Дмитрий Альберт
Елена Альберт

bhv

ActionScript 2.0



- Детальное описание синтаксиса и возможностей языка
- Динамические эффекты и интерактивная анимация
- Объектно-ориентированное программирование в стиле ActionScript 2.0

Наиболее полное руководство

+CD

В ПОДЛИННИКЕ®

Дмитрий Альберт
Елена Альберт

ActionScript 2.0

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.06
ББК 32.973.26-018.1
А56

Альберт Д. И., Альберт Е. Э.

А56 ActionScript 2.0. — СПб.: БХВ-Петербург, 2005. — 1136 с.: ил.

ISBN 5-94157-567-X

Книга является подробным руководством по языку сценариев ActionScript 2.0 для среды Macromedia Flash MX 2004. Описаны основные возможности, синтаксис и семантика языка, его базовые элементы: переменные, типы данных, циклы, условные операторы, массивы, функции. Рассмотрены событийная модель и процедура трансляции событий, встроенные классы и технология объектно-ориентированного программирования в стиле ActionScript 2.0. Изложены основанный на этой технологии подход к трехмерному моделированию и процедура создания пользовательских компонентов.

Книга написана ясным и логичным языком. Материал организован так, чтобы ее можно было использовать и как обучающее руководство, и как справочное пособие. В начале каждой темы приводятся теоретические сведения, которые затем иллюстрируются конкретными практическими примерами. В книгу включено более 100 примеров, что позволит читателю приобрести практические навыки работы с языком. Все исходные файлы примеров помещены на сопровождающий компакт-диск.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Татьяна Лапина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 17.06.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 91,59.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.02.953.Д.006421.11.04
от 11.11.2004 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ОАО "Техническая книга"
190005, Санкт-Петербург, Измайловский пр., 29.

ISBN 5-94157-567-X

© Альберт Д. И., Альберт Е. Э., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Введение **1**

Часть I. Основы ActionScript **3**

Глава 1. Представление о Flash **5**

Компоненты программы Flash MX 2004 Professional 5
Процесс создания Flash-проекта 6
Системные требования программы Flash MX 2004 Professional 7
Обзор интерфейса 8
Основы работы с векторным редактором Flash MX 2004 15
Понятие о векторной графике 15
Иерархия графических объектов Flash 17
Операции над объектами и трансформация объектов 21
Настройка глобальных параметров фильма 23
Работа с цветом 24
Работа с растровой графикой 26
Структура монтажной линейки 28
Работа со слоями 29
Работа с кадрами 31
Символы и экземпляры 35
Типы символов и операции с ними 36
Экземпляр символа и его свойства 38
Работа с библиотекой 40
Работа с общими библиотеками (Shared Libraries) 41
Разрешение вопросов политики безопасности при импорте внешних данных из других доменов 44
Анимация 45
Покадровая анимация (Frame by frame animation) 47
Анимация формы (Shape tweening) 47
Анимация движения (Motion tweening) 49
Использование звука в фильме 50
Публикация исходного документа 54
Формат Flash (SWF) 56
Формат HTML 59
Формат Windows Projector (exe) 64

Глава 2. Новые возможности ActionScript **65**

ActionScript 2.0 65
Зависимость от регистра 65

Строгий контроль типа данных ... 66
Формальный синтаксис классов ... 66
Новая архитектура компонентов ... 67
Массивы ... 67
Работа с клипами ... 68
Связывание символа с классом ... 68
Компилированные клипы ... 68
Контроль заполнения уровней стека ... 68
Свойство *_lockroot* ... 69
Метод *getSWFVersion()* ... 69
Работа с текстом ... 69
Поддержка CSS (Cascading Style Sheets) ... 69
Возможность загрузки изображений и внешних SWF-документов в текстовый блок ... 70
Прокрутка текста с помощью колеса мыши ... 70
Контроль статического текста ... 70
Класс *MovieClipLoader*. Организация процесса загрузки ... 71
Класс *Sound*. Поддержка ID3-тегов второй версии ... 71
Динамическая загрузка видео ... 72
Управление контекстным меню проигрывателя ... 72
Объект *System.capabilities* ... 73
Печать из фильма ... 73
Политика безопасности ... 74

Глава 3. Введение в ActionScript. Среда разработки ... 75

Представление о сценариях ActionScript ... 75
Использование редактора ActionScript для создания сценариев ... 76
Раздел *Script pane* (Окно сценария) ... 78
Раздел *Actions toolbox* ... 87
Раздел *Script navigator* ... 89
Носители сценариев AS ... 89
Синтаксические нормы и правила ... 92
Предложения ... 92
Комментарии ... 93
Имена ... 94
Функции ... 95
Функция *trace()* ... 96
Инициирование и последовательность выполнения сценариев. Системные и пользовательские события ... 97
Последовательность выполнения сценариев кадров ... 98
Сценарии кнопок и клипов ... 99
Представление об объектно-ориентированном программировании ... 101
Создание сценария во внешнем файле ... 103
Применение директивы *#include* ... 103
Проекты. Панель *Project* ... 104
Работа с проектами ... 104

Глава 4. Переменные ... 113

Понятие переменной ... 113
Создание и типизация переменной ... 114
Явное объявление переменной ... 114
Неявное объявление переменной ... 117
Динамическое создание переменной ... 117
Присваивание, извлечение и изменение значения переменной ... 118
Присваивание значения ... 118
Автоматическая типизация переменной ... 120
Извлечение значения ... 121
Область видимости и адресация переменных ... 121
Вопросы области видимости ... 121
Локальные переменные ... 123
Глобальные переменные ... 126
Адресация переменных ... 129
Время жизни переменной ... 135
Удаление переменной ... 136

Глава 5. Элементарные типы данных ... 138

Элементарные и ссылочные типы данных ... 138
Числовой тип. Работа с числами ... 140
Литералы чисел ... 141
Специальные числовые значения ... 142
Операции над числами ... 144
Встроенные методы для работы с числами ... 147
Строковый тип. Работа со строками ... 148
Литералы строк ... 148
Операции над строками ... 151
Встроенные свойства и методы для работы со строками ... 153
Булев тип ... 164
Логические операции ... 165
Встроенные методы для работы с булевыми значениями ... 167
Типы *null* и *undefined* ... 168
Значение *null* ... 168
Значение *undefined* ... 168
Явное и автоматическое преобразование элементарных типов данных ... 169
Явное преобразование типов данных ... 169
Автоматическое преобразование типов данных ... 172

Глава 6. Выражения и операторы ... 176

Приоритет операторов ... 176
Ассоциативность ... 180
Модификация значения ... 180
Простое присваивание ... 181
Составное присваивание ... 182
Инкрементирование/декрементирование ... 182

Побитовые операции ... 184
Битовое представление числа ... 185
Применение двоичных чисел ... 187
Логические побитовые операции ... 188
Побитовые операции сдвига ... 193

Глава 7. Управляющие структуры: Операторы ветвления. Циклы ... 198

Ветвление кода ... 198
Предложение *if* ... 199
Предложение *else* ... 202
Предложение *else if* ... 206
Предложение *switch* ... 210
Многократное выполнение кода ... 214
Понятие программного цикла ... 215
Цикл *while* ... 216
Цикл *do while* ... 218
Цикл *for* ... 219
Цикл *for...in* ... 222
Контроль выхода из цикла ... 224
Вложенные циклы ... 225
Циклы монтажной линейки ... 226

Глава 8. Пользовательские функции ... 231

Общие сведения о функциях ... 231
Создание функции ... 232
Объявление функции ... 232
Использование литерала функции ... 233
Параметры функции ... 234
Контроль типа параметров ... 235
Класс *Arguments*. Контроль фактических параметров функции ... 236
Возвращение значения ... 240
Контроль типа возвращаемого значения ... 243
Вызов функции ... 245
Вызов функции по имени ... 245
Вызов функции с помощью ссылки ... 246
Вызов функции с помощью метода *apply()* ... 248
Вызов функции с помощью метода *call()* ... 250
Периодический вызов функции. Глобальные функции *setInterval()* и *clearInterval()* ... 250
Вопросы области видимости и времени существования функций ... 255
Область видимости функции ... 255
Доступ к функции ... 258
Время жизни функции ... 259
Рекурсивные функции ... 259

Глава 9. Массивы 262

Понятие массива 262
Создание массива 263
Конструктор массива 263
Литерал массива 264
Глобальная функция *Array()* 265
Вывод массива в панель *Output* 266
Обращение к элементам массива 266
Операции над элементами массива 267
Определение количества элементов в массиве 267
Присваивание значения элементу массива 268
Добавление элементов в массив 269
Строковое представление массива 274
Сортировка массива 276
Извлечение части массива 288
Удаление элементов массива 289
Ассоциативные массивы 290
Многомерные массивы 292

Часть II. Базовые встроенные классы ActionScript 299

Глава 10. Класс *Button*. Работа с кнопками 301

Создание кнопки в рабочей среде 302
Кнопка как объект *AS* 304
Имя экземпляра 305
Режимы кнопки 306
Обработка событий кнопки 307
События кнопки 307
Обработчик событий кнопки 308
Контроль нажатия клавиатурной клавиши 311
Область видимости обработчика событий кнопок 313

Глава 11. Обработка событий клипов. Событийная модель ActionScript 315

Обработчик событий *on()*. Использование клипов в качестве кнопок 316
Создание клипа-кнопки 316
Область видимости обработчика событий *on()* клипа 317
События клипов 319
Событие *enterFrame* 320
Событие *load* 321
Событие *unload* 322
Событие *data* 322
Событие *mouseMove* 323
Событие *mouseDown* 323
Событие *mouseUp* 323
Событие *keyDown* 324

