информации пользователю, но 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
    }
  }
```

Обратите внимание, что значение аргумента `success` функции *handleLoad( )* автоматически задается интерпретатором как `true` или `false`, указывая на успех или неудачу загрузки. Однако аргумент `success` может оказаться более полезным в теории, чем на практике. Большинство сообщений веб-серверов об ошибках (например, «404 File Not Found») приходит в формате документов HTML. Поскольку HTML может быть успешно проанализирован как данные XML, получение с сервера страницы ошибки заканчивается разбором этой страницы в целевой объект документа XML. Так как страница анализируется успешно, попытка загрузки оказывается «успешной», и `success` присваивается `true`, хотя сам файл XML может оказаться ненайденным, или возникает другая ошибка сервера. Чтобы иметь уверенность в получении действительно запрошенных данных, нужно явно проверить наличие в их структуре или содержимом какой-либо идентифицирующей характеристики, например, `nodeName` для определенного дочернего узла. См. также описание обработчика события *XML.onData( )*, с помощью которого можно производить специализированный синтаксический анализ.

***См. также***

*XML.load( ), XML.onData( ), XML.sendAndLoad( ).*

---

**XML.parentNode свойство** ссылка на ближайшего предка узла

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.parentNode

***Тип доступа*** Только для чтения

***Описание***

Свойство `parentNode` возвращает ссылку на объект узла, от которого происходит *theNode* в иерархии объектов XML. Если *theNode* является верхушкой в текущей иерархии, `parentNode` возвращает `null`.

В следующем фрагменте исходного кода XML узел `MESSAGE` является `parentNode` для текстового узла `"hey"`:

```
<MESSAGE>hey</MESSAGE>
```

А здесь `parentNode` для узла `ROOM` является узел `HOTEL`:

```
<HOTEL><ROOM><SIZE>Double</SIZE></ROOM></HOTEL>
```

***См. также***

*XML*.childNodes, *XML*.firstChild, *XML*.lastChild, *XML*.previousSibling.

**XML.parseXML( ) метод** — провести синтаксический анализ строки с исходным кодом XML

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.parseXML(*string*)

***Аргументы***

*string* — Строка с исходным кодом XML.

***Описание***

Метод *parseXML( )* похож на внутреннюю функцию *load( )* – он читает и анализирует исходный код XML, содержащийся в *string*, преобразует его в иерархию объектов и помещает полученную иерархию в *XMLdoc*. Любое предшествующее содержимое *XMLdoc* заменяется новой иерархией. *XMLdoc* должен быть экземпляром класса *XML*, а не класса *XMLnode*.

Чтобы включить необработанный исходный код HTML или XML в текстовый узел, не производя его анализа, используйте раздел CDATA:

```
<![CDATA[ source ]]>
```

Например, 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>");
trace(myDoc);  // Displays: "<MESSAGE><B>Welcome</B> to my site</MESSAGE>"
```

***Пример***

Метод *parseXML*() можно использовать для замены текущей иерархии объектов в объекте *XML* новой иерархией, основанной на внутренне образованном исходном коде XML (например, некоторых данных, введенных пользователем). В следующем примере создается простое сообщение XML путем объединения разметки с данными ввода из текстовых полей username и content:

```
myDoc = new XML();
myXMLsource = "<MESSAGE><USER>" + username + "</USER><CONTENT>"
    + content + "</CONTENT></MESSAGE>";
myDoc.parseXML(myXMLsource);
```

***См. также***

*XML.load( )*, *XML*.status.

---

**XML.previousSibling свойство** — ссылка на узел, предшествующий данному

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.previousSibling

***Тип доступа*** Только для чтения

***Описание***

Свойство previousSibling возвращает объект узла, предшествующего *theNode* на текущем уровне иерархии объектов XML. Если на текущем уровне иерархии перед *theNode* нет узлов, previousSibling возвращает null.

В следующем фрагменте исходного кода XML узел CONTENT является previousSibling для узла USER:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

***Пример***

Свойство previousSibling можно использовать для обхода иерархии объектов XML, хотя чаще для этого используется nextSibling. Например, чтобы просмотреть все дочерние элементы *theNode* в обратном порядке, можно воспользоваться следующим кодом:

```
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** удалить узел из иерархии объектов XML

***Доступность*** Flash 5

***Краткий обзор*** *theNode*.removeNode()

***Описание***

Метод *removeNode( )* удаляет *theNode* из документа XML. Все потомки (дети, внуки и так далее) *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, и его место 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.send( ) метод** передать исходный код XML в сценарий

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.send(*URL*, *window*)

***Аргументы***

*URL* Строка, указывающая местоположение 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 пользовательским именем или одним из четырех предустановленных: "_blank", "_parent", "_self" или "_top". Подробности см. 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, находящимся по *URL*. 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; чтобы ответ перехватывался Flash, используйте *sendAndLoad( )*.

Когда *XML.send( )* вызывается из Flash Player, выполняющегося в броузере, *XMLdoc* посылается с помощью метода POST. Когда *XML.send( )* вызывается из Flash Player, выполняемого как самостоятельное приложение, *XMLdoc* посылается с помощью метода GET. Приложение сервера, получающее переданную строку XML, должно иметь возможность непосредственного 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 на STDIN и могут быть извлечены и сохранены , скажем, в $buffer следующим образом:

```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
```

В ASP доступ к необработанным данным POST может осуществляться с помощью метода *Request.BinaryRead*. В некоторых приложениях (например, в Cold Fusion) отсутствует 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-urlform-encoded. Однако этот тип является чисто внешним: сам текст не имеет кодировки URL. В Build 41 и выше Flash 5 Player тип содержимого MIME может быть модифицирован с помощью свойства XML.contentType. Например, типу MIME присваивается значение application/xml с помощью кода:

```
myXML = new XML();
myXML.contentType = "application/xml";
```

Тем не менее задание свойства contentType явным образом в application/x-www-urlform-encoded не влечет URL-кодирования отправляемого текста.

Обратите внимание, что в Build 41 Flash 5 Player при анализе исходного кода XML и появлении в текстовом узле символов &, ', ", < и >, эти символы преобразуются в сущности &, ', ", >, <. Такое преобразование осуществляется во Flash прозрачным образом, поскольку сущности преобразуются обратно в исходные символы, когда объект XML преобразуется в строку; однако сущности появятся в исходном коде XML, который отправляется серверу.

***Пример***

```
myDoc = new XML("<SEARCH_TERM>tutorials</SEARCH_TERM>");
myDoc.send("http://www.domain.com/cgi-bin/lookup.cgi", "remoteWin");
```

***См. также***

*XML.sendAndLoad( ), XML.load( ),* `XML.loaded`, *XML.onLoad( ),* `XML.status`.

---

**XML.sendAndLoad( ) метод** — отправить сценарию исходный код XML и получить обратно исходный код XML

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.sendAndLoad(*URL*, *resultXML*)

***Аргументы***

*URL* — Строка, указывающая местоположение сценария или приложения, которому должен быть передан *XMLdoc*.

*resultXML* — Ссылка на объект документа *XML*, который получит возвращенный исходный код XML.

***Описание***

Метод *sendAndLoad( )* преобразует *XMLdoc* последовательно в строку исходного кода XML и отправляет этот код сценарию или приложению, находящимся по указанному *URL*. 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( )*.

***Особенности применения***

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* устанавливается обработчик события *onLoad( )*, который должен реагировать на завершение загрузки. Подробности см. в описании `XML.loaded` и *XML.onLoad( )*. Чтобы узнать результат синтаксического анализа загруженных данных, проверяется свойство документа `status`.

На *XML.sendAndLoad( )* распространяются связанные с безопасностью ограничения по домену, описанные для глобальной функции *loadVariables( )* в табл. 8.

***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();
}
```

Хорошее руководство для начинающих об обмене XML с сервером есть в статье Macromedia «Integrating XML and Flash in a Web Application» (интеграция XML и Flash в приложении сети) по адресу:

*http://www.macromedia.com/support/flash/interactivity/xml*

***См. также***

*XML.load( )*, *XML*.loaded, *XML.onLoad( )*, *XML.send( )*, *XML*.status.

---

**XML.status свойство** — указывает, был ли успешным разбор XML в иерархию объектов

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.status

***Тип доступа*** Только для чтения

***Описание***

Свойство status возвращает числовой код состояния, указывающий возможные ошибки при синтаксическом анализе исходного кода XML. Синтаксический анализ происходит тогда, когда исходный код XML:

- Передается в качестве аргумента конструктору *XML( )*
- Явно анализируется в методе *parseXML( )*
- Загружается в новый объект *XML* через методы *load( )* или *sendAndLoad( )*

Коды status приведены в табл. 14. Если при синтаксическом анализе не возникло ошибок, status равен 0. Ошибки обозначаются отрицательными числами. Синтаксический анализ завершается при обнаружении первой ошибки, поэтому даже после исправления ошибок, о которых было сообщено, могут обнаружиться новые.

Обычно status используется для определения того, безопасно 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 определяет действительность документов по DTD, а только проверяет их корректность.

*Таблица 14. Коды статуса синтаксического анализа XML*

| Статус | Описание |
|---|---|
| 0 | Анализ документа не выявил ошибок (т. е. успешен). |
| −2 | Раздел CDATA завершен 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 . |

***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** исходный код узла XML в виде строки

***Доступность*** Flash 5

***Краткий обзор*** *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. Если *theNode* является объектом XML верхнего уровня, в строку включаются имеющиеся теги DOCTYPE и объявления XML. Если свойство документа ignoreWhite имеет значение false, сохраняются пробельные символы, и исходный код документа выглядит таким, каким он был при анализе.

Обычно нет необходимости явно вызывать *toString( )*: *toString( )* автоматически вызывается при любом использовании *theNode* в строковом контексте.

***Пример***

```
var myDoc = new XML('<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd"><BOOK>
```

```
<TITLE>ActionScript: 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:
 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** тег объявления документа XML

***Доступность*** Flash 5

***Краткий обзор*** *XMLdoc*.xmlDecl

***Тип доступа*** Чтение/запись

***Описание***

Строковое свойство xmlDecl представляет тег объявления XML в *XMLdoc*, если таковое существует. В противном случае, xmlDecl имеет значение undefined. *XMLdoc* должен быть узлом верхнего уровня в иерархии объектов *XML* (т. е. экземпляром класса *XML*, а не класса *XMLnode*.)

Тег объявления XML в документе XML используется для указания версии XML, используемой в документе. Тег объявления XML используется для построения корректных документов XML, действительность которых может быть проверена внешним образом.

***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);  // Выводит: "<?xml version="1.0"?>"
// Установить новое объявление XML
myXML.xmlDecl = '<?xml version="1.0" standalone="no"?>';
```

***См. также***

*XML*.docTypeDecl.

---

**XMLnode класс** внутренний надкласс класса XML

***Доступность*** Flash 5

***Описание***

Класс *XMLnode* определяет базовые свойства и методы узлов в иерархии объектов XML. Хотя *XMLnode* является внутренним инструментом, он может использоваться 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 контейнера

Узел главного контейнера является экземпляром класса *XML*. Например, если создать `myDoc` так:

```
myDoc = new XML();
```

то `myDoc` будет экземпляром класса *XML*. Класс *XML* является наследником *XMLnode*, поэтому узлы главного контейнера имеют все свойства и методы, определенные *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 использование объектов XML. Однако, если требуется добавить наследуемое свойство или метод во все объекты XML, то нужно использовать `prototype` класса *XMLnode*, а не класса *XML* (см. последующий пример). Методы и свойства, прикрепленные к `XMLnode.prototype`, наследуются всеми узлами XML фильма.

Для справки в табл. 15 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*, недоступны для экземпляров *XMLnode*. Например, метод *load( )* может быть вызван для экземпляра класса *XML*, но не для экземпляра класса *XMLnode*. Подробное описание каждого элемента см. в соответствующем месте описания класса *XML*.

*Таблица 15. Свойства, методы и обработчики событий для XMLnode и XML*

| XMLnode и XML | только XML |
|---|---|
| *appendChild( )* | `contentType` |
| `attributes` | *createElement( )* |
| `childNodes` | *createTextNode( )* |
| *cloneNode( )* | `docTypeDecl` |
| `firstChild` | `ignoreWhite` |
| *hasChildNodes( )* | *load( )* |
| *insertBefore( )* | `loaded` |
| `lastChild` | *onData( )* |
| `nextSibling` | *onLoad( )* |

*Таблица 15 (продолжение)*

| XMLnode и XML | только XML |
|---|---|
| nodeName | *parseXML( )* |
| nodeType | *send( )* |
| nodeValue | *sendAndLoad( )* |
| parentNode | status |
| previousSibling | xmlDecl |
| *removeNode( )* | |
| *toString( )* | |

***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());  // Выводит: "<PRODUCT>Game Console</PRODUCT>"
```

Совершенно допустимо расширять класс *XML* через XML.prototype, но такое расширение будет применяться только к узлам главных контейнеров (непосредственным экземплярам класса *XML*).

***См. также***

*XML* класс; раздел «Надклассы и подклассы» в главе 12 «Объекты и классы».

---

**XMLSocket класс** поддержка непрерывного соединения TCP/IP между сервером и клиентом

***Доступность*** Flash 5

***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 выполнена. |

| | |
|---|---|
| *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 имеет очень короткий срок жизни. Когда Flash запрашивает внешние данные через функции *loadMovie( )*, *loadVariables( )* или *XML.load( )*, устанавливается временный канал связи. По этому каналу передаются данные, после чего он закрывается. Такого рода кратковременные соединения могут иметь полезное применение, но они ограничены в двух важных отношениях:

- 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 повторяющихся открытий соединений, не позволяют Flash и близко подойти к операциям с сервером в реальном масштабе времени.

Flash 5 позволяет преодолеть эти ограничения с помощью класса *XMLSocket*, который дает возможность 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.

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 приемник и передатчик. 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, используемое с объектом *XMLSocket*, должно:

- Обслуживать соединения сокетов TCP/IP на заданном порту с номером, большим или равным 1024
- Передавать данные в формате XML сегментами, разделяемыми *нулевым байтом* (т. е. символом ASCII null)

Приложения сервера обычно создаются не программистами Flash, а программистами сервера. Примером простого серверного приложения, передающего все полученные сообщения всем подключенным клиентам, является сервер Java XMLSocket, который можно взять в сетевом хранилище кода.

Соединение *XMLSocket* остается открытым, пока не произойдет одно из следующих событий:

- Вызван метод *close( )* объекта *XMLSocket*.
- Ссылки на объект *XMLSocket* перестали существовать.
- Сервер завершает соединение (при этом запускается событие *onClose( )*).
- Фильм закрыт, или Flash Player завершил работу.

Удобно, что класс *XMLSocket* предоставляет также возможность следить за состоянием соединения. У него есть три свойства – `onClose`, `onConnect` и `onXML`, позволяющие определять обработчики событий, которые будут запущены при возникновении соответствующих событий. Такие обработчики обычно называют *обработчиками обратного вызова* (*callback handlers*), поскольку они автоматически запускаются Flash в ответ на некоторые события, находящиеся за пределами прямого контроля программиста. (В этом смысле они очень похожи на встроенные обработчики событий клипов и кнопок в ActionScript, но функции обработчиков определяются программистом.) Например, если соединение закрыто сервером, будет запущен обработчик, определенный в свойстве `onClose`.

Если вы не определите обработчики обратного вызова для свойств `onClose` и `onConnect`, то не сможете осуществлять проверку ошибок или обеспечивать какую-либо обратную связь. Если вы не определите обработчик обратного вызова для свойства `onXML`, вы не получите уведомление при получении сокетом данных от приложения на стороне сервера, а также не сможете извлечь эти данные.

### *Пример*

В следующем примере показан костяк кода, необходимого для реализации простого клиента чата. Посмотреть на работу клиента можно по адресу:

*http://www.moock.org/chat*

Как сервер, так и клиент можно получить из сетевого хранилища кода:

```
// Простой клиент чата
// *** Общая инициализация
var incomingUpdated = false;  // Следить, нужна ли прокрутка до конца
                              // incoming (главное текстовое поле чата)
var incoming = "";            // Присвоить начальное значение главному
                              // текстовому полю чата

// Прикрепить фильм администратора прокрутки. Он вынуждает текстовое поле
// чата показывать следующую (самую новую) строку при добавлении нового
// сообщения. Обратите внимание, что администратор прокрутки нужен только
// из-за ошибки прокрутки текстового поля в Build r30 Flash 5 Player.
attachMovie("processScroll", "processScroll", 0);

// Прикрепить звук, проигрываемый при получении сообщения
var click = new Sound();
click.attachSound("click");

// Прикрепить звук, проигрываемый при подключении или уходе
```

```
// пользователя
var welcomeSound = new Sound();
welcomeSound.attachSound("welcome");

// Отключить некрасивую желтую подсветку кнопок
_focusrect = 0;

// *** Создает новый сокет и пытается соединиться с сервером
function connect () {
// Создать новый объект XMLSocket
  mySocket = new XMLSocket();

// Присвоить функции обратного вызова обработчикам mySocket.
  mySocket.onConnect = handleConnect;
  mySocket.onClose = handleClose;
  mySocket.onXML = handleIncoming;

// Попытка соединения и присвоение значения, возвращенного
// mySocket.connect() переменной connectSuccess (connect() возвращает
// true, если начальная стадия соединения успешна)
  var connectSuccess = mySocket.connect("www.myserver.com", 1025);
  if (connectSuccess) {
    trace("initial connection succeeded");
  } else {
    // connectSuccess было false, так что соединение не установлено.
    gotoAndStop("connectionFailed");
    trace("initial connection failed");
  }
}

// *** Обработчик события, реагирующий на завершение попытки соединения
function handleConnect (succeeded) {
  // Если аргумент succeeded функции handleConnect() равен true,
  // соединение установлено, и можно приступить к чату.
  // В противном случае, вывести сообщение о неудаче.
  if (succeeded) {
  // Задать свойство, указывающее на наличие открытого соединения.
    mySocket.connected = true;
    gotoAndStop("chat");
    // Поместить курсор в текстовое поле "send message"
    Selection.setFocus("_level0.outgoing");
  } else {
    // Соединение не удалось, вывести сообщение об ошибке
    gotoAndStop("connectionFailed");
    trace("connection failed");
  }
}

// *** Обработчик события, вызываемый при закрытии сервером соединения
function handleClose () {
  // Сообщить пользователю, что связь потеряна
  incoming += ("The server has terminated the connection.\n");
  // Сообщить администратору прокрутки, что текстовое поле чата обновлено
  incomingUpdated = true;
```

```
  // Установить свойство, указывающее на потерю соединения
  mySocket.connected = false;
  numClients = 0;
}

// *** Обработчик события для получения и отображения входящих сообщений
function handleIncoming (messageObj) {
  // Вывести полученные данные XML в окне Output
  trace("--------new data received-----------");
  trace(">>" + messageObj.toString() + "<<");
  trace("-------- end of new data -----------");

  // Сообщить администратору прокрутки, что текстовое поле чата обновляется
  incomingUpdated = true;
  lastScrollPos = incoming.scroll;

  // Получить время
  var now = new Date();
  var hours = now.getHours();
  var minutes = now.getMinutes();
  var seconds = now.getSeconds();
  // Форматировать время для вывода
  hours = (hours < 10 ? "0" : "") + hours;
  minutes = (minutes < 10 ? "0" : "") + minutes;
  seconds = (seconds < 10 ? "0" : "") + seconds;

  // Сервер посылает NUMCLIENTS при каждом подключении
  // или отсоединении пользователя
  // Если в объекте XML обнаружено NUMCLIENTS...
  if (messageObj.firstChild.nodeName == "NUMCLIENTS") {
  // ...проверить, пусто ли окно входящих сообщений. Если да...
    if (incoming == "") {
    // ...то присоединился новый пользователь, добавить в чат приветствие.
      incoming += ("welcome to moock comm 1.0.0, "
          + userID + "\n"
         + "  connection time: " + hours + ":" + minutes + ":" + seconds + "\n"
         + "  server: clayton\'s javaComm generic flash xmlsocket server\n\n");
    } else {
     // В противном случае кто-то вошел или вышел, сообщить об этом пользователю
      if (parseInt(messageObj.firstChild.firstChild.nodeValue) > numClients) {
        // Сообщить о приходе клиента в окне чата
        incoming += (hours + ":" + minutes + ":"
                 + seconds + " a new user has connected.\n");
      } else {
        // Сообщить об уходе клиента в окне чата
        incoming += (hours + ":" + minutes + ":"
                 + seconds + " a user disconnected.\n");
      }
    }
    // Наконец, запомнить новое количество клиентов и
    // воспроизвести звук приветствия/прощания
    numClients = parseInt(messageObj.firstChild.firstChild.nodeValue;)
    welcomeSound.setVolume(100);
```

```
    welcomeSound.start();
  } else {
    // Узел NUMCLIENTS не обнаружен, значит, это обычное сообщение.
    // Выбрать из объекта XML имя пользователя и сообщение.
    var user = messageObj.firstChild.firstChild.nodeValue;
    var message = messageObj.childNodes[1].firstChild.nodeValue;

    // Добавить сообщение в окно чата вместе с временной меткой
    incoming += (hours + ":" + minutes
             + ":" + seconds + user + ">> " + message + "\n");

    // Теперь воспроизвести щелчок нового сообщения. Если с получения
    // последнего сообщения прошло более 30 секунд, щелчок должен
    // быть громким, иначе - тихим.
    trace("time since last message: " + (now - lastIncomingMessageTime));
    if (lastIncomingMessageTime && (now - lastIncomingMessageTime) > 30000) {
      click.setVolume(200);
    } else {
      click.setVolume(30);
    }
    click.start();
  }

  // Обрезать содержимое главного текстового поля чата,
  // если оно длиннее 5000 символов
  if (incoming.length > 5000) {
    var nearestNewline = incoming.indexOf("\n", incoming.length - 5000);
    incoming = incoming.substring(nearestNewline, incoming.length);
  }

  // Запомнить время поступления этого сообщения для следующего раза
  lastIncomingMessageTime = now;
}

// *** Пересылает новый объект XML серверу
function sendMessage() {
  // Создать сообщение, посылаемое в виде фрагмента кода XML. Обратите
  // внимание на обязательность пробелов перед тегами <USER>
  // и </MESSSAGE>, чтобы у MESSAGE и USER всегда был текстовый дочерний узел.
  var message = '<USER> ' + userID + '</USER><MESSAGE>'
              + outgoing + ' </MESSAGE>';

  // Преобразовать сообщение в иерархию объектов XML
  messageObj = new XML();
  messageObj.parseXML(message);

  // Проверить, что отправляется
  trace("Sending: " + messageObj);

  // Если объект сокета существует и находится на связи, отправить сообщение XML.
  // Иначе предупредить пользователя, что сначала нужно соединиться.
  if (mySocket && mySocket.connected) {
    mySocket.send(messageObj);
    // Очистить текстовое поле "send message"
    outgoing = "";
```

```
    } else {
      // Должно быть, сервер нас выкинул...
      incoming += "You are no longer connected. Please reconnect.\n"
      incomingUpdated = true;
    }
  }

  // *** Закрывает соединение с сервером
  function quit() {
    if (mySocket.connected) {
      mySocket.close();
      mySocket.connected = false;
      numClients = 0;
      incoming = "";
      gotoAndStop("login");
    }
  }
```

***См. также***

*oadVariables( )*, XML класс

---

**XMLSocket.close( ) метод** — завершить открытое соединение с приложением сервера

***Доступность*** Flash 5

***Краткий обзор*** `socket.close()`

***Описание***

Метод *close( )* разрывает связь между *socket* и приложением сервера. После выполнения *close( )* над *socket* попытки вызова *send( )* с *socket* оказываются безуспешными. Приложение сервера тоже больше не сможет отправлять данные Flash через *socket*.

Обратите внимание, что *close( )* не оказывает никакого действия, если *socket* уже закрыт или ранее не был соединен. Кроме того, *close( )* не запускает обработчик *onClose( )* объекта сокета: *onClose( )* генерируется только при закрытии соединения со стороны сервера.

***См. также***

*XMLSocket.connect( )*, *XMLSocket.onClose( )*.

---

**XMLSocket.connect( ) метод** — открыть соединение с приложением сервера

***Доступность*** Flash 5

***Краткий обзор*** `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, значением по умолчанию считается адрес сервера, с которого был передан фильм.

*port* Целое число, задающее номер порта TCP, больший или равный 1024.

***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 *host* по порту *port*.

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, например, когда оказываются неудачными. При вызове *connect( )* следует показывать пользователю, что идет процесс установления соединения.

Если *connect( )* возвращает `false`, то начальная фаза установки соединения не была успешно завершена. В этом случае обработчик *onConnect( )* для *socket* не будет вызван.

Важно проверять как значение, возвращаемое методом *connect( )*, так и, если *connect( )* возвращает `true`, значение параметра *success* обработчика *onConnect( )*.

***Особенности применения***

По соображениям безопасности не разрешается подключение с помощью *connect( )* к произвольным хостам в Интернет. Подключение возможно только к хостам в домене, из которого был загружен фильм. Для *connect( )* действуют такие же правила, что и для функции *loadVariables( )*. См. в описании глобальной функции *loadVariables( )* список требований к соответствию доменов, налагаемых методом *connect( )*. Метод *connect( )* возвращает `false` для попыток 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
  gotoAndStop("failureDialog");
} else {
  // Переход в кадр, где будет ожидаться генерирование onConnect
```

```
    gotoAndPlay("connecting");
  }
```

***См. также***

*XMLSocket.close( ), XMLSocket.onConnect( ).*

---

**XMLSocket.onClose( ) обработчик события** — указывает обработчик обратного вызова, вызываемый при закрытии соединения сервером

***Доступность*** Flash 5

***Краткий обзор***

```
socket.onClose = closeHandler
socket.closeHandler()
```

***Описание***

Свойство *onClose* позволяет задать обработчик обратного вызова, автоматически выполняемый при закрытии сервером соединения с *socket*. Инициируемое сервером закрытие обычно происходит при завершении работы приложении сервера или преднамеренном «выкидывании» клиента.

***Пример***

Для реакции на событие *onClose* нужно присвоить свою функцию (т. е. собственный обработчик обратного вызова) свойству *onClose* объекта *XMLSocket*. На практике этот обработчик обратного вызова используется для обнаружения потери соединения с внешним 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`:

```
mySocket = new XMLSocket();
mySocket.onClose = handleClose;

function handleClose () {
  status += ("\nThe server has terminated the connection.\n");
}
```

***См. также***

*XMLSocket.close( )*; раздел «Прикрепление обработчиков событий к другим объектам» главы 10.

---

**XMLSocket.onConnect( ) обработчик события** — определяет обработчик события, вызываемый по завершении попытки соединения, успешной или нет

***Доступность*** Flash 5

***Краткий обзор***

```
socket.onConnect = connectHandler
socket.connectHandler(success)
```

### *Аргументы*

*success* Булево значение, указывающее на то, что попытка соединения была успешной (`true`) или неудачной (`false`).

### *Описание*

Свойство *onConnect* позволяет задать обработчик обратного вызова, автоматически выполняемый при завершении ранее вызванной операции *connect( )*. Выполнение обработчика обратного вызова, заданного *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 (т. е. на то, что соединение установлено). Если это так, *success* присваивается `true`. Если попытка была неудачной (т. е. произошли перерасход времени соединения, отказ в соединении или невозможность соединения по другим причинам), аргумент *success* получает значение `false`. Обратите внимание, что ActionScript не делает различия между перерасходом времени, неизвестным хостом, отказом в доступе и другими обычными ошибками соединения. Поэтому обработчик обратного вызова может не выполняться вплоть до минуты после подачи команды *connect( )* в зависимости от настроек сервера, попытка соединения с которым осуществляется, скорости соединения, сетевого трафика и т. д.

### *Пример*

Обработчик обратного вызова, заданный *onConnect*, используется для обнаружения успеха или неудачи попытки соединения. На практике обработчик обратного вызова может использоваться для установки флага, указывающего на необходимость начала передачи, если соединение оказалось успешным. С помощью обработчика обратного вызова можно также выполнять код, нейтрализующий ошибку неудачного соединения, например, сообщающий пользователю о сущности проблемы.

Для реагирования на событие *onConnect* нужно присвоить свою функцию (т. е. обработчик обратного вызова) свойству *onConnect* объекта *XMLSocket*. В следующем коде функция *handleConnect( )* присваивается свойству *onConnect* объекта *mySocket*. Путем обновления текстового поля `status` функция *handleConnect( )* извещает пользователя об успехе или неудаче соединения:

```
mySocket = new XMLSocket();
mySocket.onConnect = handleConnect;

function handleConnect (succeeded) {
  if (succeeded) {
    status += ("Successfully connected.\n");
  } else {
    status += ("Connection attempt failed.\n");
  }
}
```

Код, показывающий использование обработчика *onConnect( )* в более полной системе, можно найти в примере для класса *XMLSocket*.

***См. также***

*XMLSocket.connect( )*; раздел «Прикрепление обработчиков событий к другим объектам» в главе 10.

---

**XMLSocket.onData( ) обработчик события** выполняется, когда внешние данные получены, но еще не разобраны как XML

***Доступность*** Flash 5 (не документирован)

***Краткий обзор*** `socket.onData(src)`

***Аргументы***

*src* Строка, содержащая загруженные данные, обычно являющиеся исходным кодом XML.

***Описание***

Обработчик *onData( )* выполняется автоматически, 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 быть присвоена пользовательская функция обратного вызова, перехватывающая `src` до того, как ActionScript может проанализировать его как XML. В некоторых случаях (например, в видеоиграх реального времени) обработка вручную данных в `src` может быть быстрее по сравнению с встроенным в ActionScript синтаксическим анализом.

***Пример***

Следующий код показывает, как присвоить *onData( )* пользовательскую функцию обратного вызова. Эта функция обратного вызова просто выводит 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** определяет обработчик обратного вызова, вызываемый, когда данные получены объектом XMLSocket и проанализированы как XML

***Доступность*** Flash 5

***Краткий обзор*** *socket*.onXML = *xmlHandler*
*socket.xmlHandler(XMLobject)*

***Аргументы***

*XMLobject* Объект XML, который будет хранить поступившие а формате XML данные.

***Описание***

Свойство onXML позволяет задать обработчик обратного вызова, который выполняется при получении Flash входящей передачи. Когда *socket* получает от сервера законченный блок данных (т. е. строку, за которой следует символ ASCII null), автоматически вызывается обработчик обратного вызова, заданный в *socket*.onXML. Сервер может посылать данные с удобной для него частотой, но обработчик обратного вызова выполняется только тогда, когда *socket* в конце получает символ null (т. е. 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 обработчику обратного вызова как аргумент *XMLobject*.

Если вы являетесь Flash-программистом, ответственным 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, переназначьте режим работы по умолчанию обработчика события сокета *onData( )*. См. *XMLSocket.onData( )*.

***Пример***

Чтобы реагировать на событие onXML, нужно присвоить свою функцию (т. е. обработчик 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:

```
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 += (message.nodeValue + "\n");
}
```

Код, показывающий использование обработчика *onXML* в более полной системе, можно найти в примере для класса *XMLSocket*.

***См. также***

*XMLSocket.send*, *XMLSocket.onData( )*; раздел «Прикрепление обработчиков событий к другим объектам» в главе 10.

---

**XMLSocket.send( ) метод** передать данные в формате XML приложению сервера

***Доступность*** Flash 5

***Краткий обзор*** `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 объектом класса *XML*, но может также являться строкой. При вызове *send( )* объект `XMLobject` преобразуется в строку и передается удаленному приложению, завершаясь нулевым байтом (первый символ ASCII, null). Удаленное приложение не обязано давать ответ – однако, если ответ прислан, он запускает обработчик события сокета *onXML( )*.

***Пример***

В следующем примере удаленному приложению посылается очень простое сообщение в формате XML через сокет `mySocket`, являющийся действительным объектом *XMLSocket*, для которого уже установлено соединение (обратите внимание, что `message` является объектом *XML*, а не объектом *XMLSocket*; см. в примере в описании класса *XMLSocket* образец полноценного 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 часто бывает достаточно следующего:

```
mySocket.send('<MESSAGE>testing...testing...</MESSAGE>');
```

***См. также***

*XMLSocket.onXML*; *XMLSocket* класс, *XML.send( )*.

# IV

# Приложения

- Приложение A «Ресурсы»
- Приложение B «Набор символов Latin 1 и коды клавиш»
- Приложение C «Обратная совместимость»
- Приложение D «Отличия от ECMA-262 и JavaScript»

# A

# Ресурсы

Ниже перечислены ресурсы, представляющие интерес. См. также адреса сайтов, приведенные в предисловии.

## ActionScript и программирование

*ActionScript: The Definitive Guide , сайт книги*

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

С этого сервера можно загрузить все примеры, имеющиеся в данной книге (это Code Depot – хранилище кода), а также десятки других общих примеров ActionScript. Также здесь содержатся изменения в книге, технические заметки, некоторые главы, замеченные ошибки и новости. Поддерживается автором.

*moockmarks*

*http://www.moock.org/moockmarks*

Закладки, сделанные автором. Перечислены сотни сайтов с технической информацией и вдохновляющими примерами дизайна для Flash и Сети в целом.

*Центр поддержки Macromedia Flash*

*http://www.macromedia.com/support/flash*

Быстро растущее собрание информации по ActionScript и Flash. Содержит полный текст Macromedia ActionScript Dictionary и регулярно обновляемую базу данных технических замечаний для разработчиков на Flash. Ознакомительную версию среды разработки Flash можно загрузить по адресу: *http://www.macromedia.com/software/flash/trial*.

*Macromedia Exchange for Flash*

*http://www.macromedia.com/exchange/flash*

Коллекция интеллектуальных клипов профессионального качества, файлов проектов *.fla* и примеров на ActionScript. Содержимое поддерживается и одобрено Macromedia, но является результатом совместных разработок сообщества Flash. Для доступа к примерам необходимо средство инсталляции под названием Extension Manager.

*Flash Kit*

*http://www.flashkit.com*

Flash Kit компании Internet.com является крупнейшим источником информации по Flash для сторонних разработчиков. На сайте представлено все, что может иметь отношение к Flash: файлы примеров, доски объявлений, учебники, новости, тематические статьи, аркадная игра, интервью, чат, галерея и многое другое. Особый интерес для программистов Flash представляют файлы примеров ActionScript.*fla*, которые можно загрузить с адреса *http://www.flashkit.com/movies/Scripting*.

*Ultrashock*

*http://www.ultrashock.com*

Общий портал для разработчиков Flash, на котором Ultrashock поддерживает яркое собрание сосредоточенных на ActionScript файлов *.fla* и учебных материалов.

*Flashcoders, почтовый список рассылки*

*http://chattyfig.figleaf.com/mailman/listinfo/flashcoders*

Поддерживаемый Бранденом Холлом (Branden Hall) из Fig Leaf Software, этот почтовый список рассылки для активных разработчиков Flash нацелен главным образом на программистов средней и высокой подготовки в ActionScript. Здесь можно узнать практические приемы и получить ответы на вопросы от коллег, но это *не* место для общих вопросов по Flash или site checks. Можно производить поиск в архивах, которые обновляются ежемесячно. Прежде чем посылать вопрос, поищите ответ в архивах.

*Brandon Williams's and Ethan Kennedy's Flash Experiments*

*http://www.homepages.go.com/~ahab_flash/exper/index.htm*

Большое собрание продвинутых демонстрационных примеров ActionScript с использованием математики. Имеются десятки доступных для изучения примеров файлов *.fla*, охватывающих трехмерную графику, фракталы и многие зрительные физические эффекты. Содержит серьезные статьи об использовании математики в программировании на Flash.

*FAQ телеконференции comp.graphics.algorithms*

*http://www.faqs.org/faqs/graphics/algorithms-faq*

Описывает математические решения для широкого круга задач программирования графики. Диапазон тем включает в себя как простые задачи типа «как найти расстояние от точки до прямой», так и сложные вопросы трехмерной графики, обычно находящиеся за пределами программирования на Flash.

*Архив плееров Macromedia Flash*

*http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm*

Загрузка старых версий Macromedia Flash Player, нужных для тестирования.

*Общие ресурсы программирования игр*

На следующих сайтах есть полезная информация для проектирования, планирования и разработки игр (эти сайты не специализируются на ActionScript):

*http://www.gamedev.net*

*http://www.flipcode.com*

*http://www.chesworth.com/pv/games*

*http://www.javascript-games.org*

*Flash и Active Server Pages (ASP)*

Следующие статьи ASP 101 дают хорошее введение в интеграцию Flash и ASP при создании простых форм Flash:

*http://www.asp101.com/articles/flash*

*http://www.asp101.com/articles/flash2*

*Macromedia Generator*

Обсуждение Macromedia Generator выходит за рамки данной книги, однако на следующих сайтах представлены отличные ресурсы по разработке Generator:

*http://www.markme.com*

*http://www.gendev.net*

# Ресурсы ECMA–262

Поскольку ActionScript и JavaScript основываются на спецификации ECMA-262 для языков сценариев, они имеют одинаковый базовый синтаксис. Поэтому, хотя следующие ресурсы не посвящены специально ActionScript, их стоит изучить программистам ActionScript:

*Спецификация языка ECMA-262*

*http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM*

Официальная спецификация языка ECMA-262, принятая ECMA – организацией, поддерживающей язык, на котором основывается ActionScript. JavaScript компании Netscape и Jscript компании Microsoft являются двумя другими реализациями ECMA-262. Этот чисто технический документ может использоваться при решении таких задач, как разработка интерпретатора, удовлетворяющего ECMA.

*Руководство Netscape по основам JavaScript*

*http://developer.netscape.com/docs/manuals/js/core/jsguide/contents.htm*

В этом подробном изложении основных характеристик языка JavaScript Netscape предоставляет очень понятную документацию по JavaScript.

## Объектно-ориентированное программирование

*Описание объектной модели Netscape*

*http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm*

Подробная статья, излагающая реализацию ООП на основе прототипов, наследование и классы в JavaScript. Понятия иллюстрируются многочисленными примерами кода. Netscape прилагает особые старания, чтобы разъяснить ООП в JavaScript на языке, понятном программистам на C и Java.

*Концепция Sun объектно-ориентированного программирования (из The Java™ Tutorial)*

*http://java.sun.com/docs/books/tutorial/java/concepts*

Предназначенный в качестве учебника для начинающих программистов Java, этот документ освещает основы ООП в обобщенных понятиях и представляет собой хорошее чтение для базовой подготовки по ООП для любого языка.

## Формат файла SWF

Исходный код Flash Player не является открытым, но сам формат файла *.swf* общедоступен (что является одной из причин широкой поддержки его сторонними разработчиками). Вот некоторые ресурсы, содержащие дополнительную информацию:

*Формат файла Macromedia Flash Player (SWF)*

*http://www.macromedia.com/software/flash/open/licensing/fileformat*

Здесь можно получить SWF SDK от Macromedia – «набор инструментов разработчика для создания файлов Macromedia Flash 5 (SWF), документация по формату файла Macromedia Flash (SWF) и код для создания файлов SWF».

*OpenSWF*

*http://www.openswf.org*

Центр общей информации для разработчиков реализаций SWF. Содержит учебники, доски объявлений, ресурсы и исходный код.

*Ming*

*http://www.opaque.net/ming*

Ming является библиотекой C с открытым кодом для создания фильмов в формате SWF и с набором оболочек для использования библиотеки в C++ и популярных языках сценариев типа PHP, Python и Ruby. Сведения о программном обеспечении с открытым кодом см. по адресу *http://www.opensource.org*.

*Perl :: Flash*

*http://www.2shortplanks.com/flash*

Библиотека Perl для «динамического, программируемого создания фильмов Flash/SWF». Разработчик Simon Wistow.

# B

# Набор символов Latin 1 и коды клавиш

В табл. B.1 перечислены символы из набора Latin 1 – основного набора символов, поддерживаемого Flash. В первой колонке (помеченной «Dec») дается кодовая позиция каждого символа в десятичном виде (стандартное значение ASCII), во второй колонке дается escape-последовательность Unicode для символа, а в третьей колонке описывается сам символ. Дополнительная информация о кодировке символов в Flash есть в главе 4 «Элементарные типы данных».