Событие *keyUp* ... 324
Область видимости обработчика *onClipEvent()* ... 325
Применение методов обработчиков событий ... 325
Понятие метода обработчика событий ... 326
Вопросы области видимости методов обработчиков событий кнопок и клипов ... 329
Удаление методов обработчиков событий ... 331
Применение методов обработчиков событий с использованием адресации ... 332
Трансляция событий ... 333
Передача и обработка событий ... 334
Прекращение трансляции событий ... 337
Создание источников событий. Объект *AsBroadcaster* ... 337
Инициализация источника и трансляция событий ... 338
Применение объекта *AsBroadcaster* ... 340

Глава 12. Класс *MovieClip*. Работа с клипами ... 344

Встроенные свойства класса *MovieClip* ... 344
Координаты. Свойства *_x* и *_y* ... 345
Размеры. Свойства *_width*, *_height*, *_xscale* и *_yscale* ... 347
Поворот. Свойство *_rotation* ... 352
Прозрачность. Свойство *_alpha* ... 352
Имя. Свойство *_name* ... 353
Отображение. Свойство *_visible* ... 354
Координаты мыши. Свойства *_xmouse*, *_ymouse* ... 355
Кадры. Свойства *_totalframes*, *_currentframe* и *_framesloaded* ... 358
Свойства клипов-кнопок: *useHandCursor*, *enabled*, *trackAsMenu*, *hitArea* ... 361
Создание выпадающего списка ... 363
Прочие свойства ... 369
Адресация с помощью предложения *with* ... 374
Управление воспроизведением ... 376
Остановка и запуск воспроизведения ... 376
Переход между кадрами ... 378
Управление воспроизведением монтажной линейки ... 383
Организация навигации внутри фильма ... 385
Программный реверс анимации ... 386
Загрузка сетевого ресурса ... 388
Организация навигации между Web-страницами ... 391
Динамическое создание клипов ... 391
Стеки графических объектов Flash ... 392
Стек документов проигрывателя ... 394
Присоединение экземпляра. Метод *attachMovie()* ... 394
Дублирование экземпляра. Метод *duplicateMovieClip()* ... 400
Контроль уровней стека ... 406
Создание пустого клипа. Метод *createEmptyMovieClip()* ... 409
Удаление динамически созданных клипов ... 409

Проверка состояния загрузки ... 411
Метод *getBytesTotal()* ... 411
Метод *getBytesLoaded()* ... 412
Создание предварительного загрузчика ... 412
Загрузка внешних Flash-документов и изображений ... 415
Загрузка на уровень стека проигрывателя ... 415
Организация навигации с использованием динамической загрузки внешних документов ... 417
Особенности загрузки в клип ... 418
Создание интерактивного слайд-шоу ... 420
Использование абсолютных ссылок во внешних документах ... 423
Загрузка переменных из внешнего источника ... 425
Удаление загруженных документов и содержимого клипов ... 427
Преобразование координат и контроль границ клипа ... 428
Преобразование в глобальные координаты ... 428
Преобразование в локальные координаты ... 430
Контроль границ клипа ... 431
Равномерное распределение клипов ... 431
Перетаскивание объектов ... 436
Перетаскивание с помощью метода *startDrag()* ... 437
Прекращение перетаскивания ... 438
Контроль наложения клипов ... 440
Контроль пересечения ограничивающих рамок клипов ... 440
Контроль попадания точки в заданную область ... 443

Глава 13. Динамическое рисование ... 446

Рисование прямолинейных сегментов ... 447
Задание начальной позиции рисования ... 447
Определение атрибутов контура ... 448
Соединение точек прямолинейным отрезком ... 448
Удаление графики ... 449
Создание инструмента "карандаш" ... 450
Применение сплошной заливки ... 454
Рисование многоугольников и звезд ... 457
Применение градиентной заливки ... 463
Определение матрицы трансформации градиента ... 465
Рисование криволинейных сегментов ... 479
Динамическое маскирование ... 483

Глава 14. Применение динамического и пользовательского текста ... 488

Создание и настройка параметров текстового блока в рабочей среде ... 489
Настройка параметров текстового блока в рабочей среде ... 490
Встроенные и машинно-независимые шрифты ... 495
Использование текстового блока, созданного в рабочей среде ... 496
Использование шрифтовых символов ... 497

Динамическое создание и форматирование текста ... 500
Класс *TextField*. Настройка параметров текстового блока ... 501
Класс *TextFormat*. Свойства форматирования текста ... 510
Создание всплывающих подсказок ... 517
Получение информации о параметрах форматирования текста ... 522
Получение информации о метриках шрифта ... 523
Проверка наличия у пользователя используемого шрифта ... 526
Организация прокрутки текста ... 527
Автоматическая прокрутка текста ... 527
Свойства управления вертикальной прокруткой текста ... 528
Свойства управления горизонтальной прокруткой текста ... 529
Контроль прокрутки содержимого текстового блока ... 531
Примеры реализации текстовой прокрутки ... 532
Контроль выделения и передачи фокуса. Класс *Selection* ... 540
Передача фокуса ... 540
Обнаружение носителя фокуса ... 542
Контроль передачи фокуса ... 543
Определение границ выделения и положения курсора ... 544
Управление выделением ... 547
Управление порядком переноса фокуса с помощью клавиши табуляции ... 548
Обработка событий текстового блока ... 550
Замена содержимого текстового блока ... 554
Удаление текстового блока ... 557
Форматирование текста с помощью HTML ... 558
Активизация поддержки HTML и вывод текста с применением HTML-форматирования ... 559
Теги HTML, используемые для форматирования текста ... 560
Поддержка и использование CSS. Класс *StyleSheet* ... 572
Создание таблицы стилей ... 573
Определение стилей ... 576
Добавление стилей в таблицу ... 581
Получение информации об используемых стилях ... 593
Очистка таблицы стилей ... 595
Загрузка текстовых данных из XML-документа ... 595
Контроль статического текста. Класс *TextSnapshot* ... 601
Создание объекта *TextSnapshot* ... 602
Чтение и поиск в статическом тексте ... 603
Выделение статического текста цветом ... 605

Глава 15. Класс *Math*. Выполнение математических вычислений ... 611

Встроенные свойства. Математические константы ... 611
Возведение числа в степень и извлечение корня ... 612
Вычисление логарифма числа ... 613
Определение наибольшего и наименьшего значений ... 614
Определение модуля числа ... 615
Округление значений ... 616
Математическое округление ... 616

Округление до большего целого ... 617
Округление до меньшего целого ... 617
Тригонометрические функции ... 617
Применение генератора случайных чисел ... 624
Эффект падающего снега ... 625

Глава 16. Класс *Date*. Контроль времени и даты ... 629

Всемирное скоординированное время и машинное время ... 629
Создание экземпляра класса *Date* ... 631
Вызов конструктора без параметров ... 631
Вызов конструктора с передачей ему машинного времени ... 631
Вызов конструктора с передачей ему параметров в формате даты ... 632
Определение и установка параметров даты ... 633
Определение местного времени и даты ... 634
Определение UTC-времени и даты ... 636
Установка местного времени и даты ... 636
Установка UTC-времени и даты ... 638
Работа с машинным временем ... 638
Создание календаря ... 639
Глобальная функция *getTimer()* ... 644

Глава 17. Классы *Mouse* и *Key*. Контроль мыши и клавиатуры ... 648

Контроль отображения курсора и обработка событий мыши ... 648
Управление отображением курсора ... 649
Обработка событий мыши ... 649
Контроль клавиатуры ... 655
События клавиатуры ... 655
Определение характеристик и проверка состояния клавиши ... 656

Глава 18. Класс *Color*. Работа с цветом ... 664

Цветовая модель RGB ... 665
Цветовая модель HSB ... 668
Перевод цвета из модели RGB в модель HSB и наоборот ... 670
Создание объекта *Color* ... 677
Окрашивание клипов. Методы *setRGB()* и *getRGB()* ... 678
Формирование каталога цветов на основе безопасной Web-палитры.
Сортировка по тону ... 681
Цветовые трансформации. Методы *setTransform()* и *getTransform()* ... 686
Типы цветовых трансформаций ... 686
Применение методов *setTransform()* и *getTransform()* ... 687
Создание панели синтеза цвета на основе модели HSB ... 691
Создание интерактивного цветового редактора ... 696

Глава 19. Класс *Sound*. Управление звуком ... 698

Создание объекта *Sound* ... 698
Присоединение звука ... 700

Управление воспроизведением звука ... 701
Запуск воспроизведения ... 701
Остановка воспроизведения ... 702
Обработка окончания звука ... 704
Определение длительности и текущей позиции воспроизведения звука ... 704
Загрузка внешнего звукового файла ... 706
Загрузка звука в потоковом и событийном режимах ... 706
Контроль состояния загрузки звука ... 707
Проверка окончания загрузки звука ... 709
Поддержка ID3-тегов ... 710
Управление уровнем звука ... 714
Управление балансом каналов ... 716
Управление выводом каналов ... 718
Пример программного управления звуком. Интерактивная перкуссия ... 720

Глава 20. Управление процессом загрузки внешних документов и изображений. Взаимодействие между SWF-документами ... 727

Класс *MovieClipLoader* ... 727
Создание экземпляра класса ... 728
Методы класса *MovieClipLoader* ... 728
Обработка событий ... 730
Начало загрузки. Событие *onLoadStart* ... 731
Поступление данных. Событие *onLoadProgress* ... 734
Окончание загрузки. Событие *onLoadComplete* ... 734
Инициализация загруженного документа. Событие *onLoadInit* ... 735
Ошибка при загрузке. Событие *onLoadError* ... 736
Пример управления загрузкой нескольких документов ... 737
Вопросы политики безопасности, связанные с загрузкой внешних SWF-документов ... 742
Класс *LocalConnection*. Взаимодействие между SWF-документами ... 744
Механизм взаимодействия ... 744
Инструментарий класса *LocalConnection* ... 746
Объявление методов объекта *LocalConnection* в документе-получателе ... 746
Открытие соединения. Метод *connect()* ... 747
Отправка данных получателю ... 748
Закрытие соединения ... 751
Пример организации взаимодействия документов ... 752
Вопросы политики безопасности, связанные с взаимодействием документов ... 754
Доменная безопасность ... 755
Безопасность протоколов ... 758