Дополнительными ресурсами по теме кодировки символов могут служить следующие:

*The ISO 8859 Alphabet Soup*

*http://czyborra.com/charsets/iso8859.html*

Ряд документов, подробно описывающих состав и смысл символов в наборе Latin 1 – основном наборе символов, поддерживаемом Flash (поддержку осуществляет Roman Czyborra)

*Shift-JIS Code Points*

*ftp://ftp.unicode.org/Public/MAPPINGS/EASTASIA/JIS/SHIFTJIS.TXT*

Список кодовых позиций Unicode для символов в наборе Shift-JIS – кодировке японских символов, поддерживаемой Flash

*Unicode FAQ*

*http://www.unicode.org/unicode/faq*

Хороший обзор Unicode – международного стандарта кодировки символов, в формате «вопрос-ответ»

*Таблица В.1. Символы ISO 8859-1 (Latin 1) и их соответствия в Unicode*

| Dec | Unicode | Описание | Dec | Unicode | Описание |
|---|---|---|---|---|---|
| 0 | \u0000 | [null] | 29 | \u001d | [разделитель групп] |
| 1 | \u0001 | [начало заголовка] | 30 | \u001e | [разделитель записей] |
| 2 | \u0002 | [начало текста] | 31 | \u001f | [разделитель элементов] |
| 3 | \u0003 | [конец текста] | 32 | \u0020 | [пробел] |
| 4 | \u0004 | [конец передачи] | 33 | \u0021 | ! |
| 5 | \u0005 | [запрос] | 34 | \u0022 | " |
| 6 | \u0006 | [подтверждение] | 35 | \u0023 | # |
| 7 | \u0007 | [звонок] | 36 | \u0024 | $ |
| 8 | \u0008 | [забой] | 37 | \u0025 | % |
| 9 | \u0009 | [горизонтальная табуляция] | 38 | \u0026 | & |
| 10 | \u000a | [перевод строки] | 39 | \u0027 | ' |
| 11 | \u000b | [вертикальная табуляция] | 40 | \u0028 | ( |
| 12 | \u000c | [подача листа] | 41 | \u0029 | ) |
| 13 | \u000d | [возврат каретки] | 42 | \u002a | * |
| 14 | \u000e | [расширение кода] | 43 | \u002b | + |
| 15 | \u000f | [восстановление кода] | 44 | \u002c | , |
| 16 | \u0010 | [переключение] | 45 | \u002d | - |
| 17 | \u0011 | [управление устройством 1] | 46 | \u002e | . |
| 18 | \u0012 | [управление устройством 2] | 47 | \u002f | / |
| 19 | \u0013 | [управление устройством 3] | 48 | \u0030 | 0 |
| 20 | \u0014 | [управление устройством 4] | 49 | \u0031 | 1 |
| 21 | \u0015 | [отрицание] | 50 | \u0032 | 2 |
| 22 | \u0016 | [синхронизация] | 51 | \u0033 | 3 |
| 23 | \u0017 | [конец передачи] | 52 | \u0034 | 4 |
| 24 | \u0018 | [аннулирование] | 53 | \u0035 | 5 |
| 25 | \u0019 | [конец носителя] | 54 | \u0036 | 6 |
| 26 | \u001a | [замена] | 55 | \u0037 | 7 |
| 27 | \u001b | [переключение кода] | 56 | \u0038 | 8 |
| 28 | \u001c | [разделитель файлов] | 57 | \u0039 | 9 |

*Таблица B.1 (продолжение)*

| Dec | Unicode | Описание | Dec | Unicode | Описание |
|---|---|---|---|---|---|
| 58 | \u003a | : | 91 | \u005b | [ |
| 59 | \u003b | ; | 92 | \u005c | \ |
| 60 | \u003c | < | 93 | \u005d | ] |
| 61 | \u003d | = | 94 | \u005e | ^ |
| 62 | \u003e | > | 95 | \u005f | _ |
| 63 | \u003f | ? | 96 | \u0060 | ` |
| 64 | \u0040 | @ | 97 | \u0061 | a |
| 65 | \u0041 | A | 98 | \u0062 | b |
| 66 | \u0042 | B | 99 | \u0063 | c |
| 67 | \u0043 | C | 100 | \u0064 | d |
| 68 | \u0044 | D | 101 | \u0065 | e |
| 69 | \u0045 | E | 102 | \u0066 | f |
| 70 | \u0046 | F | 103 | \u0067 | g |
| 71 | \u0047 | G | 104 | \u0068 | h |
| 72 | \u0048 | H | 105 | \u0069 | i |
| 73 | \u0049 | I | 106 | \u006a | j |
| 74 | \u004a | J | 107 | \u006b | k |
| 75 | \u004b | K | 108 | \u006c | l |
| 76 | \u004c | L | 109 | \u006d | m |
| 77 | \u004d | M | 110 | \u006e | n |
| 78 | \u004e | N | 111 | \u006f | o |
| 79 | \u004f | O | 112 | \u0070 | p |
| 80 | \u0050 | P | 113 | \u0071 | q |
| 81 | \u0051 | Q | 114 | \u0072 | r |
| 82 | \u0052 | R | 115 | \u0073 | s |
| 83 | \u0053 | S | 116 | \u0074 | t |
| 84 | \u0054 | T | 117 | \u0075 | u |
| 85 | \u0055 | U | 118 | \u0076 | v |
| 86 | \u0056 | V | 119 | \u0077 | w |
| 87 | \u0057 | W | 120 | \u0078 | x |
| 88 | \u0058 | X | 121 | \u0079 | y |
| 89 | \u0059 | Y | 122 | \u007a | z |
| 90 | \u005a | Z | 123 | \u007b | { |

*Таблица B.1 (продолжение)*

| Dec | Unicode | Описание | Dec | Unicode | Описание |
|---|---|---|---|---|---|
| 124 | \u007c | \| | 157 | \u009d | управляющий символ |
| 125 | \u007d | } | 158 | \u009e | управляющий символ |
| 126 | \u007e | ~ | 159 | \u009f | управляющий символ |
| 127 | \u007f | [удаление] | 160 | \u00a0 | [неразрывный пробел] |
| 128 | \u0080 | управляющий символ | 161 | \u00a1 | • |
| 129 | \u0081 | управляющий символ | 162 | \u00a2 | • |
| 130 | \u0082 | управляющий символ | 163 | \u00a3 | • |
| 131 | \u0083 | управляющий символ | 164 | \u00a4 | ¤ |
| 132 | \u0084 | управляющий символ | 165 | \u00a5 | • |
| 133 | \u0085 | управляющий символ | 166 | \u00a6 | ¦ |
| 134 | \u0086 | управляющий символ | 167 | \u00a7 | § |
| 135 | \u0087 | управляющий символ | 168 | \u00a8 | • |
| 136 | \u0088 | управляющий символ | 169 | \u00a9 | © |
| 137 | \u0089 | управляющий символ | 170 | \u00aa | • |
| 138 | \u008a | управляющий символ | 171 | \u00ab | « |
| 139 | \u008b | управляющий символ | 172 | \u00ac | ¬ |
| 140 | \u008c | управляющий символ | 173 | \u00ad | - |
| 141 | \u008d | управляющий символ | 174 | \u00ae | ® |
| 142 | \u008e | управляющий символ | 175 | \u00af | • |
| 143 | \u008f | управляющий символ | 176 | \u00b0 | ° |
| 144 | \u0090 | управляющий символ | 177 | \u00b1 | ± |
| 145 | \u0091 | управляющий символ | 178 | \u00b2 | 2 |
| 146 | \u0092 | управляющий символ | 179 | \u00b3 | 3 |
| 147 | \u0093 | управляющий символ | 180 | \u00b4 | • |
| 148 | \u0094 | управляющий символ | 181 | \u00b5 | µ |
| 149 | \u0095 | управляющий символ | 182 | \u00b6 | ¶ |
| 150 | \u0096 | управляющий символ | 183 | \u00b7 | · |
| 151 | \u0097 | управляющий символ | 184 | \u00b8 | • |
| 152 | \u0098 | управляющий символ | 185 | \u00b9 | 1 |
| 153 | \u0099 | управляющий символ | 186 | \u00ba | • |
| 154 | \u009a | управляющий символ | 187 | \u00bb | » |
| 155 | \u009b | управляющий символ | 188 | \u00bc | $^1/_4$ |
| 156 | \u009c | управляющий символ | 189 | \u00bd | $^1/_2$ |

*Таблица B.1 (продолжение)*

| Dec | Unicode | Описание | Dec | Unicode | Описание |
|---|---|---|---|---|---|
| 190 | \u00be | ¾ | 223 | \u00df | • |
| 191 | \u00bf | • | 224 | \u00e0 | • |
| 192 | \u00c0 | • | 225 | \u00e1 | • |
| 193 | \u00c1 | • | 226 | \u00e2 | • |
| 194 | \u00c2 | • | 227 | \u00e3 | • |
| 195 | \u00c3 | • | 228 | \u00e4 | • |
| 196 | \u00c4 | • | 229 | \u00e5 | • |
| 197 | \u00c5 | • | 230 | \u00e6 | • |
| 198 | \u00c6 | • | 231 | \u00e7 | • |
| 199 | \u00c7 | • | 232 | \u00e8 | • |
| 200 | \u00c8 | • | 233 | \u00e9 | • |
| 201 | \u00c9 | • | 234 | \u00ea | • |
| 202 | \u00ca | • | 235 | \u00eb | • |
| 203 | \u00cb | • | 236 | \u00ec | • |
| 204 | \u00cc | • | 237 | \u00ed | • |
| 205 | \u00cd | • | 238 | \u00ee | • |
| 206 | \u00ce | • | 239 | \u00ef | • |
| 207 | \u00cf | • | 240 | \u00f0 | ð |
| 208 | \u00d0 | Ð | 241 | \u00f1 | • |
| 209 | \u00d1 | • | 242 | \u00f2 | • |
| 210 | \u00d2 | • | 243 | \u00f3 | • |
| 211 | \u00d3 | • | 244 | \u00f4 | • |
| 212 | \u00d4 | • | 245 | \u00f5 | • |
| 213 | \u00d5 | • | 246 | \u00f6 | • |
| 214 | \u00d6 | • | 247 | \u00f7 | • |
| 215 | \u00d7 | • | 248 | \u00f8 | • |
| 216 | \u00d8 | • | 249 | \u00f9 | • |
| 217 | \u00d9 | • | 250 | \u00fa | • |
| 218 | \u00da | • | 251 | \u00fb | • |
| 219 | \u00db | • | 252 | \u00fc | • |
| 220 | \u00dc | • | 253 | \u00fd | ý |
| 221 | \u00dd | Ý | 254 | \u00fe | þ |
| 222 | \u00de | Þ | 255 | \u00ff | • |

В табл. В.2 перечислены коды некоторых специальных клавиш на клавиатуре. Эти коды клавиш используются только с объектом *Key* (см. часть III «Справочник по языку»).

*Таблица В.2. Коды специальных клавиш объекта Key*

| Клавиша | Код клавиши | Клавиша | Код клавиши |
|---|---|---|---|
| ;: | 186 | Home | 36 |
| += | 187 | Insert | 45 |
| -_ | 189 | Left arrow | 37 |
| /? | 191 | Num Lock | 144 |
| `~ | 192 | Num Pad 0 | 96 |
| [{ | 219 | Num Pad 1 | 97 |
| \| | 220 | Num Pad 2 | 98 |
| ]} | 221 | Num Pad 3 | 99 |
| “ ’ | 222 | Num Pad 4 | 100 |
| Alt | 18 (не перехватывается) | Num Pad 5 | 101 |
| Backspace | 8 | Num Pad 6 | 102 |
| Caps Lock | 20 | Num Pad 7 | 103 |
| Control | 17 | Num Pad 8 | 104 |
| Delete | 46 | Num Pad 9 | 105 |
| Down arrow | 40 | Num Pad * | 106 |
| End | 35 | Num Pad + | 107 |
| Enter | 13 | Num Pad Enter | 13 (не перехватывается как 108) |
| Escape | 27 | Num Pad - | 109 |
| F1 | 112 | Num Pad . | 110 |
| F2 | 113 | Num Pad / | 111 |
| F3 | 114 | Page Down | 34 |
| F4 | 115 | Page Up | 33 |
| F5 | 116 | Pause/Break | 19 |
| F6 | 117 | Print Screen | 44 |
| F7 | 118 | Right Arrow | 39 |
| F8 | 119 | Scroll Lock | 145 |
| F9 | 120 | Shift | 16 |
| F10 | 121 (не перехватывается) | Space | 32 |
| F11 | 122 | Tab | 9 |
| F12 | 123 | Up Arrow | 38 |

В табл. В.3 перечислены коды буквенных и цифровых клавиш на клавиатуре. Эти коды клавиш используются только с объектом *Key* (см. часть III ).

*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

Flash позволяет публиковать файлы *.swf* в формате, совместимом с прежними версиями проигрывателя. Когда вы будете это читать, у большинства пользователей уже будет установлен подключаемый модуль Flash версии не ниже 5, но, может быть, вы решите создать фильм для модуля Flash 4. Даже если вы создаете документы исключительно для Flash 5, это приложение поможет вам избежать использования устаревших элементов ActionScript и прийти к использованию методов, предпочтительных для Flash 5. Статистику распространенности различных версий Flash Player можно найти по адресу:

*http://www.macromedia.com/software/flash/survey/whitepaper*

При создании документов для Flash 4 нужно использовать устаревшие (*deprecated*) технологии (т. е. те, которые устарели, но поддерживаются для обратной совместимости), перечисленные в табл. C.1. См. также раздел «Using Flash 5 to Create Flash 4 Content» в главе «Writing Scripts with ActionScript» справочника Macromedia «Flash ActionScript Reference Guide».

Для того чтобы сценарии выполнялись во Flash 4 Player, необходимо на вкладке Flash в пункте File → Publish Settings установить Flash 4 в качестве Version для файла *.swf*. Иначе при попытке выполнения во Flash 4 Player кода в файле *.swf*, созданном во Flash 5 или более позднем, весь код будет отключен.

Основные вопросы обратной совместимости и различий между Flash 4 ActionScript и Flash 5 ActionScript сведены в табл. C.1.

*Таблица С.1. Вопросы обратной совместимости*

| Вопрос | Описание |
|---|---|
| Создание переменных | Функция Flash 4 *set* заменена предложением *var*. Для создания динамически именуемых переменных используйте *eval( )* или (еще лучше) используйте для управления данными массивы. См. главу 4 «Массивы». |
| Ссылки на переменные и временные диаграммы | Конструкции в стиле Flash 4 из косой черты и двоеточия (`/square:area`) заменены точечной нотацией (`square.area`). См. табл. 2.1. |
| Операторы сравнения строк | Операторы сравнения строк Flash 4 `eq`, `ne`, `ge`, `gt`, `le`, `lt` заменены во Flash 5 следующими операторами: ==, !=, >=, >, <=, <. См. табл. 4.2. |
| Оператор конкатенации строк | При создании содержимого Flash 4 во Flash 5 и более поздних версиях используйте оператор *add* вместо оператора Flash 4 `&`. При создании документов для Flash 5 используйте для конкатенации строк оператор +. См. табл. 4.2. |
| Длина строки | Функция Flash 4 *length( )* (например, *length(`myString`)*) заменена свойством `length` (например, *`myString`*.`length`). См. табл. 4.2. |
| Извлечение подстроки | Функция Flash 4 *substring( )* (например, `substring (myString,1,3)`) заменена методами *substring( )*, *substr( )* и *slice( )*. Обратите внимание на то, что *substring( )* имеет различия во Flash 4 и Flash 5. См. табл. 4.2. |
| 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( )* (используемые для определения кодовой позиции символа) заменены методом *String.charCodeAt( )*. См. табл. 4.2. |
| Преобразование типа данных | Когда импортируются файлы Flash 4, Flash 5 автоматически вставляет функцию *Number( )*,которая охватывает любые числовые данные, используемые в качестве операнда следующих потенциально неоднозначных операторов: +, ==, !=, <>, <, >, > =, <=. См. табл. 3.5. |
| Предложение *ifFrameLoaded* | Предложение Flash 3 *ifFrameLoaded* стало устаревшим. Для создания кода предварительной загрузки используйте свойства `_totalframes` и `_framesloaded` *MovieClip*. |
| Бесконечные циклы | Flash 4 допускал максимум 200000 итераций цикла. Flash 5 выделяет 15 секунд на цикл, после чего извещает пользователя об отсутствии отклика от фильма. См. раздел «Максимальное число итераций» в главе 8 «Предложения цикла». |
| Сравнение подпрограмм и функций | Во Flash 4 можно было создать подпрограмму, прикрепляя блок кода к кадру с меткой и выполняя его с помощью предложения *call( )*. Функции Flash 5 заменяют подпрограммы Flash 4. |

*Таблица С.1 (продолжение)*

| Вопрос | Описание |
|---|---|
| События клипов | Flash 4 поддерживал только события кнопок (т. е. функции, начинающиеся с *on( )*), показанные в табл. 10.1. События клипов (т.е. *onClipEvent( )*) не могут использоваться во Flash 4 Player. |
| Перехват нажатий клавиш | Во Flash 4 единственным средством перехвата нажатий клавиш был *keyPress*. Во Flash 5 объект *Key* с сочетании с событиями клипов *keyDown* и *keyUp* предоставляет значительно больший контроль над взаимодействием с клавиатурой. |
| *Tell Target* устарела | *Tell Target* из Flash 4 (используемая для управления удаленными клипами фильмов) заменена свойствами и методами, доступ к которым осуществляется с помощью точечной нотации и предложения *with*. См. главу 13 «Клипы». |
| *Get Property* устарела | Команда Flash 4 *Get Property* более не является необходимой для доступа к свойствам клипа фильма. Используйте вместо нее оператор точки. См. главу 13. |
| *int* устарела | Функция Flash 4 *int( )* (использовавшаяся для усечения чисел с плавающей запятой до целых) заменена функциями *Math.floor( )*, *Math.ceil( )* и *Math.round( )*. |
| Генерирование случайных чисел | Функция Flash 4 *random( )* (использовавшаяся для генерации случайного числа) заменена *Math.random( )*. |
| *toggleHighQuality* устарела | Функция Flash 4 *toggleHighQuality* (использовавшаяся для установки качества 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 4 Player. Однако значения аппроксимируются при экспорте в формате Flash 4. |
| *loadMovie* и *loadMovieNum* | Функция Flash 3 *loadMovie( )* загрузки фильма на уровень с заданным номером заменена на функцию Flash 5 *loadMovieNum( )* (которая принимает целочисленный аргумент уровня). Однако функция Flash 4 загрузки в целевой клип *loadMovie( )* по-прежнему доступна во Flash 5 как *loadMovie( )*. |
| Печать | Flash 5 поддерживает встроенную функцию *print( )*, которая во Flash 4 Build 20 и позднее была доступна только как модифицированное действие *Get URL*. |
| Не поддерживаемые объекты и классы | Flash 4 не поддерживает объекты и классы, встроенные во Flash 5. |

# Изменения во Flash 5 Player, Build 41

В следующем списке собраны основные изменения, сделанные в Build 41 (Netscape) и 42 (Internet Explorer) Flash 5 Player (предыдущей общедоступной сборкой была 30-я, первоначально выпущенная со средством редактирования Flash 5):

- Курсор в текстовом поле принимает цвет текста.
- Фильмы в ячейке таблицы не приводят к аварийному завершению Internet Explorer 5.5.
- Положение `scroll` 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( )* не возвращает значение 1. Максимальным возвращаемым значением является 0.999.
- Общая производительность улучшена, особенно под Windows 98.

# Управление клипами

В главе 13 было рассказано об управлении клипами с помощью технологий Flash 5. Сейчас будет рассмотрена эквивалентная технология Flash 4.

До появления Flash 5 для управления клипом фильма требовалось выполнить специальные *Movie Clip Actions* (*действия клипа*). Нужно 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 нужно было использовать явные команды получения и задания свойств, например:

```
GetProperty ("ball", _width)
Set Property ("ball", X Scale) = 90
```

Во Flash 5 стало возможно 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 задания и получения значений переменных используется оператор точки:

```
myClip.myVariable = 14;
x = myClip.myVariable;
```

Наконец, до Flash 5 обращение к вложенным уровням клипов производилось с использованием метафоры дерева каталогов с символами косой черты и точки:

```
clipA/clipB/clipC
../../../clipC
```

Поскольку во Flash 5 клипы являются данными сходного с объектами вида, можно хранить один клип как свойство другого. Поэтому для доступа к так называемым *вложенным* (*nested*) клипам используется оператор точки, а с помощью зарезервированного свойства клипа `_parent` осуществляется ссылка на содержащий его клип:

```
clipA.clipB.clipC;
_parent._parent._parent.clipC;
```

# D

# Отличия от ECMA–262 и JavaScript

Данная книга, естественно, сосредоточена на сложностях ActionScript, но если у вас имеется отличная книга издательства O'Reilly «JavaScript: The Definitive Guide», то вы обнаружите 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 Player и для обеспечения обратной совместимости. При переносе кода из JavaScript, Jscript или другого основанного на ECMA-262 языка большую пользу окажет табл. D.1. В ней собраны вместе умышленно сделанные различия между ECMA-262, JavaScript и Flash 5 ActionScript. Аналогично, при переносе ActionScript на другой язык вам будет легче избежать ловушек, вызванных отклонениями ActionScript от стандарта ECMA-262.

Табл. D.1 отражает *умышленные* расхождения между Flash 5 ActionScript и стандартом ECMA-262. В ней не указаны ошибки, которые могли быть сделаны при попытке реализации стандарта.

*Таблица D.1. Расхождения между ECMA-262, JavaScript и ActionScript*

| Вопрос | Описание |
|---|---|
| Преобразование строк в булевы значения | В ECMA-262 любая непустая строка преобразуется в `true`. Во Flash 5 в `true` преобразуются только строки, которые могут быть преобразованы в допустимые числа, отличные от нуля. |
| Чувствительность к регистру | В спецификации ECMA-262 требуется полная чувствительность к регистру. В ActionScript к регистру чувствительны только ключевые слова, но не идентификаторы. См. главу 14 «Лексическая структура», в особенности табл. 14.1. |

*Таблица D.1 (продолжение)*

| Вопрос | Описание |
|---|---|
| Область видимости функций | 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 нельзя изменить область видимости функции путем присвоения: она задается постоянной и зависит от местонахождения объявления функции. |
| Регулярные выражения | ActionScript не поддерживает регулярные выражения. |
| Имена обработчиков событий | В ActionScript только обработчики событий объектов имеют собственные именованные функции (например, *onLoad( )* объекта *XML*). Обработчики событий клипов определяются с помощью *onClipEvent (*`eventName`*)*, а обработчики событий кнопок определяются с помощью *on (*`eventName`*)*. См. «Синтаксис обработчиков событий» в главе 10 «События и обработчики событий». |
| Глобальные переменные | ActionScript не поддерживает переменные, подлинно глобальные для всего документа. Глобальные переменные могут моделироваться путем прикрепления свойств к `Object.prototype`, как описано в разделе «Конец цепочки наследования» в главе 12 «Объекты и классы» |
| Функция *eval( )* | В ActionScript функция *eval( )* поддерживает лишь небольшую часть функциональности, предусматриваемой ECMA-262: она работает, только когда ее аргументом является идентификатор, и используется только для динамической генерации ссылок на идентификаторы. |
| Преобразование данных типа `undefined` | В ActionScript особое значение `undefined` преобразуется в пустую строку (`""`) при использовании в строковом контексте, и в число 0 при использовании 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*, используемый в JavaScript для создания функций с помощью синтаксиса `new Function ();`. |
| Создание объекта *Date* | При создании нового объекта *Date* ActionScript не принимает (т. е. не может разобрать) строку даты, читаемую человеком – например, такую как «January 9, 2001». |
| Предложение *switch* | ActionScript не поддерживает предложения *switch*/*case*/*default* (используемые для формулировки сложных условных операторов). См. раздел «Эмуляция предложения switch» в главе 7 «Условные предложения». |

*Таблица D.1 (продолжение)*

| Вопрос | Описание |
|---|---|
| Поддержка языков | ECMA-262 требует поддержки стандарта кодировки символов Unicode, а ActionScript его не поддерживает. ActionScript использует наборы символов Latin 1 и Shift-JIS и реализует некоторое подмножество функций и соглашений в стиле Unicode (например, escape-последовательности `\u`). |
| Объектная модель | Естественно, что JavaScript содержит встроенные классы и объекты, относящиеся к веб-броузерам, в то время как классы и объекты Flash относятся к фильмам Flash. Для программистов JavaScript, привыкших к работе с DHTML, может оказаться полезным представлять себе главный фильм документа Flash как аналог объекта документа HTML, а клипы фильмов – как аналоги объектов слоев. |
| Выполнение кода по времени | Методы *setTimeout()* и *setInterval()* объекта окна в JavaScript отсутствуют в ActionScript, но могут быть смоделированы с помощью циклов событий временных диаграмм и клипов, как показано в главе 8 «Предложения цикла». |
| 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. |

# Алфавитный указатель

## Специальные символы


!, оператор логического отрицания (Flash 5), 157
!=, оператор неравенства (Flash 5), 112, 147
% , оператор деления по модулю, 143
%=, оператор деления по модулю с присвоением, 136
& (амперсанд), оператор конкатенации (Flash 4), 93, 128
&&, оператор логического И (Flash 5), 156
&, поразрядный оператор И, 395
&=, поразрядный оператор И с присвоением, 136
() (круглые скобки)
  оператор вызова функции, 163, 207, 307
  оператор группирования, 159
  приоритеты операторов и, 134
  сравнение присвоения метода с вызовом, 311
* (умножение), оператор, 103, 141
*=, оператор умножения с присвоением, 136
+ (плюс), оператор сложения, 34, 103, 139
  преобразование данных в строки, 91
++, оператор инкрементирования, 139
+=, оператор сложения с присвоением, 110
, (запятая), оператор, 159
– (минус), оператор вычитания, 35, 103, 140
–, унарный оператор отрицания, 143
– –, оператор декрементирования, 140
. (точка), оператор, 162
  доступ к свойствам клипа, 331
  обращение к свойствам объектов, 306
  обход иерархии экземпляров клипов, 351
/* */, маркеры комментариев, 384
/, оператор деления, 103, 141
//, маркеры комментариев, 384
/=, оператор деления с присвоением, 136
; (точка с запятой)
  завершение предложений с помощью, 33, 208, 382
  отсутствие, 470
< (меньше), оператор (Flash 5), 35
<<, оператор поразрядного сдвига влево, 401
<<=, оператор поразрядного сдвига влево с присвоением, 136
<>, оператор неравенства (Flash 4), 148
= (знак равенства), оператор присвоения, 138
= =, оператор равенства (Flash 5), 111, 144
?:, условный оператор, 163, 181
[], оператор элемента массива/свойства объекта, 161
  обращение к элементам массива, 278
  доступ к свойству объекта, 306
  динамические ссылки на объекты клипов, 357
^, поразрядный оператор исключающего ИЛИ, 398
^=, поразрядный оператор исключающего ИЛИ с присвоением, 136
{} (фигурные скобки), инициализаторы объекта/ограничители блока команд/предложений), 42, 167

|, поразрядный оператор ИЛИ, 396
|=, поразрядный оператор ИЛИ с присвоением, 136
||, оператор логического ИЛИ (Flash 5), 154
~, поразрядный оператор отрицания, 399
', одинарные кавычки в строках, 106
> (больше), оператор (Flash 5), 112, 149
>= (больше или равно), оператор (Flash 5), 150
>>, оператор поразрядного сдвига вправо со знаком, 400
>>=, оператор поразрядного сдвига вправо со знаком с присвоением, 136
>>>, оператор поразрядного сдвига вправо без знака, 401
>>>=, оператор поразрядного беззнакового сдвига вправо с присвоением, 136

## A

<A>, тег HTML, 452
  вызов функций ActionScript из, 457
  выполнение JavaScript из, 457
  не добавляется в обход по Tab, 453
abs() (Math, объект), 563
abstract, ключевое слово (потенциально зарезервированное слово), 386
acos() (Math, объект), 563
Actions, панель
  Expert Mode, 413
  Normal Mode, 412
    нет обратной совместимости, 413
    переключение из Expert Mode, 414
  Parameters, окно, 413
  Script, окно, 411
    добавление кода в кнопки, 415
    добавление предложений в, 412
  Toolbox, окно, 411
  открытие, 28
  перекрытие методов и глобальных функций, 369
  режимы редактирования, 412
    установка, 413
  способы нахождения кода, 417
ActionScript
  аргументы, 34
  базовый класс в, 627
  веб-сайты, использующие, 18
  выражения, 37
  зарезервированные слова, 385
  звук, управление, 648, 661
  идентификаторы, правила образования, 387
  импорт текста из внешних файлов, 419, 543
  интерактивность в, 16
  интерпретатор
    использование цепочки областей видимости, 217
    преобразование данных, 86, 146
    символы завершения предложений и, 382
    синхронное выполнение кода, 235
    удаление локальных переменных, 78
  лексическая структура, 380
  максимальные/минимальные значения, 102, 624
  модель выполнения, управляемого событиями, 43, 236
  обмен данными с сервером, 17
  операторы, 34
  отключение автопилота, 30
  переменные (см. переменные), 31
  предложения (см. предложения), 30
    условные, 38
  проблемы обратной совместимости, 743
  программирование, введение в, 25
  различия между EMCA-262 и JavaScript, 748
  ресурсы, 731, 733
  связь с JavaScript, 453, 457
    fscommand() и, 527, 531
    getURL() и, 533, 537
  справочник по языку, 477
  среда редактирования (см. среда редактирования), 411
  сценарии, 37
  типы данных, 82
  управление
    визуальным содержимым, 17
    временной диаграммой, 16
    звуковым содержимым, 17
  функции, 41
  цвет, программирование, 495, 499

циклы, 39
чувствительность к регистру в, 387, 471
эквивалентные свойства с Java, 326
Active Server Pages (ASP), ресурсы, 733
allowscale, команда (автономный Player), 528
_alpha, свойство (MovieClip, класс), 366, 582, 586
Alpha, составляющая цветов, 495, 504
and, логическое И, оператор (Flash 4), 386
appendChild() (XML, класс), 682, 689
arguments, объект, 480
доступ к лишним параметрам, 221
извлечение значений параметров из массива, 222
arguments.callee, свойство, 223, 481
arguments.length, свойство, 481
Array(), конструктор, 276
Array, класс, 482
array, тип данных, 84
Array.concat(), 288, 483
Array.join(), 296, 484
Array.length, свойство, 280, 485
расширение массивов без присвоения значений элементам, 284
удаление элементов массива, 289
Array.pop(), 290, 486
Array.push(), 284, 486
Array.reverse(), 293, 487
Array.shift(), 291, 487
Array.slice(), 146, 295, 487
Array.sort(), 293, 488
Array.splice(), 287, 292, 489
Array.toString(), 297, 490
Array.unshift(), 285, 491
ASCII
значения клавиш, 550, 736, 742
набор символов, 105
символ null, 717
автоматическое выполнение обработчика обратного вызова (свойство onXML), 727
обработчика события onData(), 726
asin() (Math, объект), 564
ASP (Active Server Pages), ресурсы, 733
atan() (Math, объект), 565
atan2() (Math, объект), 565, 566
attachMovie() (MovieClip, класс), 368, 583, 587
добавление созданных клипов в стек, 344
создание экземпляров, 338
attachSound() (Sound, класс), 649
ATTRIBUTE_NODE (тип узла XML), 703
attributes, свойство (XML, класс), 681, 690


## B


<B>, тег HTML, 450
Backspace, клавиша, 250
BACKSPACE, свойство (Key, объект), 548
Bandwidth Profiler (средство отладки), 464
Beck, Kent, 472
Boolean(), глобальная функция, 92, 491
Boolean, класс, 492
boolean, ключевое слово (потенциально зарезервированное слово), 386
Boolean, тип данных, 83
правила равенства для, 145
преобразование
значений в, 491
типов данных в, 89, 179
целые числа, 129
Boolean.toString(), 493
Boolean.valueOf(), 493
Border/Bg, опция текстовых полей, 444
<BR>, тег HTML, 452
break, предложение, 168, 195
byte, ключевое слово (потенциально зарезервированное слово), 386


## C


call( )
предложение, 168, 173
глобальная функция, 494
callee, свойство (arguments, объект), 223, 481
Caps Lock, клавиша и isToggled(), 553
CAPSLOCK, свойство (Key, объект), 548
case, ключевое слово (потенциально зарезервированное слово), 386
case, предложение, эмуляция, 183

case, чувствительность в ActionScript, 387
catch, ключевое слово (потенциально зарезервированное слово), 386
CDATA в исходном коде XML, 708
CDATA_SECTION_NODE (тип узла XML), 703
ceil() (Math, объект), 566
ch_020801132602, 335
_changed, событие (текстовые поля), 449
char, ключевое слово (потенциально зарезервированное слово), 386
charAt() (String, класс), 116, 664
charCodeAt() (String, класс), 126, 665
Check Syntax, команда, 461
childNodes, свойство (XML, класс), 681, 691
chr() (Flash 4), 128
class, ключевое слово (потенциально зарезервированное слово), 386
cloneNode() (XML, класс), 682, 692
close() (XMLSocket, класс), 722
Color, объекты класса, 495, 499
  setTransform() и, 503
  применение трансформаций к клипам, 496
  создание, 495
COLOR, атрибут тега HTML <FONT>, 450
Color.getRGB(), 499
Color.getTransform(), 500
Color.setRGB(), 502
Color.setTransform(), 503
COMMENT_NODE (тип узла XML), 703
composite datatypes, правила равенства для, 145
concat()
  Array, класс, 288, 483
  String класс
    в сравнении с += оператором, 110
  String, класс, 665
connect() (XMLSocket, класс), 722
const, ключевое слово (потенциально зарезервированное слово), 386
constructor, свойство (Object, класс), 320, 628
contentType, свойство (XML, класс), 681, 693
continue, предложение, 168, 196
CONTROL, свойство (Key, объект), 548
copyright, символ, 126
cos() (Math, объект), 567
createElement() (XML, класс), 682, 694
createTextNode() (XML, класс), 682, 695
_currentframe, свойство (MovieClip, класс), 366, 582, 587

## D

data, событие клипа, 255
  обработчик, 584
    loadMovie() и, 257
    loadVariables() и, 256
    порядок выполнения, 265
    создание предварительных загрузчиков, 258
Date объекты, создание, 508
Date(), глобальная функция, 504
Date, класс, 505, 511
Date.getDate(), 511
Date.getDay(), 511
Date.getFullYear(), 511
Date.getHours(), 512
Date.getMilliseconds(), 512
Date.getMinutes(), 512
Date.getMonth(), 512
Date.getSeconds(), 513
Date.getTime(), 513
Date.getTimezoneOffset(), 513
Date.getUTCDate(), 514
Date.getUTCDay(), 514
Date.getUTCFullYear(), 514
Date.getUTCHours(), 514
Date.getUTCMilliseconds(), 515
Date.getUTCMinutes(), 515
Date.getUTCMonth(), 515
Date.getUTCSeconds(), 515
Date.getYear(), 515
Date.setDate(), 516
Date.setFullYear(), 516
Date.setHours(), 517
Date.setMilliseconds(), 517
Date.setMinutes(), 517
Date.setMonth(), 518
Date.setSeconds(), 518
Date.setTime(), 518
Date.setUTCDate(), 519
Date.setUTCFullYear(), 520
Date.setUTCHours(), 520

Date.setUTCMilliseconds(), 520
Date.setUTCMinutes(), 520
Date.setUTCMonth(), 521
Date.setUTCSeconds(), 521
Date.setYear(), 521
Date.toString(), 522
Date.UTC(), 522
Date.valueOf(), 523
Debugger, инструмент, 466
debugger, ключевое слово (потенциально зарезервированное слово), 386
default, ключевое слово (потенциально зарезервированное слово), 386
Delete, клавиша, 250
delete, оператор, 161
именованные элементы массива, удаление, 283
элементы массива, удаление, 289
DELETEKEY, свойство (Key, объект), 548
Director, связь с, 528, 533
DOCTYPE, тег, 696
docTypeDecl, свойство (XML, класс), 681, 695
DOCUMENT_FRAGMENT_NODE (тип узла XML), 703
DOCUMENT_NODE (тип узла XML), 703
DOCUMENT_TYPE_NODE (тип узла XML), 703
double, ключевое слово (потенциально зарезервированное слово), 386
do-while, цикл, 169, 191
Down, клавиша, 250
DOWN, свойство (Key, объект), 548
dragOut, событие кнопки, 249
dragOver, событие кнопки, 249
_droptarget, свойство (MovieClip, класс), 366, 582, 588
duplicateMovieClip()
MovieClip, класс, 368, 583, 589
load, событие клипа и, 254
добавление созданных клипов в стек, 345
копирование обработчиков событий клипов, 267
порождение клипов, 334
создание экземпляров, 336
глобальная функция, 523
target, параметр и, 371

## E

E, свойство (Math, объект), 568
ECMA-262 стандарт, 12
ресурсы по объектно-ориентированному программированию, 327
ECMA-262, стандарт
оператор void и, 160
потенциально зарезервированные слова, 386
различия между ActionScript и JavaScript, 748
ресурсы, 733
требования к регистру букв, 387
ELEMENT_NODE (тип узла XML), 703
else if, предложение, 182
else, предложение, 180
Embed Fonts, опция текстовых полей, 445, 455
empty, предложение, 169
End, клавиша, 250
END, свойство (Key, объект), 548
Enter, клавиша, 250
отключение через опцию Single Line, 443
перехват, 263
ENTER, свойство (Key, объект), 548
enterFrame, событие клипа, обработчик, 252, 584
перехват комбинаций клавиш, 263
порядок выполнения, 265
проверка нажатых клавиш, 264
создание переносимого предварительного загрузчика, 590
создание циклов с помощью, 201, 334
эмуляция динамической обработки событий, 270
ENTITY_NODE (тип узла XML), 703
ENTITY_REFERENCE_NODE (тип узла XML), 703
enum, ключевое слово (потенциально зарезервированное слово), 386
eq (равенство), оператор (Flash 4), 93, 128
escape(), глобальная функция, 525
Escape, клавиша, перехват, 261
ESCAPE, свойство (Key, объект), 549
escape-последовательности
Unicode, 105, 108, 736, 740

в ActionScript, 107
в строковых литералах, 106
шестнадцатеричные, 108, 525
eval(), глобальная функция, 126, 525
Selection.getFocus() и, 644
event, протокол (getURL()), 535
exec, команда (автономный Player), 528
exp() (Math, объект), 568
Expert Mode (панель Actions), 413
добавление кода в панели Actions, 30
многострочные комментарии и, 385
export, ключевое слово (потенциально зарезервированное слово), 386
extends, ключевое слово (потенциально зарезервированное слово), 386

## F

<FONT>, тег HTML, 450
FACE, атрибут тега HTML <FONT>, 450
FIFO (первым пришел – первым ушел) стеки, 286
file, протокол (getURL()), 535
final, ключевое слово (потенциально зарезервированное слово), 386
finally, ключевое слово (потенциально зарезервированное слово), 386
firstChild, свойство (XML, класс), 681, 696
FLA-файлы (Хранилище кода), 17
Flash
веб-сайты, 18
закрытие открытых соединений с приложениями на сервере, 722
открытие соединений с приложениями на сервере, 722
передача данных в формате XML приложениям на сервере, 728
программирование, введение в, 25
соединение с серверами, 716, 722
создание динамических текстовых полей инструментом Text, 439
фильмы
добавление кода в, 27
логика в, 38
программы, основанные на событиях, 236
программы, управляемые событиями, 43
создание, 37
создание интерфейса, 47
структура слоев и, 46, 343
формы (см. формы Flash), 429
экземпляры, 43
Flash 4 в сравнении с Flash 5
ActionScript, 11, 13
call(), предложение, 173
Normal Mode, обратно не совместим, 413
undefined, тип данных, 131
взаимодействие с клавиатурой, 250
зарезервированные слова, 385
локальные переменные в подпрограммах, 79
операторы конкатенации строк, 110
преобразования типов данных, 93
присвоение значений динамически именуемым переменным, 171
проблемы обратной совместимости, 743, 747
производительность извлечения подстрок, 122
синтаксис обращения к переменным, 75
эквивалентные строковые операции, 128, 152
Flash 5 Player (см. Player), 537
Flash Debugger, инструмент, 466
Flash experiments, веб-сайт (Williams и Kennedy), 732
Flash Kit (Internet.com), ресурсы, 732
Flashcoders, почтовый список рассылки, 732
float, ключевое слово (потенциально зарезервированное слово), 386
floor() (Math, объект), 104, 569
_focusrect, глобальное свойство, 527
for, цикл, 169, 192
использование оператора запятой, 160
for-in, цикл, 169, 193
встроенные объекты и, 194
действия со свойствами объектов, 307
доступ к именованным элементам массива, 282
доступ к клипам, 359