Глава 21. Класс *SharedObject*. Хранение информации на стороне клиента ... 759

Понятие об объекте *SharedObject* ... 759
Ограничения локального хранения информации ... 760
Вопросы безопасности локального хранения информации ... 762

Создание объекта *SharedObject* ... 763
Запись и чтение данных в объекте *SharedObject* ... 766
Сохранение объекта *SharedObject* ... 768
Определение объема локального объекта ... 771
Удаление локальных данных ... 771
Пример применения объекта *SharedObject* для хранения пользовательских настроек интерфейса ... 772

Глава 22. Контроль параметров документа и проигрывателя. Информация о системе и оборудовании ... 779

Класс *Stage*. Контроль размеров и масштабирования окна фильма ... 779
Управление масштабированием фильма ... 779
Управление выравниванием фильма ... 782
Контроль размеров сцены и окна проигрывателя ... 784
Управление отображением меню проигрывателя ... 789
Управление автономным проигрывателем. Функция *fscommand()* ... 790
Класс *ContextMenu*. Управление контекстным меню проигрывателя ... 792
Механизм управления контекстным меню ... 792
Управление стандартным содержимым меню ... 794
Добавление и настройка пользовательских элементов меню ... 798
Примеры управления контекстным меню ... 801
Информация о системе и оборудовании. Объект *System.capabilities* ... 804

Глава 23. Класс *PrintJob*. Печать из Flash-фильма ... 810

Механизм создания и передачи задания на печать ... 811
Вызов окна с настройками параметров печати ... 812
Отправка задания в спулер печати ... 814
Вывод содержимого на печать ... 816
Применение класса *PrintJob* ... 816

Глава 24. Динамическая загрузка и воспроизведение видео ... 824

Формат *Flash Video*. Конвертирование видео в формат FLV ... 827
Механизм выполнения загрузки FLV-файлов ... 832
Создание объекта *Video*. Методы и свойства видеообъекта ... 833
Открытие локального соединения ... 834
Контроль загрузки и воспроизведения FLV-видео. Класс *NetStream* ... 835
Инициирование выполнения загрузки ... 835
Контроль процесса загрузки ... 836
Установка размера буфера ... 837
Управление воспроизведением ... 838
Ручной поиск, перемотка ... 839
Контроль звукового сопровождения ... 840
Обработка событий объекта *NetStream* ... 842
Прекращение загрузки ... 846
Создание видеопроигрывателя с динамической загрузкой FLV-файла ... 846

Часть III. Объектно-ориентированное программирование. Создание компонентов 855

Глава 25. Создание пользовательских классов 857

Файлы классов. Понятие *classpath* 858
Объявление класса 861
Создание экземпляра класса 863
Пакеты 864
Объявление свойств 865
Установка слежения за значением свойства 869
Объявление конструктора 871
Объявление методов 874
Регулирование доступа к членам класса 876
Статические члены класса 880
Организация считывания и установки значений свойств. Определение *get/set*-методов 882
Динамические классы 886
Суперклассы и подклассы 888
Представление о внутреннем механизме реализации наследования 893
Связывание символа с пользовательским классом 895
Связывание символа с классом с помощью окна *Linkage Properties* 896
Связывание символа с классом с помощью метода *Object.registerClass()* 898
Директивы *#initclip* и *#endinitclip* 899
Интерфейсы 901
Пример создания и применения пользовательских классов 903

Глава 26. Трехмерное моделирование и анимация 918

Понятие о матрицах 918
Определение матрицы и ее представление в ActionScript 919
Умножение матриц 920
Класс *Calculus* 921
Изображение и преобразование трехмерных точек 924
Проецирование трехмерных точек на плоскость 928
Создание класса *Vertex3D* 932
Класс Vertex3D. Трехмерные преобразования и визуализация точек 933
Пример реализации: управление центрированным кубом со срезанным углом с помощью клавиатуры 945
Пример реализации: управление трехмерной плоскостью с произвольным числом точек с помощью мыши 952

Глава 27. Создание пользовательских компонентов 959

Представление о компонентах 960
Общий процесс создания компонента 962
Ключевые этапы создания компонента 962
Проектирование компонента 964
Создание клипа компонента 967

Объявление класса компонента ... 969
Объявление свойств ... 970
Применение тегов метаданных ... 973
Тег *Inspectable*. Описание параметров внешнего интерфейса компонента ... 973
Тег *InspectableList*. Список внешних параметров ... 978
Тег *Event*. Описание событий компонента ... 979
Тег *IconFile*. Создание пиктограммы компонента ... 979
Объявление *get/set*-методов ... 980
Трансляция и обработка событий компонентов ... 988
Инициализация источника событий ... 988
Трансляция события. Событийный объект ... 990
Обработка событий компонентов ... 991
Трансляция пользовательского события *onUpdate* ... 997
Инициализация компонента ... 998
Изменение размеров компонента ... 1000
Динамическое присоединение графических элементов компонента ... 1001
Компоновка составляющих элементов ... 1003
Обеспечение функциональности графических элементов компонента ... 1004
Использование режима предварительного просмотра (Live Preview) ... 1007
Распространение компонента ... 1008
Компиляция клипа компонента ... 1009
Экспортирование SWC-файла ... 1009
Создание MXP-пакета расширения ... 1010

Глава 28. Обработка исключительных ситуаций. Отладка сценариев. Тестирование проекта ... 1014

Обработка исключительных ситуаций ... 1014
Тестирование проекта ... 1028
Управление отображением и воспроизведением фильма ... 1028
Эмуляция загрузки ... 1029
Отчет о размерах элементов фильма ... 1034
Средства отладки сценариев ... 1035
Панель *Output* ... 1035
Применение отладчика (панель *Debugger*) ... 1039

Приложения ... 1051

Приложение 1. Справочник: методы, свойства и обработчики событий основных встроенных классов ActionScript ... 1053

Класс *Array*. Создание и обработка массивов ... 1053
Класс *Button*. Работа с кнопками ... 1054
Класс *Color*. Работа с цветом ... 1056
Класс *ContextMenu*. Управление контекстным меню проигрывателя ... 1057
Класс *ContextMenuItem*. Пользовательские элементы контекстного меню ... 1057
Класс *Date*. Контроль времени и даты ... 1058

Класс *Error*. Исключительные ситуации ... 1060
Класс *Key*. Контроль клавиатуры ... 1060
Класс *LocalConnection*. Взаимодействие между фильмами ... 1060
Класс *Math*. Выполнение математических вычислений ... 1061
Класс *Mouse*. Контроль мыши ... 1063
Класс *MovieClip*. Работа с клипами ... 1063
Класс *MovieClipLoader*. Управление процессом загрузки внешних документов и изображений ... 1069
Класс *NetConnection*. Установка локального соединения для динамической загрузки FLV-файлов ... 1070
Класс *NetStream*. Динамическая загрузка и воспроизведение FLV-файлов ... 1070
Класс *PrintJob*. Печать из Flash-фильма ... 1071
Класс *Selection*. Контроль выделения и передачи фокуса ... 1071
Класс *SharedObject*. Хранение информации на стороне клиента ... 1072
Класс *Sound*. Управление звуком ... 1072
Класс *Stage*. Контроль параметров документа ... 1074
Класс *StyleSheet*. Поддержка и использование CSS ... 1074
Класс *System*. Информация о системе и оборудовании ... 1075
Класс *TextField*. Контроль состояния и параметры текстового блока ... 1075
Класс *TextFormat*. Динамическое форматирование текста ... 1079
Класс *TextSnapShot*. Контроль статического текста ... 1080
Класс *Video*. Работа с видеообъектами ... 1081

Приложение 2. Описание компакт-диска ... 1082

Приложение 3. Виртуальные коды клавиш ... 1088

Приложение 4. Горячие клавиши инструкций ActionScript ... 1090

Приложение 5. Интернет-ресурсы, посвященные Flash ... 1092

Предметный указатель ... 1094

Введение

Если бы строители строили здания так же,
как программисты пишут программы,
первый залетевший дятел разрушил бы цивилизацию.

Второй закон Вейберга

Авторам этой книги часто приходится слышать от своих студентов, начинающих обучение по курсу Macromedia Flash, вопрос о том, насколько велика роль программирования в создании Flash-проектов и является ли владение ActionScript необходимым условием для профессиональной реализации специалиста в области Flash-технологии. Как правило, получив на свой вопрос утвердительный ответ, задающий его вскоре сам понимает, насколько широкие возможности открывает ActionScript, и с увлечением (порой смешанным с некоторой опаской) погружается в эту неизведанную область.

Характерной особенностью Flash-технологии является тесная взаимосвязь графического содержания и программной функциональности. Создание Flash-проекта выполняется как бы в двух разных плоскостях.

В одной плоскости лежит работа, связанная с разработкой визуальных образов, рисованием и созданием анимации. Эта часть работы является прерогативой художника и дизайнера.

В другой плоскости лежит работа по проектированию и реализации функциональности и интерактивного наполнения проекта. Эта часть работы входит в обязанности разработчика, программиста.

Несмотря на разный характер этих видов деятельности, они имеют общие точки соприкосновения, поскольку направлены на формирование единого конечного продукта, успех которого в равной степени определяется и визуальным представлением, и интерактивным содержанием. Поэтому профессионал, который обладает навыками работы с графикой и анимацией и владеет таким мощнейшим средством, как ActionScript, способен самостоятельно решать задачи широкого диапазона и тем самым повышает свою конкурентоспособность как Flash-специалист.

Настоящая книга посвящена изучению новейшей версии языка сценариев ActionScript 2.0 и ориентирована на читателей, имеющих опыт работы с Flash MX 2004 (Professional), которые желают освоить возможности программирования в этой среде.

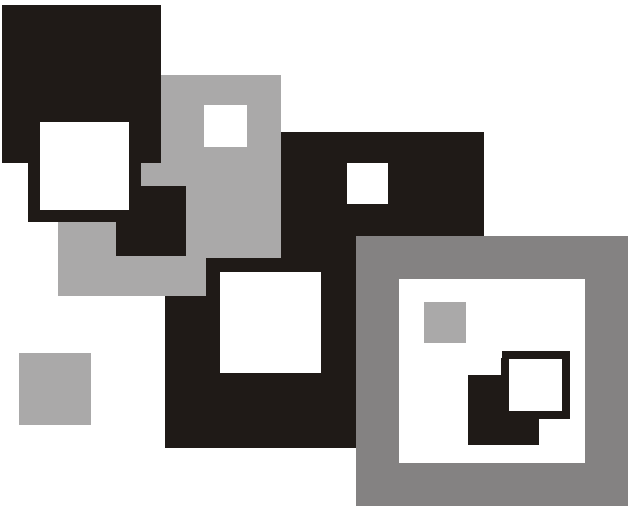
В *главе 1* книги проведен краткий обзор базовых возможностей рабочей среды, связанных с созданием графики, анимации и структурированием документа. Однако в остальных главах при изложении материала предполагается, что читатель уверенно владеет базовыми навыками работы в Flash. Если в процессе чтения читатель столкнется с трудностями, вызванными незнакомыми понятиями или неизученными возможностями рабочей среды Flash, он может обратиться для разъяснения к нашей

предыдущей книге, посвященной основам работы в Flash MX 2004, которая вышла в издательстве "БХВ-Пеьербург" в 2004 году (серия "Самоучитель").

В настоящей книге приводится детальное описание основных возможностей новейшей версии ActionScript, даются подробные описания приемов и методов эффективной работы, пошаговые инструкции, приводятся многочисленные практические примеры. При написании книги авторы учли свой многолетний опыт преподавания Flash и направили все усилия на то, чтобы сделать повествование полным и вместе с тем предельно ясным и доступным не только профессиональным программистам, но и читателям, не имеющим опыта программирования, но желающим освоить возможности ActionScript и приобрести необходимые навыки работы. Материал в книге организован таким образом, чтобы ее можно было использовать одновременно и как обучающее руководство, и как справочное пособие. Последовательность тем сформирована по принципу от общего к частному. Несмотря на то, что многие элементы языка тесно связаны друг с другом, авторы попытались обеспечить максимальную последовательность изложения и по возможности избежать перекрывания тем и "забегания" вперед. В основном при изложении материала используется принцип, в соответствии с которым в начале каждой темы приводятся подробные теоретические сведения, которые затем иллюстрируются конкретными практическими примерами. В книгу включено большое количество практических примеров, демонстрирующих как модели применения элементов языка, так и создание конкретных проектов. Это даст читателю возможность не просто получить формальные сведения о языке, но и приобрести практические навыки работы с ним. Все исходные файлы практических примеров, обладающих самостоятельной ценностью, помещены на сопровождающий книгу компакт-диск. В тексте глав листинги таких примеров обладают заголовком, выведенным на сером фоне. Все программные коды, приведенные в качестве примеров, снабжены подробными комментариями.

Мы надеемся, что данная книга окажется полезной и поможет читателю освоить, возможно, пока неизвестную, но увлекательнейшую область Flash-технологии и уверенно овладеть таким мощным и гибким инструментом, как язык сценариев ActionScript.

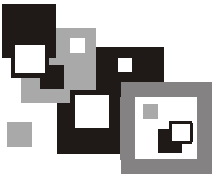
Мы будем признательны вашим отзывам, комментариям и пожеланиям, которые можно направлять по адресу: **book_flash@mail.ru**.



ЧАСТЬ I

ОСНОВЫ ACTIONSCRIPT

Глава 1



Представление о Flash

Если ничто другое не помогает,
прочтите, наконец, инструкцию.

Аксиома Кана

Компоненты программы Flash MX 2004 Professional

Рабочую среду Flash MX 2004 условно можно подразделить на следующие компоненты: векторный редактор, монтажная линейка (Timeline), интегрированная среда разработки (IDE) ActionScript.

- **Векторный редактор** Flash MX 2004 представляет собой инструментарий, позволяющий создавать векторные графические изображения в среде разработки и элементы интерфейса, обеспечивающие удобный доступ к различным инструментам. Среди инструментов векторного редактора можно выделить несколько групп, объединенных общим назначением:
  - *инструменты рисования* — **Oval** (Овал), **Rectangle** (Прямоугольник), **Line** (Линия), **Polystar** (Многоугольник), **Pencil** (Карандаш), **Brush** (Кисть), **Pen** (Перо), — с помощью которых непосредственно создаются графические объекты;
  - *инструменты выделения* — **Selection** (Выделение), **Lasso** (Лассо), **Subselection** (Частичное выделение), — позволяющие организовать доступ к отдельным элементам или совокупностям графических объектов для их дальнейших преобразований;
  - *инструменты работы с цветом* — **Ink Bottle** (Чернильница), **Paint Bucket** (Ведро заливки), **Fill Transform** (Трансформация заливки), **Eyedropper** (Пипетка), — задающие цвет и различные параметры окрашивания графических объектов и их отдельных элементов;
  - *инструменты трансформации* — **Free Transform** (Свободное трансформирование) — многофункциональный инструмент, позволяющий выполнять различные преобразования графических объектов;
  - *инструмент работы с текстом* **Text** (Текст), — позволяющий снабдить проект информационным содержанием и задать параметры его применения;

- *вспомогательные инструменты* — **Zoom** (Линза), **Hand** (Рука), — с помощью которых осуществляется изменение масштаба просмотра содержимого и перемещение по экрану.

Кроме перечисленных групп инструментов векторный редактор включает в себя целый набор средств (панели, команды меню, диалоговые окна), дающих широкие возможности манипуляции графическими объектами.

- **Монтажная линейка** (Timeline) — это средство, позволяющее организовать изменение состояния графических объектов во времени, то есть это инструмент, с помощью которого, собственно, и создается анимация. Монтажная линейка Flash включает в себя также палитру слоев, что позволяет решить множество вопросов, связанных с организацией структуры фильма (в частности, с использованием разных планов: переднего, среднего, заднего) и одновременным выполнением нескольких анимационных процессов.
- **Интегрированная среда разработки ActionScript** — это совокупность инструментария, позволяющего создавать программный код, интегрировать его в структуру документа и осуществлять отладку и тестирование работы программы. Среда разработки ActionScript включает в свой состав следующие элементы.
  - Панель **Actions** — представляет собой редактор ActionScript, который применяется для создания сценариев, включаемых непосредственно в исходный FLA-документ в рабочей среде. Панель **Actions** содержит множество удобных возможностей, таких как всплывающие подсказки, подсветка элементов языка, проверка синтаксиса и анализ ошибок, удобную и разветвленную справочную систему и т. д.
  - Автономный редактор ActionScript — предназначен для создания программного кода, который включается в результирующий SWF-документ на этапе публикации. Автономный редактор применяется при создании пользовательских классов и элементов кода, которые включаются в фильм с помощью директивы `#include`. Интерфейс и функциональные возможности автономного редактора схожи с панелью **Actions**.
  - Панель **Output** и панель **Debugger** — средства отладки программного кода, позволяющие проследить ход выполнения программы и диагностировать ошибки в ее выполнении.

Процесс создания Flash-проекта

В данной книге продукт, созданный в рабочей среде Flash, условимся называть *фильмом*. Это название отражает основную направленность Flash — разработку динамических приложений с использованием анимации. Таким образом, любой проект, будь то анимированный баннер, мультимедиа-презентация или интерактивная заставка, условно будем называть Flash-фильмом.

Технология разработки Flash-фильма включает в себя несколько этапов. В рабочей среде разработчик создает так называемый авторский документ. Этот документ сохраняется с расширением fla. FLA-файл содержит графику, звук информацию об

организации внутренней структуры фильма и применяемых разработчиком технологиях, а также включенные в него сценарии ActionScript. Исходный файл всегда имеет существенно больший объем, чем объем результирующего, конечного файла, и никогда не покидает пределы команды разработчиков, если, конечно, нет особой договоренности с клиентом.

Для того чтобы на основе исходного FLA-документа получить "конечный продукт", предназначенный для дальнейшего распространения среди пользователей, необходимо выполнить процедуру, которая называется *публикация*. В результате публикации Flash транслирует исходный документ в один или сразу несколько результирующих форматов.

В первую очередь, это формат SWF — что называется, "родной" для Flash. Файл обладает всей функциональностью, которой был снабжен исходный FLA-файл, и имеет существенно меньший размер. Это достигается за счет реорганизации данных, осуществляемой на этапе публикации (генерирования конечного файла). В ходе этой реорганизации упрощается структура фильма, неиспользованные элементы библиотеки отбрасываются, растровые изображения и звуки подвергаются компрессии, исходный программный код подвергается компиляции и преобразуется в более простую форму — байткод (bytecode). Кроме того, сам конечный файл также может быть подвергнут компрессии. Для распространения Flash-фильма по Сети его, как правило, необходимо разместить на HTML-странице. SWF-файл связывается с HTML-страницей с помощью специальных тегов, описывающих параметры его позиционирования и воспроизведения, после чего оба файла могут быть размещены в сети Интернет или интранет (intranet).