неопределенные текстовые поля и, 458
_framesloaded, свойство (MovieClip, класс), 366, 582, 590
добавление сценариев в кадры, 415
использование вместо предложения ifFrameLoaded, 175
Frame Actions (заголовок панели Actions), 411
fromCharCode() (String, класс), 108, 125, 666
fscommand(), глобальная функция, 527, 531
ftp, протокол (getURL()), 536
fullscreen, команда (автономный Player), 528
Function(), конструктор, отсутствие поддержки, 214
function, предложение, 42, 169, 171

## G

ge (больше или равно), оператор (Flash 4), 93, 128
Generator, ресурсы, 733
getAscii() (Key, объект), 550
getBeginIndex() (Selection, объект), 642
getBounds() (MovieClip, класс), 368, 583, 592
getBytesLoaded() (MovieClip, класс), 368, 583, 593
getBytesTotal() (MovieClip, класс), 368, 583, 594
getCaretIndex() (Selection, объект), 643
getCode() (Key, объект), 551
getDate() (Date, класс), 511
getDay() (Date, класс), 511
getEndIndex() (Selection, объект), 643
getFocus() (Selection, объект), 644
getFullYear() (Date, класс), 511
getHours() (Date, класс), 512
getMilliseconds() (Date, класс), 512
getMinutes() (Date, класс), 512
getMonth() (Date, класс), 512
getPan() (Sound, класс), 651
getRGB() (Color, класс), 499
getSeconds() (Date, класс), 513
getTime() (Date, класс), 513
getTimer( ), глобальная функция, 532
getTimeZoneOffset() (Date, класс), 513
getTransform()
Color, класс, 500
Sound, класс, 651
getURL()
MovieClip, класс, 368, 583, 594
глобальная функция, 533, 537
передача данных формы веб-серверу, 430
getUTCDate() (Date, класс), 514
getUTCDay() (Date, класс), 514
getUTCFullYear() (Date, класс), 514
getUTCHours() (Date, класс), 514
getUTCMilliseconds() (Date, класс), 515
getUTCMinutes() (Date, класс), 515
getUTCMonth() (Date, класс), 515
getUTCSeconds() (Date, класс), 515
getVersion(), глобальная функция, 537, 539
getVolume() (Sound, класс), 652
getYear() (Date, класс), 515
getсвойство(), глобальная функция, 531
globalToLocal() (MovieClip, класс), 368, 583, 595
GMT (Greenwich Mean Time)
Date(), глобальная функция и, 505
getTimezoneOffset() и, 514
toString() и, 522
в сравнении с UTC, 509
goto, ключевое слово (потенциально зарезервированное слово), 386
gotoAndPlay()
MovieClip, класс, 368, 583, 596
глобальная функция, 539
не работает в событиях клипов, 540
gotoAndStop()
MovieClip, класс, 368, 583, 597
глобальная функция, 541
добавление меток адров, 52
не работает в событиях клипов, 542
сценарии для кнопок, 52
gt (больше), оператор (Flash 4), 93, 128

## H

hasChildNodes() (XML, класс), 682, 697
_height, свойство (MovieClip, класс), 366, 582, 597
hide() (Mouse, объект), 579, 580
_highquality, глобальное свойство, 542

hitTest() (MovieClip, класс), 259, 368, 583, 598, 601
Home, клавиша, 250
HOME, свойство (Key, объект), 549
HTML, поддержка, 449
  ввод текста в текстовые поля с включенным HTML, 456
  встроенные шрифты в текстовых полях, 455
  вывод на экран текста в формате HTML, 454
  гипертекстовые ссылки в текстовых полях, 452
    вызов функций ActionScript из, 457
    выполнение JavaScript из, 457
    не добавляются в обход по Tab, 453
  заключение атрибутов в кавычки, 453
  курсив в текстовых полях, 450
  перенос строк в текстовых полях, 452
  подчеркнутый текст в текстовых полях, 450
  полужирный текст в текстовых полях, 450
  разбиение на параграфы в текстовых полях, 451
  теги якорей в текстовых полях, 452
    вызов функций ActionScript из, 457
    не добавляются в обход по Tab, 453
  управление шрифтами в текстовых полях, 450
HTML, создание форм в, 432
http, протокол (getURL()), 536

## I

<I>, тег HTML, 450
If Frame Is Loaded, действие, 175
if
  ключевое слово в сценарии расчета, 56
  предложение, 178
if-else if-else, предложение, 169
  эмуляция switch-предложений, 183
if-else, предложение, 181
ifFrameLoaded, предложение, 169
  экземпляры, 175
ignoreWhite, свойство (XML, класс), 681, 698
implements, ключевое слово (потенциально зарезервированное слово), 386
Import From File, операция, 419
import, ключевое слово (потенциально зарезервированное слово), 386
#include, директива, 419, 543
  запрещена точка с запятой, 383
indexOf() (String, класс), 116, 666
  substring() и, 122
–Infinity, глобальное свойство, 103, 545
  (см. также NEGATIVE_INFINITY, свойство), 545
  isFinite() и, 547
Infinity, глобальное свойство, 103, 545
  (см. также POSITIVE_INFINITY, свойство), 545
  isFinite() и, 547
Insert Target Path, инструмент, 356
Insert, клавиша, 250
INSERT, свойство (Key, объект), 548
insertBefore() (XML, класс), 682, 698
instanceof, ключевое слово (потенциально зарезервированное слово), 386
int(), глобальная функция, 546
  Math.floor() и, 569
  parseInt() и, 631, 633
int, ключевое (потенциально зарезервированное) слово, 386
interface, ключевое (потенциально зарезервированное) слово, 386
isDown() (Key, объект), 552
isFinite(), глобальная функция, 547
isNaN(), глобальная функция, 102, 547
ISO-8859, наборы символов, 105
ISO-8859-1 (Latin 1), набор символов, 105
isToggled() (Key, объект), 553

## J

Java и эквивалентные свойства ActionScript, 326
JavaScript
  switch, оператор в, 183
  как работает eval(), 126

модель для ActionScript, 12
область видимости статических функций, 406
различия между ActionScript и ECMA-262, 748
связь из ActionScript, 453, 457
fscommand(), 527, 531
getURL(), 533, 537
чувствительность к регистру и ActionScript, 388
javascript, протокол (getURL()), 536
join() (Array, класс), 296, 484


## K


Key, объект, 548
keyDown, событие клипа и, 260
в сравнении с событием кнопки keyPress, 250
коды специальных клавиш, 741
свойства-коды клавиш, 548, 549
Key.getAscii(), 260, 550
Key.getCode(), 260, 551
Key.isDown(), 260, 552
перехват комбинаций клавиш, 263
Key.isToggled(), 260, 553
keyDown, событие клипа, 259
обработчик, 260, 585
updateAfterEvent() и, 267
keyPress, событие кнопки t, 249
в сравнении с объектом Key, 250
keyUp, событие клипа, 263
обработчик, 585
updateAfterEvent() и, 268


## L


lastChild, свойство (XML, класс), 682, 699
lastIndexOf() (String, класс), 117, 668
Latin 1, набор символов, 105
кодовые позиции, 108, 126, 665, 736, 740
коды клавиш и кодовые позиции, 550
создание строк по кодовым позициям, 125, 666
сортировка массивов, 293
шестнадцатеричные escape-последовательности и, 525
le (меньше или равно), оператор (Flash 4), 93, 128
Left, клавиша, 250
LEFT, свойство (Key, объект), 548
length() (функция Flash 4), 115
length, свойство
arguments, объект, 481
Array, класс, 280, 485
расширение массивов без присвоения значений элементам, 284
удаление элементов массивов, 289
String, класс, 115, 668
_leveln, глобальное свойство, 74, 554
ссылки на документы в Player, 355
стеки документов, 346, 355
LIFO (первым пришел – последним ушел) стеки, 286
Line Display, опции текстовых полей, 443
Lingo, связь с, 528
lingo, протокол (getURL()), 536
List Objects, команда (инструмент отладки), 463
List Variables, команда (инструмент отладки), 463
ll, свойство (transformObject), 654
LN10, свойство (Math, объект), 569
LN2, свойство (Math, объект), 570
load() (XML, класс), 682, 699
load, событие клипа, 253
обработчик, 253, 584
порядок выполнения, 265
loaded, свойство (XML, класс), 682, 700
XML.status, свойство и, 712
loadMovie()
MovieClip класс
load/unload, событие клипа, 255
использование обработчиков события data, 257
MovieClip, класс, 368, 583, 601
глобальная функция, 555
target, параметр и, 371
загрузка в плеер внешних файлов SWF, 340
использование с attachMovie(), 342
loadMovieNum(), глобальная функция, 557
loadVariables()
MovieClip класс
использование обработчиков события data, 256

MovieClip, класс, 368, 583, 601
глобальная функция, 558, 560
target, параметр и, 371
ограничения, связанные с безопасностью, 559
пересылка данных формы приложению на сервере, 433
loadVariablesNum(), глобальная функция, 560
localToGlobal() (MovieClip, класс), 368, 584, 602
log() (Math, объект), 570
LOG10E, свойство (Math, объект), 570
LOG2E, свойство (Math, объект), 571
long, ключевое слово (потенциально зарезервированное слово), 386
lr, свойство (transformObject), 654
lt (меньше), оператор (Flash 4), 93, 128

## M

Macromedia Generator, ресурсы, 733
Math, объект, 103
встроенные математические функции, 104, 561
математические константы, 103, 561
Math.abs(), 563
Math.acos(), 563
Math.asin(), 564
Math.atan(), 565
Math.atan2(), 565, 566
Math.ceil(), 566
Math.cos(), 567
Math.E, свойство, 568
Math.exp(), 568
Math.floor(), 104, 569
Math.LN10, свойство, 569
Math.LN2, свойство, 570
Math.log(), 570
Math.LOG10E, свойство, 570
Math.LOG2E, свойство, 571
Math.max(), 571
Math.min(), 572
Math.PI, свойство, 572
Math.pow(), 100, 572
Math.random(), 104, 573
Math.round(), 575
Math.sin(), 575
Math.sqrt(), 576
Math.SQRT1_2, свойство, 577
Math.SQRT2, свойство, 577
Math.tan(), 577
Max Characters, опция текстовых полей, 445
max() (Math, объект), 571
MAX_VALUE, свойство (Number класс), 102
MAX_VALUE, свойство (Number, класс), 624
maxscroll, свойство (текстовых полей), 448, 578
scroll, свойство и, 640
mbchr() (Flash 4), 128
mblength() (Flash 4), 128
mbord() (Flash 4), 128
mbsubstring() (Flash 4), 128
McConnell, Steve, 381, 472
MIME, тип содержимого
XML.send() и, 710
задание, 693
установка, 431
min() (Math, объект), 572
MIN_VALUE, свойство (Number, класс), 102, 625
Ming (библиотека C с открытым кодомlibrary), 735
Mouse, объект, 578
Mouse.hide(), 579, 580
Mouse.show(), 580
mouseDown, событие клипа, 259
обработчик, 584
updateAfterEvent() и, 267
mouseMove, событие клипа, 259
обработчик, 585
updateAfterEvent() и, 267
mouseUp, событие клипа, 259
обработчик, 584
updateAfterEvent() и, 267
Movie Explorer, 418
MovieClip, «класс», 581, 586
movieclip, тип данных, 84, 331
в сравнении с типом данных object, 407
MovieClip._alpha, свойство, 366, 582, 586
MovieClip._currentframe, свойство, 366, 582, 587
MovieClip._droptarget, свойство, 366, 582, 588
MovieClip._framesloaded, свойство, 366, 582, 590

добавление сценариев в кадры, 415
использование вместо предложения ifFrameLoaded, 175
MovieClip._height, свойство, 366, 582, 597
MovieClip._name, свойство, 366, 582, 603
определение имен клипов, 360
хранение имен экземпляров, 339
MovieClip._parent, свойство, 366, 582, 604
доступ к переменным в различных временных диаграммах, 72
недоступно на главной временной диаграмме, 353
ссылка на текущий клип, 332, 352
MovieClip._rotation, свойство, 366, 582, 606
MovieClip._target, свойство, 360, 366, 582, 610
MovieClip._totalframes, свойство, 366, 582, 610
использование вместо предложения ifFrameLoaded, 175
MovieClip._url, свойство, 366, 582, 611
MovieClip._visible, свойство, 366, 582, 613
MovieClip._width, свойство, 366, 582, 613
MovieClip._x, свойство, 366, 582, 614
MovieClip._xmouse, свойство, 366, 582, 616
MovieClip._xscale, свойство, 366, 582, 617
MovieClip._y, свойство, 366, 582, 618
MovieClip._ymouse, свойство, 366, 582, 619
MovieClip._yscale, свойство, 366, 582, 619
MovieClip.attachMovie(), 368, 583, 587
добавление созданных клипов в стек, 344
создание экземпляров, 338
MovieClip.duplicateMovieClip(), 368, 583, 589
добавление созданных клипов в стек, 345
порождающие клипы, 334
создание экземпляров, 336
MovieClip.getBounds(), 368, 583, 592
MovieClip.getBytesLoaded(), 368, 583, 593
MovieClip.getBytesTotal(), 368, 583, 594
MovieClip.getURL(), 368, 583, 594
MovieClip.globalToLocal(), 368, 583, 595
MovieClip.gotoAndPlay(), 368, 583, 596
MovieClip.gotoAndStop(), 368, 583, 597
MovieClip.hitTest(), 368, 583, 598, 601
MovieClip.loadMovie(), 368, 583, 601
MovieClip.loadVariables(), 368, 583, 601
MovieClip.localToGlobal(), 368, 584, 602
MovieClip.nextFrame(), 369, 584, 604
MovieClip.play(), 369, 584, 605
MovieClip.prevFrame(), 369, 584, 605
MovieClip.removeMovieClip(), 369, 584, 606
MovieClip.startDrag(), 369, 584, 607
MovieClip.stop(), 369, 584, 608
MovieClip.stopDrag(), 369, 584, 609
MovieClip.swapDepths(), 369, 584, 609
MovieClip.unloadMovie(), 369, 584, 611
MovieClip.valueOf(), 369, 584, 612
Multiline, опция текстовых полей, 444
Myers, Gene, 404

## N

_name, свойство (MovieClip, класс), 366, 582, 603
определение имен клипов, 360
NaN (не число)
isNan() и, 547
Number, класс, 625
глобальное свойство, 87, 102, 620
native, ключевое слово (потенциально зарезервированное слово), 386
ne (не равно), оператор (Flash 4), 93, 128
NEGATIVE_INFINITY, свойство (Number, класс), 625
(см. также – Infinity, глобальное свойство), 625
Netscape
руководство по ООП в JavaScript, 734
руководство по основам языка JavaScript, 734

new, оператор, 161
  создание экземпляров объектов, 305, 313
newline, константа, 621
nextFrame()
  MovieClip, класс, 369, 584, 604
  глобальная функция, 621
nextScene(), глобальная функция, 621
nextSibling, свойство (XML, класс), 682, 701
nodeName, свойство (XML, класс), 682, 702
nodeType, свойство (XML, класс), 682, 703
nodeValue, свойство (XML, класс), 682, 704
Normal Mode (Actions, окно), 412
  отсутствие обратной совместимости, 413
  переключение из Expert Mode, 414
not (логическое отрицание), оператор (Flash 4), 386
NOTATION_NODE (тип узла XML), 703
null, тип данных, 83, 132
  правила равенства для, 145
Num Lock, клавиша и isToggled(), 553
Number(), глобальная функция, 91, 622
Number, класс, 623
  максимальные/минимальные значения, 102
number, тип данных, 83, 98, 104
  Infinity/– Infinity, глобальные свойства, 103, 545
  NaN (не число), 102
  использование операторов, 103
  литералы чисел с плавающей точкой, 100
  максимальное/минимальное значения, 624
  правила равенства для, 144
  преобразование строк в, 99
  преобразование типов данных в, 88, 139
  целочисленные литералы, 99
Number.MAX_VALUE, свойство, 102, 624
Number.MIN_VALUE, свойство, 625
Number.NaN, свойство, 625
Number.NEGATIVE_INFINITY, свойство, 625
  (см. также – Infinity, глобальное свойство), 625
Number.POSITIVE_INFINITY, свойство, 626
  (см. также Infinity, глобальное свойство), 626
Number.toString(), 626

## O

Object Actions (заголовок окна Actions), 411
Object(), конструктор, 305
Object, класс, 627
  эмуляция глобальных переменных, 71, 326
object, тип данных, 84
  в сравнении с типом данных movieclip, 407
Object.__proto__, свойство, 320, 629
Object.constructor, свойство, 320, 628
Object.toString(), 629
Object.valueOf(), 630
on (обработчик события кнопки)
  определение, 238
  события кнопок для, 246
onClipEvent, событие клипа
  обработчик
    определение, 238
    события клипов для, 251, 584, 585
onClose, обработчик события (XMLSocket, класс), 724
onConnect, обработчик события (XMLSocket, класс), 724
onData(), обработчик события
  XML, класс, 682, 705
  XMLSocket, класс, 726
onLoad(), обработчик события, 682, 706
onXML, обработчик события (XMLSocket, класс), 726
OpenSWF, информационный центр, 735
or (логическое ИЛИ), оператор (Flash 4), 386
ord() (Flash 4), 128
Output, окно, 30
  trace() и, 32, 40
  сообщения об ошибках, посылаемые в, 461

## P

<P>, тег HTML, 451
package, ключевое слово (потенциально зарезервированное слово), 386
Page Down (см. PGDN), 548
Page Up (см. PGUP), 548
Parameters, окно (Actions, окно), 413
_parent, свойство (MovieClip, класс), 366, 582, 604
  доступ к переменным в различных временных диаграммах, 72
  недоступно на главной временной диаграмме, 353
  ссылка на текущий клип, 332, 352
parentNode, свойство (XML, класс), 682, 707
parseFloat(), глобальная функция, 92, 631
parseInt(), глобальная функция, 92, 631, 633
parseXML() (XML, класс), 682, 708
Password, опция текстовых полей, 444
Perl
  библиотека для создания фильмов Flash/SWF, 735
  интеграция с Flash, 432, 434
PgDn, клавиша, 250
PGDN, свойство (Key, объект), 548
PgUp, клавиша, 250
PGUP, свойство (Key, объект), 548
PI, свойство (Math, объект), 572
play()
  MovieClip, класс, 369, 584, 605
  глобальная функция, 633
Player
  XML.send() и, 710
  автономный, 527, 531
    отключение команд меню, 261
  версия, 681
  загрузка внешних файлов SWF, 340, 555, 601
  качество вывода
    _highquality, глобальное свойство, 542
    _quality, глобальное свойство, 638
    toggleHighQuality, глобальная функция, 677
  обновление для (Build 41/42), 746
  опредление версий, 537, 539
  получение количества загруженных байт, 593
  стек уровней документов, 74, 346, 355, 554
  удаление
    клипов фильмов из, 606, 611, 639, 679
    прикрепленных и дублированных экземпляров, 364
  управление звуком, 648, 661
pop() (Array, класс), 290, 486
POSITIVE_INFINITY, свойство (Number, класс), 626
  (см. также Infinity, глобальное свойство), 626
pow() (Math, объект), 100, 572
press, событие кнопки, 247
prevFrame()
  MovieClip, класс, 369, 584, 605
  глобальная функция, 634
previousSibling, свойство (XML, класс), 682, 708
prevScene(), глобальная функция, 634
print(), глобальная функция, 634
  target, параметр и, 371
print, протокол (getURL()), 536
printAsBitmap(), глобальная функция, 636
  target, параметр и, 371
printAsBitmapNum(), глобальная функция, 637
printNum(), глобальная функция, 637
private, ключевое слово (потенциально зарезервированное слово), 386
PROCESSING_INSTRUCTION_NODE (тип узла XML), 703
protected, ключевое слово (потенциально зарезервированное слово), 386
prototype, свойство, создание наследуемых свойств с помощью, 317
public, ключевое слово (потенциально зарезервированное слово), 386
push() (Array, класс), 284, 486