В результате публикации можно также автоматически создать HTML-документ, на котором будет размещен результирующий SWF-файл. Все настройки, на основании которых будут сгенерированы соответствующие HTML-теги, задаются в настройках публикации с помощью диалоговой формы; таким образом, разработчику не нужно заботиться о создании соответствующего HTML-кода вручную.

При создании проекта, который должен распространяться как независимое приложение (например, электронная презентация или обучающая программа), удобно опубликовать исходный файл как исполняемый файл с расширением exe. Такой файл называется проектор (projector) и включает в себя проигрыватель Flash Player, с помощью которого данный файл можно воспроизвести. Это удобная возможность, когда нет уверенности в том, что в файловой системе у конечного зрителя установлен проигрыватель Flash Player.

Системные требования программы Flash MX 2004 Professional

В табл. 1.1 приведены рекомендуемые системные требования Flash MX 2004 Professional для платформ Windows и Macintosh.

Таблица 1.1. *Системные требования Flash MX 2004*

Системные требования	Платформа	
	Windows	**Macintosh**
Процессор	600 МГц Intel Pentium III	500 МГц PowerPC G3
Операционная система	Windows 98 SE, Windows 2000, Windows XP	Mac OS 10.2.6
Оперативная память	128 Мбайт (желательно 256 Мбайт)	128 Мбайт (желательно 256 Мбайт)
Свободное дисковое пространство	190 Мбайт	190 Мбайт

Для воспроизведения Flash-фильмов на HTML-страницах проигрывателем Flash Player 7 в зависимости от операционной системы необходимо использовать следующие браузеры (табл. 1.2).

Таблица 1.2. *Совместимость браузеров и платформ ОС Windows для Flash Player 7*

Браузер	Платформа			
	Windows 98	**Windows ME**	**Windows 2000**	**Windows XP**
Microsoft Internet Explorer 5.x	+	–	+	–
Microsoft Internet Explorer 5.5	+	+	–	–
Microsoft Internet Explorer 6.0	+	–	–	+
Netscape 4.7	+	+	+	–
Netscape 7.x	+	+	+	+
CompuServe 7	–	–	+	+
Mozilla 1.x	+	+	+	+
AOL 8	+	+	+	+
Opera 7.11	+	+	+	+

Обзор интерфейса

Внешний вид окна документа Flash MX 2004, содержащего основные элементы интерфейса, представлен на рис. 1.1. В качестве основных элементов интерфейса можно выделить следующие составляющие: Главное меню, Стандартная панель, Рабочая область, Вспомогательная область, Полоса редактирования, Служебные панели, Па-

нель инструментов, Инспектор свойств, Монтажная линейка, Контроллер, Библиотека, Палитра истории.

- **Главное меню** включает десять разделов, содержащих все основные команды Flash. Некоторые разделы содержат подменю, позволяющие выбрать необходимое действие.

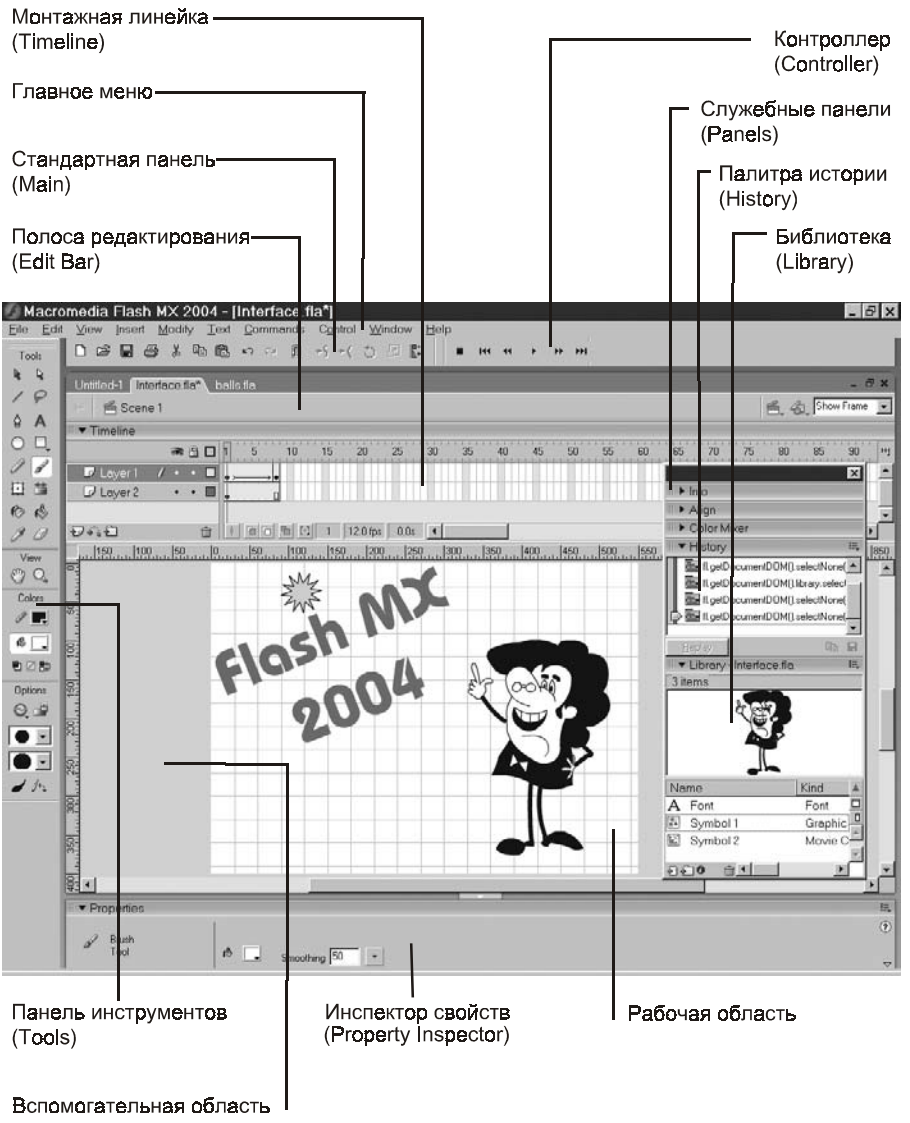


Рис. 1.1. Интерфейс Flash MX 2004

Под главным меню расположены закладки, позволяющие быстро переключаться между открытыми документами Flash. Закладки видны только в случае, если окно документа развернуто.

- **Стандартная панель.** Чтобы включить отображение стандартной панели, нужно выполнить команду меню **Window>Toolbars>Main**. Стандартная панель содержит ряд кнопок, позволяющих выполнить типовые процедуры (создание, открытие, сохранение, печать документа, команды редактирования, отмены произведенных действий и некоторые операции трансформации графических объектов).
- **Рабочая область** (**WorkArea**) по умолчанию представляет собой прямоугольник белого цвета. Все объекты, размещенные в пределах рабочей области, будут видны в конечном фильме. Размер и цвет рабочей области могут быть заданы произвольно.
- **Вспомогательная область** — область серого цвета, расположенная в рабочем окне за пределами рабочей области. Так же, как и на рабочей, на вспомогательной области можно размещать графические объекты и применять к ним все имеющиеся в распоряжении приемы работы. Однако объекты, находящиеся в пределах вспомогательной области, как правило, не видны в конечном фильме (это зависит от параметров публикации). Вспомогательная область используется для размещения элементов, которые не должны быть видны зрителю (например, объекты, содержащие сценарий), или для имитации эффекта маневрирования камерой (наезд, панорамирование и т. д.), когда в пределах рабочей области находится только часть целого изображения.

  В совокупности рабочая область и вспомогательная область называются *сценой* (**Stage**).

Примечание

Во Flash термин "сцена" имеет несколько значений. Причиной этого является то, что два понятия, используемых в официальной терминологии, — Stage и Scene — переводятся на русский язык совершенно одинаково — "сцена". Конечно, можно было бы найти соответствующий эквивалент, например, "подмостки", однако он несколько уводит от основного значения слова. Термин Stage используется, когда имеется в виду физическое размещение объектов в пределах рабочей и вспомогательной областей, а термин Scene применяется, когда речь идет о структурной организации основной монтажной линейки.

- **Полоса редактирования** (**Edit Bar**) служит для ориентации в структуре рабочего проекта. На ней отображается информация о том, в какой среде происходит работа в данный момент (сцена, режим редактирования группы, среда редактирования символа). Кроме того, полоса редактирования содержит элементы управления, позволяющие перемещаться между структурными единицами проекта (сценами), получать доступ к редактированию символов и задавать произвольный масштаб просмотра.
- **Служебные панели.** В рабочий среде Flash MX 2004 имеется целый набор рабочих панелей (Panels), служащих для разнообразных целей дизайна и разработки проектов. Вызов любой служебной панели осуществляется с помощью главного меню **Window**.

Здесь все панели организованы в три группы в соответствии с их назначением.

- **Design Panels** — панели, с помощью которых выполняются задачи по созданию, трансформации и организации графики.
- **Development Panels** — панели, используемые при разработке интерактивности.
- **Other Panels** — остальные панели, позволяющие оптимизировать рабочий процесс создания проекта.

Для того чтобы открыть ту или иную панель, необходимо установить флажок рядом с ее названием. Снятие флажка, соответственно, закрывает панель. Чтобы сразу закрыть все панели, можно использовать команду **Window>Hide Panels** или ее клавиатурный эквивалент <F4>. Повторное выполнение этой команды возвращает все панели на место.

В связи с тем, что число панелей достаточно высоко, и они занимают большую площадь экрана, держать открытыми их все одновременно представляется крайне нецелесообразным. Существует несколько возможностей, позволяющих оптимизировать схему размещения панелей.

Панели могут быть плавающими, парковаться и объединяться в группы.