## Q

_quality, глобальное свойство, 638
quit, команда (автономный Player), 528

## R

random()
Math, объект, 104, 573
глобальная функция, 639
Red, Green, Blue (см. RGB-тройка), 495
Red, Green, Blue и Alpha (см. RGBA-четверка), 495
release, событие, 248
releaseOutside button event, 248
removeMovieClip()
MovieClip, класс, 369, 584, 606
глобальная функция, 639
target, параметр и, 371
удаление прикрепленных и дублированных экземпляров, 364
removeNode() (XML, класс), 682, 709
Reset, кнопка на формах, 437
Return, клавиша, перехват, 263
return, предложение, 169, 173
возврат значений из функций, 213
завершение функций, 212
reverse() (Array, класс), 293, 487
RGBA-четверка, 495, 499
getTransform(), 500
setTransform(), 503
RGB-тройка, 495, 499
getRGB(), 499
setRGB(), 502
Right, клавиша, 250
RIGHT, свойство (Key, объект), 548
rl, свойство (transformObject), 654
rollOut, обработчик, 249
rollOver, обработчик, 248
_root, глобальное свойство, 640
доступ к переменным в различных временных диаграммах, 72
ссылка на текущий уровень главного фильма, 354
_rotation, свойство (MovieClip, класс), 366, 582, 606
round() (Math, объект), 575
rr, свойство (transformObject), 655

## S

Script, окно (Actions, панель), 411
добавление кода в кнопки, 415
добавление предложений в, 412
Script, окно (Actions, панель), 28
Scroll Lock, клавиша и isToggled(), 553
scroll, свойство (текстовых полей), 447, 640
maxscroll, свойство и, 578
Selectable, опция текстовых полей, 445
Selection, объект, 641
выделение текста в текстовых полях, 458
Selection.getBeginIndex(), 642
Selection.getCaretIndex(), 643
Selection.getEndIndex(), 643
Selection.getFocus(), 644
Selection.setFocus(), 644, 647
Selection.setSelection(), 647
send()
XML, класс, 682, 709
передача данных формы веб-серверу, 430
XMLSocket, класс, 728
sendAndLoad() (XML, класс), 682, 711
XML.loaded, свойство и, 700
передача данных формы веб-серверу, 430
Set Variable (Flash 4), 171
set, предложение, 169
в сравнении с Set Variable в Flash 4, 171
setDate() (Date, класс), 516
setFocus() (Selection, объект), 644, 647
setFullYear() (Date, класс), 516
setHours() (Date, класс), 517
setInterval()
обрабатывающие клипы в качестве альтернативы, 334
эмуляция, 198, 750
setMilliseconds() (Date, класс), 517
setMinutes() (Date, класс), 517
setMonth() (Date, класс), 518
setPan() (Sound, класс), 652
setProperty(), глобальная функция, 647
setRGB() (Color, класс), 502
setSeconds() (Date, класс), 518
setSelection() (Selection, объект), 647
setTime() (Date, класс), 518
setTimeout()
обрабатывающие клипы в качестве альтернативы, 334
эмуляция, 198, 750
setTransform()
Color, класс, 503
Sound, класс, 654

setUTCDate() (Date, класс), 519
setUTCFullYear() (Date, класс), 520
setUTCHours() (Date, класс), 520
setUTCMilliseconds() (Date, класс), 520
setUTCMinutes() (Date, класс), 520
setUTCMonth() (Date, класс), 521
setUTCSeconds() (Date, класс), 521
setVolume() (Sound, класс), 655
setYear() (Date, класс), 521
shift() (Array, класс), 291, 487
SHIFT, свойство (Key, объект), 548
Shift-JIS, символы (японские), 105
  шестнадцатеричные escape-последовательности и, 525
short, ключевое слово (потенциально зарезервированное слово), 386
show() (Mouse, объект), 580
showmenu, команда (автономный Player), 528
sin() (Math, объект), 575
Single Line, опция текстовых полей, 443
SIZE, атрибут тега HTML <FONT>, 450
slice()
  Array, класс, 146, 295, 487
  String, класс, 120, 669
Smart Clips, 333, 421
  xch, (обмен) экземпляров и, 425
  добавление пользовательских интерфейсов в, 427
  использование в фильмах, 427
  параметры клипов
    конфигурирование, 423
    удаление/переупорядочение, 425
  построение пользовательских интерфейсов для, 425
  построение стандартных интерфейсов для, 423
sort() (Array, класс), 293 488
_soundbuftime, глобальное свойство, 659
Sound-объекты
  прикрепленные звуки, 649
    воспроизведение, 656
    выключение, 657
  создание, 649
Sound, класс, 648
Sound.attachSound(), 649
Sound.getPan(), 651
Sound.getTransform(), 651
Sound.getVolume(), 652
Sound.setPan(), 652
Sound.setTransform(), 654
Sound.setVolume(), 655
Sound.start(), 656
Sound.stop(), 657
Space, клавиша, 250
SPACE, свойство (Key, объект), 548
splice() (Array, класс), 287, 292, 489
split() (String, класс), 121, 670
sprite, событие, обработчик, 536
sqrt() (Math, объект), 576
SQRT1_2, свойство (Math, объект), 577
SQRT2, свойство (Math, объект), 577
start() (Sound, класс), 656
startDrag()
  MovieClip, класс, 369, 584, 607
  глобальная функция, 659
    target, параметр и, 371
static, ключевое слово (потенциально зарезервированное слово), 386
status, коды анализа XML, 712
status, свойство (XML, класс), 682, 712
stop()
  MovieClip, класс, 369, 584, 608
  Sound, класс, 657
  глобальная функция, 49, 660
stopAllSounds(), глобальная функция, 661
stopDrag()
  MovieClip, класс, 369, 584, 609
  глобальная функция, 661
String(), глобальная функция, 90, 662
String, класс, 662
string, тип данных, 83, 104
  правила равенства для, 145
  преобразование
    значений в, 662
    типов данных в, 88
String.charAt(), 116, 664
String.charCodeAt(), 126, 665
String.concat(), 665
  в сравнении с оператором +=, 110
String.fromCharCode(), 108, 125, 666
String.indexOf(), 116, 666
  substring() и, 122
String.lastIndexOf(), 117, 668
String.length, свойство, 115, 668
String.slice(), 120, 669
String.split(), 121, 670
String.substr(), 119, 672

String.substring(), 118, 673
  indexOf() и, 122
String.toLowerCase(), 124, 674
String.toUpperCase(), 123, 675
Submit, кнопка в формах Flash, 432
substr() (String, класс), 119, 672
substring() (String, класс), 118, 673
  indexOf() и, 122
super, ключевое слово (потенциально зарезервированное слово), 386
swapDepths() (MovieClip, класс), 369, 584, 609
  назначение уровней экземплярам в программно генерируемых стеках клипов, 346
  обмен экземпляров в стеках внутренних слоев, 343
SWF-файлы
  _leveln, глобальное свойство, 346, 355, 554
  _url, свойство и, 611
  внешнее размещение кода ActionScript, 419
  добавление файлов SWF пользовательских интерфейсов в Smart Clips, 427
  загрузка в Player, 340, 555, 601
  загрузка на уровни документов
    посредством loadMovie(), 346, 355
    посредством loadMovieNum(), 557
  обработчики события data и, 257
  событие клипа unload и, 255
  создание файлов SWF пользовательских интерфейсов, 426
SWF, формат файла
  обратная совместимость, 743
  ресурсы для, 734
switch
  ключевое слово (потенциально зарезервированное слово), 386
  предложение, эмуляция, 183
synchronized, ключевое слово (потенциально зарезервированное слово), 386

## T

Tab, клавиша, нажатие, перехват, 250, 262
TAB, свойство (Key, объект), 549
tan() (Math, объект), 577
_target, свойство (MovieClip, класс), 360, 366, 582, 610
target, параметр и глобальные функции, 370
targetPath(), глобальная функция, 361, 675
Tell Target, инструмент, 361, 676
tellTarget(), глобальная функция, 361, 676
Test Movie, режим, 461
Text Options, панель, 443
TEXT_NODE (тип узла XML), 703
this, ключевое слово
  настройка свойств объекта во время его создания, 313
  передача в функции анонимных клипов как ссылок, 244
  явная ссылка на текущий экземпляр, 350
  явная ссылка на объекты с помощью, 244
throws, ключевое слово (потенциально зарезервированное слово), 386
toggleHighQuality(), глобальная функция, 677
toLowerCase() (String, класс), 124, 674
Toolbox, окно (Actions, окно), 411
Toolbox, список (Actions, панель), 28
toString(), 90
  Array, класс, 297, 490
  Boolean, класс, 493
  Date, класс, 522
  Number, класс, 626
  Object, класс, 629
  XML, класс, 682, 713
_totalframes, свойство (MovieClip, класс), 366, 582, 610
  использование вместо предложения ifFrameLoaded, 175
toUpperCase() (String, класс), 123, 675
trace(), глобальная функция, 677
  интерпретатор ActionScript и, 32
  использование в циклах, 39, 187
  как инструмент отладки, 462
transformObject, свойства, 654, 655

transient, ключевое слово (потенциально зарезервированное слово), 386
trapallkeys, команда (автономный Player), 261, 528
try, ключевое слово (потенциально зарезервированное слово), 386
typeof, оператор, 35, 67, 161
возвращаемые значения, 94
разрешение преобразований неожиданных типов, 470

## U

<U>, тег HTML, 450
Ultrashock.com, 732
undefined, тип данных, 83, 131
правила равенства для, 145
unescape(), глобальная функция, 678
Unicode
escape-последовательности, 105, 108, 736, 740
не поддерживается в ActionScript, 105
unload, событие клипа, 254
обработчик, 254, 584
порядок выполнения, 265
unloadMovie(), 363
MovieClip, класс, 369, 584, 611
глобальная функция, 679
target, параметр и, 371
удаление содержимого экземпляров клипов, 363
удаление уровней в стеках документов, 362
unloadMovieNum(), глобальная функция, 679
unshift() (Array, класс), 285, 491
Up, клавиша, 250
UP, свойство (Key, объект), 549
updateAfterEvent(), глобальная функция, 680
обновление экрана с помощью, 267
эмуляция пользовательского курсора мыши, 271
_url, свойство (MovieClip, класс), 366, 582, 611
(см. также getURL()), 366
UTC (Coordinated Universal Time – универсальное время), 505, 511
getTimezoneOffset() и, 514
getUTCDate() и, 514
getUTCDay() и, 514
getUTCFullYear() и, 514
getUTCHours() и, 514
getUTCMilliseconds() и, 515
getUTCMinutes() и, 515
getUTCMonth() и, 515
getUTCSeconds() и, 515
UTC() (Date, класс), 522

## V

valueOf()
Boolean, класс, 493
Date, класс, 523
MovieClip, класс, 369, 584, 612
Object, класс, 630
var
ключевое слово, 32
локальные переменные и, 78
предложение, 169
внутри и вне функций, 170
Variable, опция текстовых полей, 444
vbscript, протокол (getURL()), 536
$version, «глобальное свойство», 681
_visible, свойство (MovieClip, класс), 366, 582, 613
void, оператор, 160
volatile, ключевое слово (потенциально зарезервированное слово), 386

## W

while, цикл, 40, 169, 186
Windows, виртуальные коды клавиш, 549
_width, свойство (MovieClip, класс), 366, 582, 613
with, предложение, 169, 174
Word Wrap, опция текстовых полей, 445

## X

_x, свойство (MovieClip, класс), 366, 582, 614
xch, (обмен) экземпляры, 425
XML-документы
DOCTYPE, тег, 696
импорт внешних, 700
обход иерархии объектов, 701
построение, 685

преобразование в исходный код XML, 713
создание
текстовых узлов, 695
узлов элементов, 694
тег объявления XML, 714
удаление пробельных символов из, 698
удаление узлов из, 709
XML, иерархия объектов (см. иерархия объектов), 709
XML, исходный код
импорт из внешних документов, 699
отправка сценарию, 709
и получение в ответ исходного кода XML, 711
преобразование объектов узлов XML в, 713
синтаксический анализ, 708
возможные ошибки, 712
XML, класс, 681, 688
иерархия объектов, 682, 686
внутренний надкласс для, 714
методы, 682
область видимости обработчиков событий, 246
обработчики событий, 682
прикрепление обработчиков событий к, 241, 269
свойства, 681
XML.appendChild(), 682, 689
XML.attributes, свойство, 681, 690
XML.childNodes, свойство, 681, 691
XML.cloneNode(), 682, 692
XML.contentType, свойство, 681, 693
XML.createElement(), 682, 694
XML.createTextNode(), 682, 695
XML.docTypeDecl, свойство, 681, 695
XML.firstChild, свойство, 681, 696
XML.hasChildNodes(), 682, 697
XML.ignoreWhite, свойство, 681, 698
XML.insertBefore(), 682, 698
XML.lastChild, свойство, 682, 699
XML.load(), 682, 699
XML.loaded, свойство, 682, 700
XML.status, свойство и, 712
XML.nextSibling, свойство, 682, 701
XML.nodeName, свойство, 682, 702
XML.nodeType, свойство, 682, 703
XML.nodeValue, свойство, 682, 704
XML.onData(), обработчик события, 682, 705
XML.onLoad(), обработчик события, 682, 706
XML.parentNode, свойство, 682, 707
XML.parseXML(), 682, 708
XML.previousSibling, свойство, 682, 708
XML.removeNode(), 682, 709
XML.send(), 682, 709
передача данных формы веб-серверу, 430
XML.sendAndLoad(), 682, 711
XML.loaded, свойство и, 700
передача данных формы веб-серверу, 430
XML.status, свойство, 682, 712
XML.toString(), 682, 713
XML.xmlDecl, свойство, 682, 714
xmlDecl, свойство (XML, класс), 682, 714
XMLnode, класс, 714
XMLSocket, класс, 716, 722
клиент чата, создание, 718, 722
область видимости обработчиков событий, 246
прикрепление обработчиков событий к, 241, 269
XMLSocket.close(), 722
XMLSocket.connect(), 722
XMLSocket.onClose, обработчик события, 724
XMLSocket.onConnect, обработчик события, 724
XMLSocket.onData, обработчик события, 726
XMLSocket.onXML, обработчик события, 726
XMLSocket.send(), 728
_xmouse, свойство (MovieClip, класс), 366, 582, 616
_xscale, свойство (MovieClip, класс), 366, 582, 617


## Y


_y, свойство (MovieClip, класс), 366, 582, 618
_ymouse, свойство (MovieClip, класс), 366, 582, 619
_yscale, свойство (MovieClip, класс), 366, 582, 619

## А

абсолютные
пути к клипам, 361, 612, 675
ссылки, 72, 354, 604, 640
автоматическая типизация, 66, 85
автоматическое
преобразование
значений переменных, 66
типа данных, 87
создание переменных, 61
автономный
Player
trapallkeys, команда, 261
fscommand() и, 527, 531
отключение команд меню, 261
Projector, выполнение в полноэкранном режиме, 529
автопилот ActionScript, отключение, 30
активная область кнопок, 247
mouseDown не привязано к, 259
алфавитное сравнение, 112
анимация, синхронизированная, 203
аргументы, 34
в сравнении с параметрами, 210
использование функций в качестве, 228
передача нескольких в функцию, 211
элементарных типов в сравнении с составными, 223
арифметические
операторы, 139
операции в ActionScript, 34
арккосинус, вычисление, 563
арксинус, вычисление, 564
арктангенс, вычисление, 565
асинхронное выполнение кода, 43, 236
ассоциативность операторов, 135
ассоциативные массивы, эмуляция, 282
атрибуты, заключение значений в кавычки, 453

## Б

базовый класс ActionScript, 627
баланс левого/правого каналов
определение, 651
установка, 652
байты
получение числа загруженных в Player, 593
размер клипов или фильмов, 594
беззнаковый сдвиг вправо с присвоением (>>>=), оператор, 136
безопасность, проблемы
loadVariables() и, 559
XMLSocket.connect() и, 723
бесконечная рекурсия, предотвращение, 225
бесконечные циклы, 187, 191
способы избегания, 197
библиотека
создание новых экземпляров из, 338, 587
создание экземпляров клипов вручную, 336
библиотеки совместного доступа и импорт этапа исполнения, 420
бинарные операторы, 134
биты (двоичные цифры), 392
блоки предложений, 167
создание функций, 207
условные предложения и, 185
больше (>), оператор (Flash 5), 112, 149
больше или равно (>=), оператор (Flash 5), 112, 150
броузеры, веб (см. веб-броузеры), 530
буксировка клипов, 607, 659
завершение операции буксировки, 609, 661
булевы
значения
оператора сравнения и, 112
операторы равенства/неравенства и, 111
логические операторы (см. логические операторы), 154

## В

ввод данных пользователем, события клипов, связанные с
updateAfterEvent() и, 680
веб-броузеры
открытие окон из Flash, 533, 537, 594
полноэкранный режим в, 530
веб-сайты
использующие ActionScript, 18

векторная графика
печать кадров клипов, 634
печать кадров уровней документов, 637
версии, определение, 537, 539
ветвление программы и логические операторы, 153
вложенные
клипы, доступ к, 747
массивы, 278
concat() и, 288
разделитель – запятая и, 297
создание, 298
циклы, 190
экземпляры клипов, ссылка на, 351
внешние
файлы ActionScript, импорт текста из, 419, 543
фильмы, импортирование, 340
внутренние функции, 207, 227
(см. также, встроенные функции), 228
доступность, 228
возврат каретки в строках (см. перенос строки), 106
вопросник с вариантами ответов
использование массивов для хранения ответов, 299
создание, 45
централизация кода, 230
воспроизведение фильма, события клипа, 252
временные диаграммы
воспроизведение с заданного кадра, 539, 596
добавление меток кадров, 51
доступ к переменным различных, 71
количество кадров в, 610
остановка в заданном кадре, 541, 597
переменные, 70
поиск скрытых слоев, 417
создание переменных, 49, 68
ссылки, абсолютные/относительные, 72, 604, 640
структура слоев и, 46
структуры слоев и, 343
временные переменные, 217
временные преобразования, 93
время
определение истекшего, 532
работа с, 505, 511
встроенные
классы, 328, 479
область видимости обработчика событий, 246
прикрепление обработчиков событий к, 241, 269
клипы
методы, 368
свойства, 365
объекты, 329, 479
for-in циклы и, 194
функции, 42, 477
математические, 104, 561
шрифты в текстовых полях HTML, 455
вызов функций, 114, 163, 170, 207, 307
выключение звука (см. звук, выключение), 655
выражения, 37, 166
литеральные, 37, 84
операторы и, 133
регулярные (не поддерживаются), 118, 673
сложные, 84
автоматическая типизация, 86
создание данных с помощью, 84
высота клипов/фильмов
в пикселах, 597
в процентном отношении, 619
вычитание (– ), оператор, 35, 140

## Г

гипертекстовые ссылки (поддержка HTML в текстовых полях), 452
вызов функций ActionScript из, 457
выполнение JavaScript из, 457
не добавляются в обход по Tab, 453
главные фильмы, 333
ссылка на, 348
ссылка на документы через _levelN, 355
ссылка на текущий уровень через _root, 354
удаление, 362
явные и неявные ссылки, 349
глобальные
переменные
отсутствие поддержки, 70

эмуляция, 71, 326
свойства, 478
функции, 42, 477
target, необходимый параметр, 370
доступность, 228
перекрытие методов и глобальных функций, 369
годы
извлечение, 511
время UTC, 514
относительно 1900, 515
назначение, 516
время UTC, 520
относительно 1900, 521
головка воспроизведения
воспроизведение временных диаграмм из заданного кадра, 539, 596
использование циклов временной диаграммы, 198
остановка в заданном кадре, 541, 597
перемещение на один кадр вперед, 604, 621
перемещение на один кадр назад, 605, 634
громкость
получение текущей установки, 652
установка, 655
Гроссман, Гари (Grossman, Gary), 11, 13, 407
группирования оператор, 159