Для того чтобы открепить панель, необходимо взяться за пиктограмму с изображением двух вертикальных полос в ее левом верхнем углу и перетащить в сторону. Закрыть плавающую панель можно, щелкнув по пиктограмме крестика в ее правом верхнем углу.

Чтобы припарковать панель к границе сцены, нужно взяться за ту же пиктограмму и подтащить панель к одной из границ. Появляющаяся черная рамка укажет на то, что панель будет закреплена. Для включения запрета на прикрепление панелей можно установить флажок **Disable panel docking** меню **Edit>Preferences**.

Для группировки панели необходимо просто припарковать ее к другой панели.

Стандартный вид панели представлен на рис. 1.2. Любая служебная панель может быть развернута и свернута. У развернутой панели черный треугольник в заголовке обращен вправо. Для того чтобы свернуть панель, необходимо щелкнуть по полосе с заголовком в ее верхней части. В свернутом состоянии черный треугольник в левом верхнем углу панели обращен вниз. Повторный щелчок позволяет вновь развернуть панель. Два щелчка по верхней полосе окна (полосе захвата) откреплённой панели позволяют минимизировать панель. Для возврата в первоначальное состояние необходимо повторно дважды щелкнуть по полосе захвата.

Любая служебная панель содержит собственное контекстное меню, содержащее набор команд как специфических только для нее, так и стандартных для всех остальных панелей. Чтобы вызвать контекстное меню панели, нужно щелкнуть по пиктограмме в ее правом верхнем углу. Панель при этом должна быть развернута. Контекстное меню каждой панели содержит три стандартных команды **Help** (Вызов справки), **Maximize Panel** (Развернуть панель), **Close Panel** (Закрыть панель).

Для того чтобы развернуть окно сцены по ширине на весь экран, нужно либо закрыть все припаркованные справа панели, либо щелкнуть по пиктограмме стрелки,

расположенной на середине правой грани сцены. Повторный щелчок вернет сцену в исходное положение. Развернуть окно сцены также можно с помощью команды главного меню **Window>Hide Panels** или ее клавиатурного эквивалента <F4>.

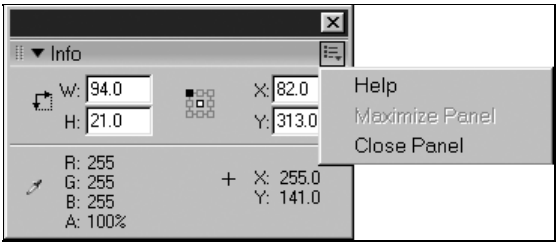


Рис. 1.2. Стандартный вид рабочей панели с ее контекстным меню

Для того чтобы каждый раз не тратить время на организацию панелей, можно сохранить схему их размещения с помощью команды **Window>Save Panel Layout...**. В поле **Name** одноименного диалогового окна нужно ввести имя схемы, таким образом можно создать несколько различных схем на все случаи жизни.

Команда **Window>Panel Sets** позволяет выбрать и применить одну из стандартных или пользовательских схем размещения.

- **Панель инструментов** (**Tools**) — содержит инструментарий, предназначенный для создания графики и работы с ней. Панель инструментов состоит из четырех разделов (рис. 1.3).
  - **Tools** (Инструменты) — в этом разделе содержатся инструменты для создания графики и работы с ней.
  - **View** (Вид) — раздел, содержащий вспомогательные инструменты, используемые для изменения масштаба просмотра и перемещения по экрану.
  - **Colors** (Цвета) — блок управления цветом, с помощью которого могут быть заданы цвета элементов векторного объекта.
  - **Options** (Модификаторы) — в этом разделе содержатся модификаторы (**Modifiers**), позволяющие задать дополнительные параметры или режимы работы соответствующих инструментов.

  Для того чтобы включить (или убрать с экрана) панель инструментов, нужно выполнить команду главного меню **Window>Tools** или воспользоваться ее клавиатурным эквивалентом <Ctrl>+<F2>.
- **Инспектор свойств** (**Property Inspector**) обеспечивает быстрый доступ к различным параметрам настройки и атрибутам выделенного объекта как на сцене, так и на монтажной линейке. С помощью Инспектора свойств можно задавать значения тех или иных параметров объекта или документа, не прибегая к использованию дополнительных меню или панелей (рис. 1.4).

  Для управления открытием или закрытием Инспектора свойств используется команда главного меню **Window>Properties** или сочетание клавиш <Ctrl>+<F3>.

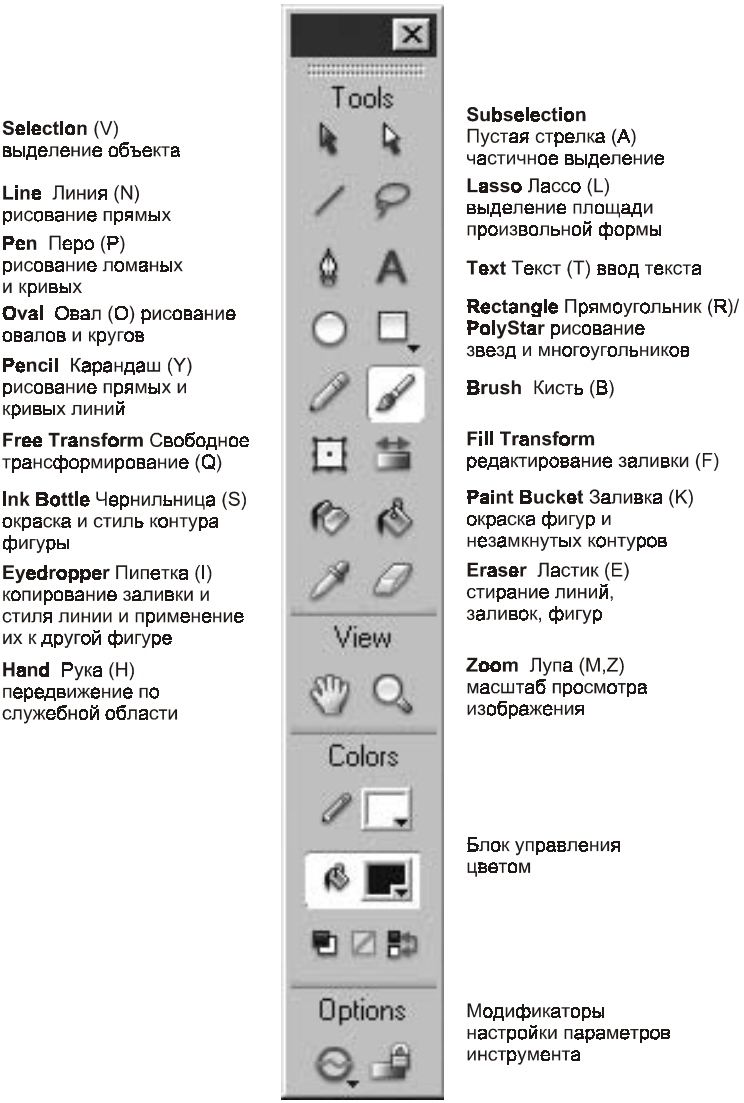


Рис. 1.3. Панель инструментов **Tools**

В левом верхнем углу Инспектора свойств указывается тип выделенного объекта. В зависимости от того, какой именно объект выделен в данный момент, инспектор свойств содержит информацию и настройки текущего документа, контура, векторной формы, группы, растрового изображения, видеоролика, текста, кадра, символа или звукового фрагмента. Если в области выделения одновременно находятся несколько разнородных объектов, то Инспектор свойств указывает на смешанное выделение (**Mixed**).

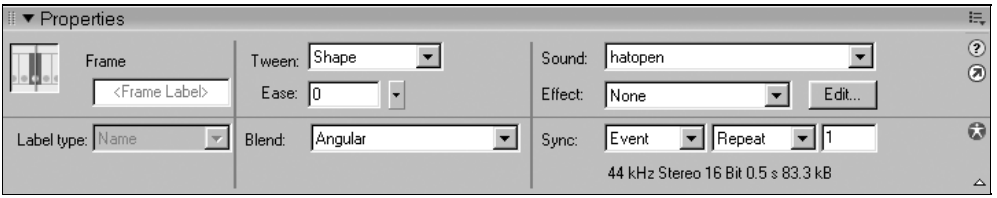


Рис. 1.4. Инспектор свойств при установке свойств кадра

Как и вспомогательные панели, Инспектор свойств может парковаться к краю окна или становиться плавающим; его также можно минимизировать или развернуть. Для выполнения этих операций используются те же приемы, что и для вспомогательных панелей.

Инспектор свойств может иметь более компактный вид или разворачиваться, открывая дополнительные параметры. Для того чтобы развернуть его, необходимо щелкнуть по белому треугольнику в правом нижнем углу, соответственно, эту же пиктограмму можно использовать для приведения к более компактному виду.

- **Монтажная линейка** (**Timeline**) — это инструмент, с помощью которого создается анимация. Монтажная линейка включает в свой состав палитру слоев (**Layers**) и кадры (**Frames**), размещенные в пределах соответствующих слоев. Любой объект, созданный в рабочей среде, размещается в определенном кадре слоя монтажной линейки. При создании нового документа монтажная линейка практически пуста, она содержит один слой и один кадр.

  Для включения (отключения) монтажной линейки используется команда главного меню **Window>Timeline** или сочетание клавиш <Ctrl>+<T>.

  Монтажная линейка может быть плавающей или парковаться к границам сцены. Ее можно минимизировать или развернуть.

- **Контроллер** (**Controller**) — панель, содержащая кнопки, с помощью которых осуществляется управление монтажной линейкой непосредственно в рабочей среде. Контроллер содержит стандартные команды воспроизведения (**Playback**).

  Для включения (отключения) контроллера используется команда главного меню **Window>Toolbars>Controller**. Кроме того, для управления воспроизведением фильма можно использовать главное меню **Control**.

  Контроллер может парковаться или становиться плавающим. Для того чтобы открепить его, нужно взяться курсором за верхнюю грань над элементами управления или два раза щелкнуть по ней и оттащить в сторону. Вернуть панель на место можно, два раза щелкнув по ее заголовку.

- **Библиотека** (**Library**) представляет собой хранилище символов (**Symbols**), а также всех импортированных объектов (растровой графики, видеороликов, звуков). Любой внешний объект, будучи импортированным во Flash, попадает в библиотеку. Библиотека позволяет организовать эти элементы, сгруппировав их с помощью папок. Используя возможности библиотеки, можно задать параметры оптимизации растровых изображений и звуков.

Открыть или закрыть окно библиотеки можно с помощью команды главного меню **Window>Library** или сочетания клавиш <Ctrl>+<L>.

Любой документ Flash содержит собственную библиотеку. Существует возможность импорта объектов из библиотеки другого документа в библиотеку текущего.

- **Палитра истории** (**History**) — это служебная панель, которая позволяет отслеживать все произведенные в ходе работы действия и при необходимости возвращаться на соответствующую стадию. Кроме того, с помощью Палитры истории можно сохранять последовательность действий в виде макросов (**Commands**), которые могут быть применены в текущем или другом документе Flash.

Основы работы с векторным редактором Flash MX 2004

Для представления графической информации в компьютерной среде используются два совершенно разных (с точки зрения технологии реализации) подхода: *векторная* графика и *растровая* графика. Каждый из этих подходов имеет свои характерные особенности и, как следствие, свою область применения.

Понятие о векторной графике

Любое векторное изображение состоит из *контуров (paths)*, в свою очередь представляющих собой совокупность математических объектов, которые могут быть описаны аналитически. Объекты векторной графики очень легко поддаются трансформации, поскольку результат трансформации вычисляется путем изменения значений соответствующих коэффициентов уравнений, описывающих трансформируемый объект.

В частности именно это свойство векторной графики объясняет *независимость изображения от масштаба*. Так, если мы имеем дело с кругом, то для векторной графики неважно, каков диаметр этого круга; изменение масштаба никак не повлияет на внешний вид объекта. Таким образом, у векторного изображения не существует жесткой привязки к размеру. На практике это означает, что внешний вид изображения определяется только параметрами устройства вывода (монитора, принтера и т. д.). Векторное изображение на экране монитора будет выглядеть настолько четко и аккуратно, насколько позволяют его возможности (разрешение, цветовой охват и т. д.).

В векторной графике форма превалирует над цветом, то есть форма *первична*. Векторное изображение состоит из отдельных объектов, каждый из которых является независимым от остальных. Это дает широчайшие возможности для манипуляции объектами, их выделения, перемещения, изменения различных параметров. Процесс работы с векторными объектами чрезвычайно гибок и подвижен. Их можно редактировать без опасения сделать необратимые преобразования.

Векторный формат представления графических данных, как правило, позволяет получать конечные *файлы небольшого размера*. Это связано с тем, что при записи сохраняется не само изображение, а определенный набор данных, описывающих входящие в его состав формы и их атрибуты. Таким образом, для вывода графической информации всякий раз при воспроизведении машина производит расчет, результа-

том которого является формируемое изображение. Размер файла, содержащего векторную графику, главным образом зависит от количества контуров, описывающих изображение. В некоторых случаях, если детализация слишком высока, размер конечного файла может занимать огромный объем.

Векторный формат позволяет *интегрировать текст в изображение,* предоставляя при этом широкие возможности для форматирования. Это позволяет обращаться с текстом, используя все преимущества работы с векторными объектами. При этом качество отображения текста будет оптимальным.

Однако векторная графика имеет и свои ограничения. Сложность принципа реализации векторной графики не позволяет добиться полной реалистичности изображения. Несмотря на то, что современные векторные редакторы располагают широкими возможностями использования различных визуальных эффектов, свойственных скорее растровой графике, в векторных изображениях трудно добиться фотографичности при передаче тонких нюансов.

Другое важное ограничение — отсутствие адекватного механизма ввода графической информации с автоматическим получением векторного изображения. На сегодняшний день совершенного векторного сканера не существует. Разнообразие природных форм безгранично, и вряд ли возможно найти алгоритм, позволяющий точно описывать их с помощью математического аппарата. Единственная возможность автоматизировать процесс ввода — использование автоматической трассировки. Однако трассировка далеко не всегда дает удовлетворительный результат, особенно в случае ее применения к фотореалистичным изображениям.

Flash использует векторный формат представления графики. Это означает, что, находясь в рабочей среде, мы создаем изображения, используя векторную технологию. Однако Flash также содержит достаточно широкие возможности работы с растровыми изображениями (и даже с видеофайлами). При этом растровое изображение обрабатывается как особый объект.

Элементы векторного объекта

В общем случае векторный объект состоит из двух элементов — обводки и заливки. *Обводка (Stroke)* — это линия (контур), которая может быть замкнута и ограничивает внутреннюю область (что и отражает название) или разомкнута (в этом случае никакой внутренней области обводка не содержит). *Заливка (Fill)* — это окрашенная внутренняя область (векторная форма), ограниченная обводкой.

Далее в этой книге термины *контур* и *обводка* являются взаимозаменяемыми.

Разделение на обводку и заливку вызвано тем, что поведение этих элементов и технология работы с ними различны.

Как частный случай, векторный объект может состоять только из одной обводки или, наоборот, представлять собой только векторную форму, то есть заливку. Такие объекты могут являться результатом использования инструментов, создающих только контуры или только формы, если изначально задано отсутствие определенного элемента либо он намеренно удален.

Иерархия графических объектов Flash

В рабочей среде Flash имеет место строгая иерархия графических объектов. Наличие этой иерархии связано с тем, что различные графические элементы Flash-документа могут обладать различной степенью доступа к своему содержимому. Это обуславливает то, что внешне одинаковые графические объекты могут качественно отличаться друг от друга по возможностям обработки. Для того чтобы лучше формализовать иерархию графических объектов и сделать более понятными действующие в ее пределах правила, вводятся понятия *рабочего* и *наложенного* уровней. Любой графический объект, входящий в состав Flash-документа, принадлежит либо к рабочему уровню, либо к наложенному уровню. Не следует путать понятие рабочего и наложенного *уровней* с понятием *слоев*. Слои (layers) — это физические элементы структурной организации документа, в то время как рабочий и наложенный уровни — это более абстрактные понятия, отражающие возможности манипулирования объектами. В пределах каждого слоя всегда имеется свой рабочий и наложенный уровни, поскольку в каждом слое всегда могут существовать графические объекты, обладающие разными возможностями доступа к своему содержимому. Принадлежность к одному или к другому уровню определяет характерные особенности поведения объекта и набор действий, которые могут быть с ним произведены.

Следует иметь в виду, что в пределах одного слоя содержимое рабочего уровня всегда находится под содержимым наложенного уровня. Именно это обстоятельство лежит в основе используемой терминологии.

Рабочий уровень

Характерными особенностями рабочего уровня являются следующие.

- **Возможность изменения базовых свойств объектов** (цвет составляющих их элементов, форма). Эта возможность позволяет выполнять следующие процедуры:
  - выделять объекты целиком и по частям (обводку, заливку или части этих элементов);
  - изменять цвет (и тип заливки) элементов векторного объекта;
  - изменять форму объекта с помощью инструментов **Selection**, **Subselection**, **Pen** или любых операций трансформации.
- Наличие **взаимодействия между объектами**. Взаимодействие объектов между собой выражается в том, что при наложении (пересечении) объекты определенным образом воздействуют друг на друга, что приводит к их изменению. Это взаимодействие может носить различный характер и включает в себя две группы операций:
  - **объединение** — векторные формы без контура (заливки) одного цвета при пересечении объединяются в одну векторную форму. Для объединения объектов необходимо выделить одну фигуру и перетащить ее поверх другой. После снятия выделения они объединятся в одну;
  - **сегментирование** — векторные формы разного цвета и контуры при пересечении сегментируются друг другом. Объект может либо делиться на части другой фигурой, либо с помощью контура от него может быть отрезана часть.

В результате сегментирования образуются новые формы и/или контуры. Контуры также могут быть сегментированы векторными формами.

Примечание

Любой рисующий инструмент создает объекты рабочего уровня. После создания объекта его можно сделать объектом наложенного уровня с помощью операции группирования или конвертирования в символ.

Наложенный уровень

Характерными особенностями наложенного уровня являются:

- отсутствие возможности изменения базовых свойств объектов (цвет составляющих их элементов, форма);
- отсутствие взаимодействия между объектами;
- все объекты наложенного уровня выделяются прямоугольными рамками;
- объекты наложенного уровня всегда располагаются над объектами рабочего уровня (что и отражено в названии).

На наложенном уровне отсутствует непосредственная возможность редактирования формы и цвета элементов векторного объекта. Операции трансформации, доступные на наложенном уровне, также ограничены и включают в себя только: масштабирование, вращение, скос, отражение. На наложенном уровне нельзя выполнять искажение (Distort) и редактирование с помощью огибающих (Envelope), поскольку эти операции изменяют базовые свойства объекта.

Существует целый ряд объектов, относящихся к наложенному уровню. В их число входят: группа, экземпляр символа, текстовый блок, экземпляр растрового изображения, видеоролик.

Объекты наложенного уровня размещаются друг над другом в стеке графических объектов, не взаимодействуя между собой. Для изменения порядка следования объектов наложенного уровня, лежащих в одном слое, можно воспользоваться командами главного меню **Modify>Arrange**.

Группировка объектов

Группа представляет собой контейнер, который может содержать любые графические объекты (в том числе другие группы). Этот контейнер изолирован от других объектов и его содержимое извне не может подвергнуться каким-либо модификациям, связанным с изменением их базовых свойств.