## Д

дата
вычисление через миллисекунды, 508, 513, 518, 523
время UTC, 522
загрузка файлов SWF в зависимости от, 511
прибавление/вычитание дней, 510
применение пользовательского формата, 510
двоичные числа и поразрядные операторы, 392
двойная точность, чисел с плавающей точкой, 101
двойные кавычки в строках, 106
двумерные масссивы, 298
декрементирование (– –), оператор, 140
деление
(/), оператор, 103, 141
по модулю (%), оператор, 143
по модулю с присвоением (%=), оператор, 136
с присвоением (/=), оператор, 136
день недели
получение, 511
время UTC, 514
десятичная точка в литералах чисел с плавающей точкой, 100
десятичный (по основанию 10)
логарифм e, 570
целые числа
преобразование чисел в, 631, 633
действительные числа, 99
действия
в сравнении с предложениями, 176
в сравнении с функциями, 227
динамическая установка типа данных для переменных, 66
динамически именуемые переменные, 62
динамические функции, 214
обработчики событий клипов и, 269
дочерние узлы
доступ, 691
обращение к
первому потомку, 696
последнему потомку, 699
проверка наличия потомков, 697

## Ж

жесткий перенос
добавление в текст (см. перенос строки), 106
обработка в текстовых полях, 443

## З

заголовки циклов, 191
закодированные строки, создание, 525
заполняемые формы (см. формы Flash), 432
запятая (,), оператор, 159
в циклах for, 193
зарезервированные слова, 385
зациклившиеся сценарии, проблемы прерывания, 198

звездное скопление, генерация с помощью события load, 254
звук
получение текущей настройки, 652
выключение, 661
Sound.setVolume() и, 655
Sound.stop() и, 657
stopAllSounds() и, 661
громкость
установка, 655
задание предварительной загрузки потокового, 659
левый/правый канал
определение баланса, 651
распределение по левому/правому динамикам, 651, 654
установка баланса, 652
панорамирование, 651, 654
прикрепленные звуки, 649
воспроизведение, 656
выключение, 657
управление в ActionScript, 648, 661
экспорт, 650
знак равенства (=), оператор присвоения, 138
значения данных и логические выражения, 158

## И

идентификаторы
в операциях присвоения, 138
правила образования, 387
создание имен экземпляров, 339
чувствительность к регистру, 387
идентификаторы связи, 658
иерархия объектов
parseXML() и, 708
XML, класс, 682, 686
XMLnode, класс, 714
обход, 701
удаление узлов, 709
именование переменных, 51, 62
именованные элементы массивов
создание/обращение к, 282
удаление, 283
импорт времени исполнения и библиотеки совместного доступа, 420
импорт кода
использование библиотек совместного доступа, 420
использование директивы #include, 419
индексные переменные в циклах, 188, 193
индексы символов
в текстовых полях, 641, 644
строковые функции и, 115
инициализация
переменных, 49
Smart Clips в качестве альтернативы для, 421
цикла, 187, 190
инкрементирование (++), оператор, 139
интерпретатор, ActionScript (см. ActionScript, интерпретатор), 32
интерфейс с буксировкой, создание, 588
иррациональные числа, 103, 561
итерация в циклах, 39, 188, 190, 193

## К

кавычки в строках, 106
возможные проблемы, 470
кадры
вывод в случайном порядке, 573
выполнение удаленных сценариев, 494
добавление пустых ключевых кадров, 55
добавление сценариев в, 414
ключевые кадры как устройства хранения, 415
количество на временной диаграмме, 610
метки кадров
в сравнении с номерами кадров, 52
добавление, 51
структура слоев и, 46
панель Frame Actions, 51
перемещение головки
воспроизведения на один кадр вперед, 604, 621
воспроизведения на один кадр назад, 605, 634
печать
как векторной графики, 634, 637
как растровых изображений, 636, 637

прикрепление кода к, 30, 414
программирование на основе событий и, 44
создание переменных главной временной диаграммы, 49, 68
текущее положение головки воспроизведения, 587
каналы
баланс левого/правого
определение, 651
установка, 652
определение распределения по левому/правому динамику, 651
установка распределения по левому/правому динамику, 654
категорирование данных, 84
качество вывода Player
_highquality, глобальное свойство, 542
_quality, глобальное свойство, 638
toggleHighQuality, глобальная функция, 677
квадратный корень из 2, 577
клавиатура
обнаружение команд, 549
события
клипов и, 251, 259
кнопок и, 246
классы, 302
(см. также классы объектов), 241
базовый класс ActionScript, 627
встроенные, 328, 479
добавление методов в, 315
надклассы и подклассы, 321
настройка свойств объектов, 313
объектно-ориентированное программирование и, 311
определение функций-конструкторов, 312
создание членов, 313
создание, основание для, 311
классы объектов, 311
(см. также классы), 241
область видимости обработчика событий, 246
прикрепление обработчиков событий к, 241, 269
классы-оболочки
Boolean, 492
Number, 623
String, 662
клипы, 330
Smart Clips и, 333, 421
абсолютные пути, 361, 612, 675
буксировка (см. буксировка клипов), 661
вложенные, доступ через оператор точки, 332
воспроизведение, 633
выключение звука, 655
вычисление расстояния между, 80
главных фильмов (см. главные фильмы), 333
добавление сценариев в, 417
иерархия, 351
имена экземпляров
использование, 349
клипы сценариев, 334
круговое движение, 567
методы, 367, 583
встроенные, 368
вызов, 367
перекрытие методов и глобальных функций, 369
создание, 367
неправильные ссылки, 470
непрозрачность, задание, 586
обрабатывающие клипы, 334
обычные клипы фильмов, 333
ограничивающие рамки, определение, 592
остановка, 608, 660
переменные
доступ с помощью оператора «точка» (.), 331
срок жизни, 76
переменные клипов, 70
печать
как векторной графики, 634
как растрового изображения, 636
поворот, 606
порождающие клипы, 334, 524
дублированные экземпляры, трансформированные, 337
назначение экземпляров стекам, 345
прикрепление кода к, 30
прикрепление обработчиков событий к, 239
применение, 372
прозрачность, задание, 586

свойства, 581
доступ с помощью оператора «точка» (.), 331
встроенные, 365
получение значений, 531
присвоение значений, 647
связанные клипы, 334
символы, 335
экспортирование из библиотеки, 338
скрытие/показ, 613
следование за курсором, 607, 659
создание, 335
создание копий, 523, 589
сообщение размера в байтах, 594
типы, 332
точка регистрации, 585
тройки RGB, 495, 499
getRGB(), 499
setRGB(), 502
удаление из Player, 606, 611, 639, 679
управление методами Flash 4, 746
цвет и прозрачность, программирование, 495, 499
целевой путь, 610
экземпляры, 43, 335
вложенные, 351
динамическое создание ссылок, 357
задание имен, 603
изменение уровней, 343
назначение уровней в стеках программно генерируемых клипов, 346
обмен уровнями, 609
программно генерируемые, 343
расположение в стеке, 343
создание вручную, 336
создание из экспортированных библиотечных символов, 587
создание из экспортированных символов библиотеки, 338
создание с помощью duplicateMovieClip(), 336
создание ссылок с помощью Insert Target Path, 356
ссылки на, 348
стеки документов, 346
стеки уровней, 346
текущий, ссылка на, 349
удаление, 362
предложение with и, 174
ключевые
кадры, добавление, 55
слова, 166
зарезервированные, 385
чувствительность к регистру, 387, 471
ключи (биты), 393
кнопки
активные области, 247
выключение желтого прямоугольника подсветки, 527
добавление кода в, 52
добавление сценариев в, 415
прикрепление обработчиков событий к, 239
код
выполнение
асинхронное (основанное на событиях), 236
синхронное, 235, 414
добавление в кнопки, 415
импорт с помощью директивы #include, 419
комментарии в, 49, 383
обнаружение скрытого, 417
оптимизация создания, 418
организация в функции, 230
отладка, 460
прикрепление к кадрам, 30, 414
советы по написанию оптимального кода, 465
централизация, 230, 269, 416
кодированные строки, создание, 105
кодировка символов, 105, 736, 742
кодовые позиции для символов Latin 1, 105, 736, 740
и коды клавиш, 550
получение, 126, 665
символы, отсутствующие на клавиатуре, 108
создание строк по, 125, 666
сортировка массивов, 293
шестнадцатеричные escape-последовательности и, 525
коды клавиш, 551
коллизии, обнаружение, 586, 598, 601
комбинации клавиш, перехват, 263
комментарии в коде, 49, 383

конкатенация (+), оператор (Flash 5), 34, 91, 139
преобразование данных в строки, 91
константа e (основание натуральных логарифмов), 568
возведение в заданную степень, 568
константы
математические, 103, 561
числовые, 623
конструктор, функция
добавление методов в классы, 315
наследуемые свойства и, 316
определение, 312
переопределение, 386
создание, 318
членов класса, 313
экземпляров объектов, 305, 313
контейнеры данных, хранение ссылок на клипы в, 358
контроль ошибок, 462
косинус, вычисление, 567
круг, перемещение клипа по, 567
круглые клипы, обнаружение коллизий, 598, 601
круглые скобки ()
оператор вызова функции, 163, 207, 307
оператор группирования, 159
приоритеты операторов и, 134
сравнение присвоения метода с вызовом, 311
курсив (поддержка HTML в текстовых полях), 450
курсор
(см. также пользовательский курсор мыши и, точка ввода), 566, 578
показ, 580
разворот клипа в сторону курсора, 566
скрытие, 579, 580
следование клипов за, 607, 659
кэширование, предотвращение с помощью loadVariables(), 558, 560, 601

## Л

лексическая структура ActionScript, 380
литералы, 37
в сравнении с переменными, 84
объектов, 627
строковые, 106
функций, 214
добавление методов в классы, 315
реализация методов с помощью, 310
целочисленные, 99
чисел с плавающей точкой, 100
числовые, 99
логарифм
(см. также натуральный логарифм), 568
логика в фильмах Flash, 38
логические выражения и значения данных, 158
логические операторы, 153
в сравнении с поразрядными операторами, 160
И (&&), оператор (Flash 5), 156
И (and), оператор (Flash 4), 386
ИЛИ (||), оператор (Flash 5), 154
ИЛИ (or), оператор (Flash 4), 386
отрицания (!), оператор (Flash 5), 157
отрицания (not), оператор (Flash 4), 386
локальная область видимости функций, 217
локальные переменные, 70, 80
нелокальные переменные и, 79
создание, 219

## М

маркеры комментариев
/* */, 384
//, 384
маскированные слои, поиск скрытых, 417
массивы, 273
анатомия, 274
ассоциативные, 282
в сравнении с объектами, 304
вложенные (см. вложенные массивы), 278
добавление элементов
без присвоения значений, 284
в конец массива с помощью push(), 284, 486

в начало с помощью unshift(), 285, 491
присвоением значений, 283
с помощью splice(), 287
именованные элементы
создание/обращение к, 282
удаление, 283
как объекты, 297
многомерные, 298
обработка данных с помощью циклов, 188
преобразование в строки
с помощью join(), 296, 484
с помощью tostring(), 297, 490
размер, определение, 280
создание
использование литералов массивов, 277
используя подмножество элементов существующего массива, 487
путем добавления элементов в конец, 288, 483
с помощью оператора new и конструктора Array(), 276, 482
с помощью подмножества элементов существующего массива, 295
сортировка, 293, 488
удаление элементов
из конца массива с помощью pop(), 290, 486
из начала массива с помощью shift(), 291, 487
с помощью splice(), 287, 292, 489
с помощью оператора delete, 289
с помощью свойства length, 289, 485
элементы, 274
извлечение значений, 278
инвертирование порядка, 293, 487
индексы, указание, 278
определение количества, 280, 485
присвоение значений, 279
сортировка, 293, 488
ссылки на, 278
математические
константы, 103, 561
функции, 104, 561
меньше (<), оператор (Flash 5), 112, 149
меньше или равно (<=), оператор (Flash 5), 150
месяц года
получение, 512
время UTC, 515
присвоение, 518
время UTC, 521
метки, кадров, 46
методы, 301, 307
класса XML, 682
классов, в Java и эквивалентные свойства ActionScript, 327
клипов, 367, 583
объектов
предложение with и, 174
фиксированная область видимости, 405
перекрытие методов и глобальных функций, 369
получение значений, 309
присвоение объектам класса, 315
реализация с помощью литералов функций, 310
установка значений свойств объектов, 309
экземпляров, Java и эквивалентные свойства ActionScript, 326
миллисекунды в дате
использование для расчета дат, 508, 513, 518, 523
время UTC, 522
получение, 512
время UTC, 515
присвоение, 517
время UTC, 520
минус (–), оператор вычитания, 35, 103, 140
минуты в дате
получение, 512
время UTC, 515
присвоение, 517
время UTC, 520
многобайтовые и однобайтовые символы, операции, 128
многомерные массивы, 298
многострочные комментарии, 384
множественное наследование, 321
модель выполнения, управляемого событиями, 43, 236
модульный код в ActionScript, 41

мышь
(см. также пользовательские курсоры мыши), 258
показ курсора, 580
разворот клипа в сторону курсора, 566
скрытие курсора, 579, 580
следование клипа за, 607, 659
мышь, события
события клипов и, 251, 258
события кнопок и, 246

## Н

надклассы
класса XML, 714
проверка принадлежности объектов, 325
создание, 321
наследуемые свойства, 316
Object._ _proto_ _ свойство и, 320
Object.constructor, свойство и, 320
использование класса Object для создания глобальных переменных/методов, 326
надклассы и подклассы, 321
переопределение, 319
полиморфизм и, 324
создание с помощью свойства prototype, 317
натуральный логарифм
10 (LN10, свойство), 569
2 (LN2, свойство), 570
вычисление, 570
основание (константа e), 568
возведение в заданную степень, 568
не равно (!=), оператор (Flash 5), 112, 147
не равно (ne), оператор, 93, 128
не число, свойство, 87
нелокальные переменные, 79
непрозрачность в клипах, задание, 586
неравенство
(!=), оператор (Flash 5), 147
(<>), оператор (Flash 4), 148
неявное и явное объявление переменных, 61
неявное преобразование типа данных, 87
нулевой байт, 717
автоматическое выполнение
onData(), обработчик события, 726
обработчика обратного вызова (onXML, свойство), 727

## О

область видимости
возможные проблемы, 471
встроенных классов объектов, 246
обработчика событий, 242
обработчиков событий клипа, 242
обработчиков событий кнопок, 245
переменных, 68
область видимости функции
ActionScript в сравнении с JavaScript, 406
более сложные вопросы, 404
методы объектов, область видимости, 405
область видимости функций, 216
обмен уровней экземпляров, 609
обновляющие предложения в циклах, 187, 190
обрабатывающие клипы, 334
обработчики обратного вызова
XML.onLoad(), обработчик события и, 706
XMLSocket, класс и, 718
XMLSocket.onClose, обработчик события и, 724
XMLSocket.onConnect, обработчик события и, 724
XMLSocket.onXML, обработчик события и, 726
обработчики событий, 44, 237
onClose (XMLSocket, класс), 724
onConnect (XMLSocket, класс), 724
onData()
XML, класс, 682, 705
XMLSocket, класс, 726
onLoad(), 682, 706
onXML (XMLSocket, класс), 726
XML, класс, 682
в сравнении с функциями, 237
клипы фильмов, 584
кнопок
определение, 238
правила для областей видимости, 245

события кнопок для, 246
область видимости, 242
определение, 238
отладка, 470
отсутствие, 470
прикрепление к
встроенным классам объектов, 241, 269
кнопкам/клипам, 239
применение, 270
примеры, 270
создание, 238
срок жизни, 242
сценарии для кнопок, 53
централизация кода, 269
обратная величина для корня из 2, 577
объект узла (см. узел, объект), 683
объектно-ориентированное программирование (ООП), 301
введение в, 301
классы и, 311
краткая сводка, 327
надклассы и подклассы, 321
полиморфизм, 324
ресурсы, 734
объекты, 302
анатомия, 304
в сравнении с массивами, 304
встроенные, 329, 479
документы XML, обработка, 682, 686
инкапсуляция, 302
интерфейсы, 302
массивы как, 297
методы, 174, 307
поведение, 302
принадлежность надклассам, 325
свойства, 174
циклы for-in и, 193
создание экземпляров, 305
экземпляры, 302
объявление
переменных, 61
функции, 207
обычные клипы фильмов, 333
ограничивающие рамки, определение, 592
ограничители блока предложений ({ }), 42, 383
одинарные кавычки (') в строках, 106
однобайтовые и многобайтовые символы, операции, 128
однострочные комменарии, 384
окна, открытие из Flash, 533, 537, 594
округление чисел до произвольного числа десятичных знаков, 101
операнды, 35, 133
количество, 134
различных типов данных, сравнение, 146
операторы, 34, 133
Flash 4 в сравнении с Flash 5, 93
number, тип данных и, 103
арифметические, 139
ассоциативность, 135
вызова функции, 207, 307
выражения и, 133
поразрядные, 392
приоритетность, 134
приоритеты, 104
сложное присвоение, 138
типы данных и, 137
опрос клавиатуры, 264
оптимальный код, советы по написанию, 465
опции текстовых полей, 442
основание 10
логарифм e, 570
преобразование целых чисел в, 631, 633
основание 16
escape-последовательности, 108, 525
значения цветов, 499, 502
целые числа
parseInt() и, 632
основание 2
логарифм e, 571
системы счисления (двоичной), 392
основание системы счисления
parseInt() и, 632
toString() и, 626
отладка, 460
инструменты, 461
методология, 467
проблемы
исправление, 471
определение источника, 468
распознавание/воспроизведение, 467
частые причины, 469

относительные ссылки, 72, 604, 640
отношения операторы (см. сравнения операторы), 148
отсечение дробной части чисел, 546
ошибки
  программные (см. отладка), 467
  устранение, 460

## П

пакетные и управляемые событиями программы, 43
панорамирование звука, 651, 654
параметры, 34
  в сравнении с аргументами, 210
  вызов функций с, 210
  доступ к дополнительным через объект arguments, 221
  доступность предложений, 217
  значения элементарных типов в сравнении с ссоставными, 223
  клипа, 423
    конфигурирование, 423
    удаление/переупорядочение, 425
  количество, 221
  определение, 210
  получение значений из массива arguments, 222
  создание функций с, 209
первым пришел – первым ушел (FIFO) стеки, 286
первым пришел – последним ушел (LIFO) стеки, 286
переменные, 60
  List Variables, команда, 463
  undefined, тип данных, 83, 131
  автоматическая типизация, 66, 85
  автоматическое преобразование значений, 66
  в сравнении с литералами, 84
  временной диаграммы
    доступность, 217
    инициализация, 231
    использование локальных переменных для предотвращения конфликта, 220
    создание, 220
  временные, 217
  вывод на экран, 58
  глобальные
    отсутствие поддержки, 70
    эмуляция, 71, 326
  динамическая установка типа данных, 66
  динамически именуемые, 62
  доступ в различных временных диаграммах, 71
  заглавные буквы в, 51
  изменение значений, 64
  именование, 51, 62
  импорт, 558, 560, 601
  инициализация, 49
    Smart Clips как альтернатива для, 421
  как выражения, 38
  классов, в Java и эквивалентные свойства ActionScript, 327
  клипов, срок жизни, 76
  локальные, 70, 80
  нелокальные и, 79
    создание, 219
  область видимости, 68
  обращение к находящимся на разных уровнях, 74
  объявление, 61
  определение типа данных вручную, 67
  отправка сценариям, выполняемым на сервере, 559
  официальная номенклатура, 31
  прикрепление к уровням документов, 560
  примеры кода, 80
  присвоение значений, 32, 49, 63
  синтаксис точки и, 71
  создание, 61
    переменные временной диаграммы, 49
    переменных временной диаграммы, 68
    текстовых полей, 58
  ссылки
    абсолютные/относительные, 72
    разрешение через цепочку областей видимости, 217
  типы значений, 66
  экземпляров, Java и эквивалентные свойства ActionScript, 326
перенос строки, 106, 621
  в коде ActionScript, 380

поддержка HTML в текстовых полях, 452
переносимость кода и циклы событий клипа, 202
переполнение, 626
платформа, определение (Player), 537, 539
плюс (+), оператор сложения, 34, 103, 139
преобразование данных в строки, 91
поворот клипа в сторону курсора мыши, 566
подпрограммы, локальные переменные в, 79
подстроки
извлечение из строк на основе начальной позиции и длины, 119, 672
положительных или отрицательных позиций, 669
положительных или отрицательных позиций символов, 120
положительных позиций, 673
положительных позиций символов, 118
поиск вхождения
первого, 116, 666
последнего, 117, 668
подпредложения, 167
в предложении if, 178
в циклах for-in, 194
в циклах while, 186
подчеркнутый текст (поддержка HTML в текстовых полях), 450
поиск в строках, 116, 666, 673
полиморфизм, 324
полноэкранный режим, запуск
в автономном Projector, 529
в веб-броузерах, 530
положение по вертикали
клипов/фильмов, 618
курсора мыши, 619
положение по горизонтали
клипов/фильмов, 614
курсора мыши, 616
полужирный текст (поддержка HTML в текстовых полях), 450
пользовательские
курсоры мыши
Mouse.hide() и, 579, 580
mouseMove, событие и, 259
эмуляция, 271
события, 236
добавление кода в кнопки, 415
поразрядные операторы, 392
XOR (^), 398
в сравнении с логическими операторами, 160
И (&), 395
И с присвоением (&=), 136
ИЛИ (|), 396
ИЛИ с присвоением (|=), 136
исключающее ИЛИ (^), 398
исключающее ИЛИ с присвоением (^=), 136
отрицания (~), 399
практические операции, 403
сдвига, 135, 400
вправо без знака, 401
вправо со знаком, 400
вправо со знаком с присвоением (>>=), 136
левого сдвига, 401
порождающие клипы, 334, 524
назначение экземпляров стекам, 345
наследование трансформаций, 337
порядок
выполнения
кода в слоях, 347
обработчиков событий клипов, 264
обхода по Tab
тег якоря HTML не включается в очередь, 453
текстовых полей, настройка, 642
установка, 645
постоянные преобразования, 93
постфиксное
декрементирование (операнд – –), оператор, 141
инкрементирование (операнд ++), оператор, 139
потеря значимости, 626
потоковый звук, 659
предварительные загрузчики
_framesloaded, свойство и, 590
getBytesLoaded() и, 593
доступ к загруженным фильмам, 342

использование булевых значений при создании, 130
использование обработчиков событий для создания, 258
создание предварительных загрузчиков XML, 701
предложения, 165
в сравнении с Actions, 176
вызовы функций, 170, 172
выражений, 170
добавление кода в панели Actions, 30
завершение, 33, 382
область видимости функций, 216
подпредложения, 167
синтаксис, 166
типы, 166
условные, 38, 170
в сценарии расчета, 56
циклов, 170
преобразование
Flash 4 в Flash 5, 93
автоматическая типизация, 87
временное, 93
значений, автоматическое, 66
неожиданный тип данных, 470
операции равенства и преобразования типов, 146
операции сравнения и преобразования типов, 151
постоянное, 93
регистра, функции, 123, 674
строк, раскодирование, 678
температуры, пример, 81
типа данных, явное, 90
типов данных в
булевы значения, 89, 179
строку, 88
числа, 88, 139
префиксное
декрементирование (– –, операнд), оператор, 141
инкрементирование (++, операнд), оператор, 139
прикрепленные звуки, 649
воспроизведение, 656
выключение, 657
приложения сервера
закрытие открытых соединений с приложениями на сервере, 722
открытие соединений с приложениями на сервере, 722
передача данных в формате XML приложениям на сервере, 728
поддержка, 716, 722
приоритеты операторов, 104, 134
присвоение
(=), оператор, 138
значений переменным, 32, 49, 63
пробельные символы
в коде, 380
удаление из документов XML, 684, 686, 688, 698
проверка данных, 462
проверочные выражения в циклах, 190
программирование, введение в, 25
программные функции, 206
доступность, 215
прозрачность в клипах, 495, 499
задание, 586
производительность
измерение с помощью Bandwidth Profiler, 464
при извлечении подстрок, 122
пространства координат (MovieClip, класс), 585
globalToLocal() и, 595
localToGlobal() и, 602
процентные отношения для цветов, 495, 504
пустые
ключевые кадры, добавление, 55
предложения, 176
строки, 106

## Р

рабочий стол
(см. также главные фильмы), 330
globalToLocal() и, 595
localToGlobal() и, 602
высота, 366, 582, 597
ширина, 366, 582, 613
равенство
(= =), оператор (Flash 5), 111, 144
элементарных типов данных, 144
разбиение на параграфы (поддержка HTML в текстовых полях), 451
разделители
в строках, преобразованных из массивов, 296

размывание (см. сглаживание), 615
раскодирование строк, 678
регулярные выражения
  не поддерживаются, 673
  отсутствие поддержки, 118
рекурсия
  нахождение всех клипов, расположенных на временной диаграмме, 359
  создание функций для, 225, 267
  удаление пробельных символов с помощью, 688
ресурсы по программированию игр, 733

## C

самодокументируемый код, 384
свойства
  Number, класс, 623
  XML, класс, 681
  глобальные, 478
  клипов фильмов, 581
    получение значений, 531
    присвоение значений, 647
  константы объекта Math, 103, 561
  неточность в дробной части значения, 581
  объектов, 301, 304
    for-in, циклы и, 307
    назначение обработчиков событий с помощью, 241
    наследование, 316
    настройка, 313
    область видимости функций, 406
    оператор точки (.) и, 162, 306
    получение значений, 309
    предложение with и, 174
    разрешение ссылок через цепочку областей видимости, 217
  прикрепление к функциям, 229
  текстовых полей, 447
  только для чтения и для чтения/записи, 581
связанные клипы, 334
сглаживание
  аппроксимация дробных значений _x, 615
  встроенные шрифты в текстовых полях, 445
  прокрутка текстовых полей, 448
сдвиг
  влево (>>), оператор, 401
  вправо без знака (>>>), оператор, 401
  вправо с присвоением (>>=), оператор, 136
  вправо со знаком (>>), оператор, 400
секунды даты
  получение, 513
    время UTC, 515
  присвоение, 518
    время UTC, 521
сервер/клиент, соединение TCP/IP (XMLSocket, класс), 716, 722
символы
  индексы (см. индексы символов), 641
  клипов, 335
    экспортирование из библиотеки, 338
  подчеркивания во встроенных свойствах клипов, 581
  пробелов в коде, 380
  табуляции в коде, 380
синтаксис
  автоматическая типизация литералов и, 85
  обработчика событий, 238
  обращения к переменным, 75
  предложения, 166
  с косой чертой и свойство _target, 610
  точки
    доступ к переменным, 71
синус, вычисление, 575
синхронизированная анимация, 203
синхронное выполнение кода, 235, 414
системные события, 236
скорость вывода кадров, воздействие на циклы событий временных диаграмм и клипов, 203
скрытие/показ клипов, 613
скрытые слои, проверка, 417
сложение (+), оператор, 34, 139
  преобразование данных в строки, 91
сложные выражения, 84
  автоматическая типизация, 86
  условные проверочные выражения, 179

слои, поиск скрытых, 417
случайные кадры, вывод, 573
смещения для цветов, 495, 504
события, 235
  _changed, событие (текстовые поля), 449
  this, ключевое слово для явной ссылки на объект, 243
  клипа (см. события клипов), 201
  кнопок, 246
  типы, 236
события клипов, 251, 584
  data, 255
  enterFrame, 252
  keyDown, 259
  keyUp, 263
  load, 253
  loops
    enterFrame, событие и, 253
  mouseDown, 259
  mouseMove, 259
  mouseUp, 259
  unload, 254
  ввод пользователя, 258
    updateAfterEvent() и, 267
  воспроизведения фильма, 252
  копирование, 267
  обзор, 251
  определение, 238
  порядок выполнения, 264
  правила для областей видимости, 242
  связанные с вводом данных пользователем
    updateAfterEvent() и, 680
  циклы, 201
    воздействие частоты кадров на, 203
    примеры цикла событий, 202
    проблемы переносимости кода, 202
  эмуляция динамической обработки событий, 269, 270
события кнопок, 246
  dragOut, 249
  dragOver, 249
  press, 247
  release, 248
  releaseOutside, 248
  rollOut, 249
  rollOver, 248
  модель исполнения, основанная на, 236
  обработчики, 416
  эмуляция динамической обработки событий, 270
соединения Flash с серверами, 716, 722
сообщения об ошибках, 461
  этапа исполнения, 462
  этапа компиляции, 462
составные
  данные в сравнении с элементарными, 96
    копирование, сравнение и передача, 389
    передача значений в функцию, 223
  операторы присвоения, 138
специальные клавиши
  коды клавиш для, 741
справочник по языку, ActionScript, 477
сравнения операторы, 112, 148
  преобразование типов данных и, 151
  элементарные типы данных в сравнении с составными, 391
среда редактирования, 411
  Actions, панель, 411
  оформление компонетов в виде Smart Clips, 421
  поиск кода, 417
  расположение кода ActionScript во внешних файлах, 419
  советы по повышению производительности, 418
средства устранения неполадок, 460
ссылки
  абсолютные/относительные, 72, 354, 604, 640
  динамическое создание объектов клипов, 357
  на вложенные экземпляры, 351
  разрешение через цепочку областей видимости, 217
  создание с помощью Insert Target Path, 356
  явные/неявные на клипы, 350
стеки
  pop()/push() и, 286
  внутренних слоев, 343
  документов, 346

порядок исполнения, 347
порядок помещения на стек, 343
программно генерируемые, 343
стереозвучание, 654
строки
выполнение кода с помощью eval(), 126
извлечение
вместе с изучением подстрок, 122
кодовых позиций заданных символов, 665
символов из заданных позиций, 116, 664
целых чисел из, 631, 633
чисел с плавающей точкой в, 631
извлечение подстрок на основе
начальной позиции и длины, 119, 672
положительных или отрицательных позиций, 120, 669
положительных позиций, 118, 673
изучение, 115
вместе с извлечением подстрок, 122
индексы символов, 115
кавычки в, 106
возникающие проблемы, 470
кодирование, 105, 525
конкатенация, 665
на верхнем регистре, создание, 123, 675
на нижнем регистре, создание, 124, 674
операторы равенства/неравенства, 111
описание объектов как, 629
подсчет количества символов, 668
поиск подстрок
первого вхождения, 116
последнего вхождения, 117
поиск подстрок в
первого вхождения, 666
последнего вхождения, 668
получение
кодовых позиций заданных символов, 126
получение числа символов, 115
преобразование
в идентификаторы, 126, 525
в массивы, 121, 670
значений булевых объектов в, 493
значений объектов Number в, 626
массивов в, 296, 297, 484, 490
объектов узлов XML в, 713
производительность извлечения подстрок, 122
пустые, 106
раскодирование, 678
создание из кодовых позиций, 666
создание строк из кодовых позиций, 125
сравнение, 111, 149
учет регистра, 123
строковые
литералы, 106
операторы, 152
структура слоев в фильмах Flash, 46, 343
сценарии, 37
выполняемые удаленно, 494
добавление
в кадры, 414
в клипы, 417
в кнопки, 415
поиск с помощью Movie Explorer, 418
расчета, 55
создание
переменные главной временной диаграммы, 49, 68
сценарий расчета, 55
функции в, 41
сцены
в сравнении с псевдосценами, 539, 542
перемещение головки воспроизведения
к предыдущей сцене, 634
к следующей сцене, 621
счетчики в циклах, 188, 193

## Т

тангенс, вычисление, 577
тайм-аут, эмуляция, 198, 750
таймер обратного отсчета времени, создание, 532

теги якорей (поддержка HTML в текстовых полях), 452
вызов функций ActionScript из, 457
выполнение JavaScript из, 457
не добавляется в обход по Tab, 453
текстовые поля, 439
_changed, событие, 449
Border/Bg, опция, 444
Embed Fonts, опция, 445, 455
Line Display, опции, 443
Max Characters, опция, 445
maxscroll, свойство, 448
Multiline, опция, 444
Password, опция, 444
scroll, свойство, 447
Selectable, опция, 445
Selection, объект, 458, 641
Single Line, опция, 443
Variable, опция, 444
Word Wrap, опция, 445
выделение символов в фокусе, 647
динамические текстовые поля
извлечение значений, 441
изменение содержимого, 440
создание, 440
жесткий перенос в, 443
наибольшее допустимое значение scroll, 578
неверный тип данных в, 470
неопределенные текстовые поля и циклы for-in, 458
нераспознаваемые теги и атрибуты, 454
обработка событий, 449
опции, 442
переменные, создание, 58
поддержка HTML, 449
получение индекса
первого выделенного символа, 642
последнего выделенного символа, 643
точки ввода, 643
получение фокуса ввода с клавиатуры, 644
правильное использование кавычек с атрибутами HTML, 453
преобразование в числа, 91
прокрутка, 640
свойства, 447
сглаживание, 448
встроенные шрифты, 445
текст не виден, 445, 454
текстовые поля для ввода данных пользователем
добавление в формы Flash, 432
извлечение/использование значений, 442
изменение содержимого, 441
использование в заполняемых формах, 442
создание, 441
установка фокуса ввода с клавиатуры, 644, 647
шрифты устройств и, 445
текстовые узлы, 703
raw XML source code in, 708
XML.nodeValue, свойство и, 704
пробельные символы в, 684, 686, 688
создание, 695
тело
функции, 207
цикла, 191
тернарные операторы, 134
типы данных, 82
встроенные ActionScript, 83
операторы и, 137
определение вручную, 67, 94
переменные и, 66
преобразования, 86
операции равенства и, 146
операции сравнения и, 151
преобразование в
булевы значения, 89, 179
строки, 88
числа, 88, 139
элементарные, 98
в сравнении с составными, 96
равенство, 144
типы узлов, 703
только для чтения и для чтения/записи свойства, 581
точечный синтаксис
абсолютные пути клипов, 612, 675
абсолюттные пути к клипам, 361
управление клипами, 747
цепочка областей видимости и, 218
точка (.), оператор, 162
доступ к свойствам клипа, 331
обращение к свойствам объектов, 306

обход иерархии экземпляров клипов, 351
точка ввода
получение индекса, 643
установка в тексте, 647
точка с запятой (;)
завершение предложений с помощью, 33, 166, 208, 382
отсутствие, 470
точки (см. пространства координат), 585
регистрации клипов, 585
_x, свойство и, 615
_y, свойство и, 618
точность представления чисел с плавающей точкой, 101
трансформации (настройка цветов), 496, 504
тригонометрия
арккосинус, вычисление, 563
арксинус, вычисление, 564
арктангенс, вычисление, 565
косинус, вычисление, 567
синус, вычисление, 575
тангенс, вычисление, 577

## У

удаление локальных переменных, 78
узел, объект
XML, класс, 683, 686
XMLnode, класс, 714
добавление новых узлов, 689
доступ к дочерним узлам, 691
значение текущего узла, 704
имя текущего узла, 702
перемещение узлов между документами, 690
создание
копий, 692
текстовых узлов, 695
узлов элементов, 694
ссылка на
ближайшего предка, 707
предшествующий узел, 708
следующий узел, 701
типы, 703
удаление из иерархии объектов, 709
установка нового предыдущего одноуровневого, 698
хранение атрибутов элементов, 690
узлы элементов, 703
XML.nodeValue, свойство и, 704
создание, 694
умножение
(*), оператор, 103, 141
с присвоением (*=), оператор, 136
унарные операторы, 134
унарный (– ) оператор отрицания, 143
управление циклом, 191
управляемая событиями модель исполнения, 43
уровни документов, 74
_leveln, глобальное свойство, 346, 355, 554
загрузка файлов SWF
посредством loadMovie(), 346, 355
посредством loadMovieNum(), 557
печать кадров
как векторной графики, 637
как растровых изображений, 637
прикрепление внешних переменных к, 560
удаление с помощью unloadMovie(), 362
удаление фильмов из, 679
условные выражения, 177
блоки предложений и, 185
условный (?:) оператор, 163, 181

## Ф

факториалы, вычисление с/без рекурсии, 225
файлы сценариев, импорт, 419, 543
флаги (биты), 393
фокус ввода с клавиатуры
определение положения, 644
установка для заданного текстового поля, 644, 647
формы Flash, 429
ввод данных клиентом, 430
данные
обработка и результаты, 431
отправка серверу, 430, 433
получение результатов с сервера, 432, 436
проверка, 430
добавление
кнопки Reset, 437

кнопки Submit, 432
текстового поля для данных пользователя, 432
проблемы взаимодействия клиент/сервер, 430
создание заполняемых форм, 432
текстовые поля для ввода данных пользователем, 442
цикл данных, 429
функции, 41, 206
arguments, объект, 221
в сравнении с Actions, 227
внутренние (см. внутренние функции), 227
возврат значений из, 213
встроенные, 42
вызов, 163, 207, 215, 307
как методов, 308
с параметрами, 210
динамические, 214
обработчики событий клипов и, 269
доступность, 215
завершение, 212
запуск, 207
как объекты, 228
локальная область видимости, 217
локальные переменные и, 77
обеспечение сохранности, 216
область видимости функций (см. область видимости функций), 216
объявление, 207
организация кода в, 230
оставные данные в сравнении с элементарными, 223
параметры (см. параметры), 221
передача
информации в, 208
нескольких аргументов, 211
при инициализации, 49
рекурсивные, 225
свойства, прикрепление к, 229
создание, 207
с параметрами, 209
сравнения, создание, 294
срок жизни, 216
централизация кода, 230
элементарные типы данных в сравнении с составными, 392
функциональные клавиши
перехват, 261

## Х

хеши, эмуляция, 282
Хранилище кода, 17

## Ц

цвет
генерирование случайного, 498
значения смещений, 495, 504
изменение яркости, 498
процентные отношения, 495, 504
трансформации, 496, 504
установка для клипа, 502, 504
целочисленные литералы, 99
целые числа, 98
извлечение из строк, 631, 633
цепочка
областей видимости, 217, 404
прототипов, проход по, 325
циклы, 170, 186
break, предложение в, 195
continue, предложение в, 196
бесконечные, 187, 191
избегание, 197
вложенные, 190
временных диаграмм, 198
воздействие частоты кадров на, 203
диаграммы пустого клипа, создание, 200
создание обрабатывающих клипов, 334
досрочное завершение, 195
индексные переменные, 188, 193
инициализация, 187, 190
итераторы, 188, 193
итерация в, 39, 190
проверочные предложения в, 190
событий клипов (см. события клипов, циклы), 201
счетчики, 188, 193
терминология, 190
предложения обновления в, 187, 190

## Ч

час суток
получение, 512
время UTC, 514
установка, 517

время UTC, 520
частное, 141
час
в аналоговом стиле, создание, 372
создание, 372
чат, создание клиента, 718, 722
числа
большее из двух, определение, 571
возведение в заданную степень, 100, 572
вычисление абсолютного значения, 563
квадратный корень, вычисление, 576
меньшее из двух чисел, определение, 572
натуральные логарифмы, вычисление, 570
округление, 575
до ближайшего большего целого, 566
до ближайшего меньшего целого, 104, 569
определение угла по точке, 565, 566
отсечение дробной части, 546
случайные, генерирование, 104, 573, 639
сравнение, 149
числа с плавающей точкой, 98
parseFloat() и, 92, 631
литералы, 100
точность представления, 101
числитель, 141
число месяца
назначение, 516
время UTC, 519
получение, 511
время UTC, 514
числовые литералы, 99
чувствительность к регистру в ActionScript, 471


## Ш


шестнадцатеричные (по основанию 16)
escape-последовательности, 108, 525
значения цветов, 499, 502
целые числа
parseInt() и, 632
ширина клипов/фильмов
в пикселах, 613
в процентном отношении, 617
шрифты
встроенные, 445, 455
устройств, использование в текстовых полях, 445


## Э


экземпляры
классов объектов, 302
динамическое присвоение свойств объектов, 314
создание, 305, 313
клипов, 43
экранные текстовые поля (см. текстовые поля), 439
экспонента в литералах чисел с плавающей точкой, 100
экспорт звуков, 650
элементарные
булевы значения, 492
данные в сравнении с составными, 96
передача в функции, 223
значения объектов, 630
копирование, сравнение и передача, 389
типы данных, 98
равенство, 144


## Я


явное и неявное объявление переменных, 61
явное преобразование типа данных, 90
японские символы (Shift-JIS), 105
шестнадцатеричные escape-последовательности и, 525
яркость, установка с помощью объекта Color, 498

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-028-6, название «ActionScript. 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.