Основные цели группирования объектов:

- изоляция объектов, запрет взаимодействия между объектами рабочего уровня после создания законченного рисунка;
- коллективное преобразование нескольких объектов, входящих в состав группы графических объектов (например, позиционирование сразу нескольких объектов);
- упорядочение сложных структур (вложение групп друг в друга).

Операция **группирования** позволяет переместить объект с рабочего уровня на наложенный уровень и, таким образом, исключить возможность изменения его базовых свойств (цвет и форма). Существует два способа создания группы:

- создать группу из уже имеющихся объектов;
- создать пустую группу и наполнить ее содержимым изнутри.

Для того чтобы сгруппировать один или несколько объектов, необходимо выделить их и выполнить команду главного меню **Modify>Group** (<Ctrl>+<G>). Созданная группа будет выделена прямоугольной рамкой. Для внесения изменений в содержимое группы необходимо войти в режим ее редактирования. Для этого достаточно дважды щелкнуть на группе. О том, что режим редактирования группы активен, свидетельствуют два факта.

- Все остальное содержимое сцены отображается с пониженной степенью насыщенности, указывая на то, что пользователь находится в изолированной среде.
- Над сценой на полосе редактирования появляются надпись **Group** (Группа) и цветная пиктограмма.

Чтобы выйти из режима редактирования группы, можно щелкнуть на надписи **Scene** в полосе редактирования (**Edit bar**), либо использовать стрелку, находящуюся там же, чуть левее, или же просто дважды щелкнуть на любом участке сцены, свободном от объектов данной группы.

Другой способ создания группы предполагает создание пустой группы (пустого контейнера) и наполнение ее графическим содержимым изнутри. Для того чтобы создать пустую группу, необходимо убедиться в отсутствии на сцене выделенных объектов и выполнить все ту же команду главного меню **Modify>Group**. В результате этого будет создана новая группа и активизирован режим ее редактирования. Для завершения создания группы необходимо наполнить ее графическим содержимым. При этом можно использовать любые рисующие инструменты и инструменты трансформирования, поскольку в режиме редактирования группы ее содержимое может обрабатываться с применением всех операций, доступных на рабочем уровне. Следует иметь в виду, что Flash автоматически удаляет все группы, лишенные графического содержимого, поэтому, если пустая группа не будет наполнена содержимым, то при выходе из режима ее редактирования она будет автоматически удалена.

Создание группы из пустой группы применяется, например, в ситуации, когда необходимо нарисовать и разместить объект поверх уже имеющихся объектов рабочего уровня. Если рисовать его непосредственно сверху, то объекты провзаимодействуют друг с другом. Создание пустой группы исключает возможность взаимодействия, в то же время позволяя в процессе рисования видеть все остальное содержимое сцены.

При организации групп часто применяется иерархический подход, при котором группы путем группировки объединяются друг с другом и с другими объектами рабочего и наложенного уровней. В результате образуются группы, содержащие другие группы, называемые **вложенными** (nesting groups). Количество уровней вложенности групп не ограничено, то есть группа может содержать любое число вложенных групп. Для того чтобы изменить содержимое вложенной группы, необходимо после-

довательно входить в режим редактирования групп более высокого уровня, при этом число пиктограмм с надписью **Group**, появляющихся на полосе редактирования, будет соответствовать текущему уровню вложенности.

В табл. 1.3 обобщены сведения об иерархии графических объектов Flash.

Таблица 1.3. *Иерархия графических объектов Flash*

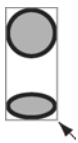
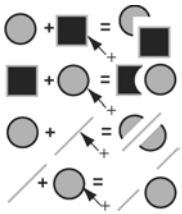
Операция	Рабочий уровень	Наложенный уровень
Тип объектов	Векторные формы и контуры, созданные инструментами рисования	Группа, растровое изображение, видеоролик, текстовый блок, экземпляр символа
Выделение объекта инструментом Selection	 Выделен сегмент контура	Выделена группа (выделение щелчком или рамкой) 
	 Выделен весь контур (2 щелчка по контуру)	
	 Выделена заливка	
	 Выделены контур и заливка (2 щелчка по заливке)	
	 Выделение рамкой	
Редактирование объектов	Изменение формы заливки и контура инструментом **Selection** (объект предварительно *не выделен*)	Режим редактирования группы (2 щелчка по группе)
	 Исходный объект	
	 Перемещение вершины объекта	Разгруппировать на составляющие элементы **Modify>Ungroup** (<Ctrl>+<Shift>+<G>)
	 Изменение кривизны отрезка	Разбить группу на части **Modify>Break Apart** (<Ctrl>+<B>)
	 Добавить вершину (потянуть с <Ctrl> за отрезок	

Таблица 1.3 *(окончание)*

Операция	Рабочий уровень	Наложенный уровень
Взаимодействие объектов	**Объединение** объектов одного цвета без контура 	Объекты не взаимодействуют между собой
	Сегментирование (деление на части) объектов разного цвета и контуров 	
Трансформирование объектов	Возможны все виды трансформирования выделенного объекта, его части или нескольких объектов: перемещение, масштабирование, зеркальное отражение, поворот, скос, искажение, редактирование с помощью огибающих	Возможны только виды трансформации, не изменяющие базовых свойств объектов: перемещение, масштабирование, зеркальное отражение, поворот, скос

Операции над объектами и трансформация объектов

К их числу относятся:

- Перемещение (Move).
- Копирование (Copy).
- Вырезание в буфер обмена.
- Вставка из буфера обмена (Paste).
- Удаление (Delete).
- Изменение цвета и параметров элементов объектов.
- Операции трансформации:
  - Масштабирование (Scale).
  - Зеркальное отражение (Flip).
  - Вращение (Rotate).
  - Скос (Skew).
  - Искажение (Distort).
  - Редактирование с помощью огибающих (Envelope).

Как и многие другие операции Flash, операции трансформации могут выполняться различными способами. Информация о способах преобразования объектов обобщена в табл. 1.4.

Таблица 1.4. *Способы преобразования объектов*

Операция/ Способ	**Инструмент (ручное преобразование)**	**Палитра (точное преобразование)**	**Команда меню**
Move перемещение	Инструмент **Selection** <V> Управляющие клавиши ("стрелки") на клавиатуре, с <Shift> шаг 10 пикселов Инструмент **Free Transform**	Палитра **Info**, ввод координат *x*, *y* для левого верхнего угла или центральной точки объекта	Палитра **Align** выравнивания и распределения объектов **Window>Design Panels>Align...**, <Ctrl>+<K>
Copy копирование	Инструмент **Selection** + + <Alt> или <Ctrl> <Ctrl>+<D> Через буфер обмена Контекстное меню	Палитра **Transform**, кнопка **Copy**	**Edit>Copy**, <Ctrl>+<C> **Edit>Duplicate** **Edit>Paste in Center** вставить в центр **Edit>Paste in Place**
Scale масштабирование	Инструмент **Free Transform**, модификатор **Scale**	Палитра **Info**, ввод *w*, *h* (ширины и высоты), палитра **Transform** ввод % *w*, *h* по высоте и ширине, пропорциональное (Constrain) и непропорциональное масштабирование	**Modify>Transform> Scale** Палитра **Align Window>Design Panels >Align...**, <Ctrl>+<K> выровнять по размеру **Match Size**
Rotate поворот	Инструмент **Free Transform**, модификатор **Rotate**, перенос диагональных маркеров, с <Shift> на углы, кратные 45	Палитра **Transform**, ввод угла поворота в градусах, по часовой стрелке — положительное число, против часовой — отрицательное	**Modify>Transform> Rotate 90 CW** поворот на 90 влево **Modify>Transform> Rotate 90 CCW** поворот на 90 против часовой стрелки
Skew наклон/ скос	Инструмент **Free Transform**, модификатор **Rotate and Skew**, перенос боковых и вертикальных маркеров	Палитра **Transform**, ввод углов скоса *w*, *h* по горизонтали и вертикали	**Modify>Transform> Rotate and Skew**

Таблица 1.4 (окончание)

Операция/ Способ	Инструмент (ручное преобразование)	Палитра (точное преобразование)	Команда меню
Distort искажение	Инструмент **Free Transform**, модификатор **Distort**	нет	**Modify>Transform> Distort**
Envelope оболочка	Инструмент **Free Transform**, модификатор **Envelope**	нет	**Modify>Transform> Envelope**
Flip зеркальное отображение	Инструмент **Free Transform**, модификатор **Scale** перенос боковых и диагональных маркеров через объект	нет	**Modify>Transform> Flip Horizontal** **Modify>Transform> Flip Vertical**
Delete удаление	Выделить объект и нажать клавишу <Delete>	нет	**Edit>Clear, <Backspace>**

Настройка глобальных параметров фильма

К глобальным параметрам фильма относятся:

- размер рабочей области;
- цвет фона;
- скорость воспроизведения монтажной линейки;
- единицы измерения рабочей среды.

Для настройки перечисленных параметров используется команда меню **Modify>Document...** или сочетание клавиш <Ctrl>+<J>. Выполнение данной команды приводит к появлению окна **Document Properties**, представленного на рис. 1.5.

Задать глобальные параметры фильма также можно с помощью инспектора свойств, содержащего аналогичные настройки при отсутствии выделенных объектов (графики или кадров). Нажатие клавиши **Size** (Размер) инспектора свойств также приводит к появлению окна **Document Properties**. Окно содержит следующие настройки:

- **Dimensions** — размеры рабочей области по ширине (width) и высоте (height), задаются в текущих единицах измерения. Диапазон допустимых значений в пикселях — от 1 до 2880;
- **Match** — установка фиксированного размера рабочей области:
  - **Printer** — автоматическая установка размеров рабочий области равными размеру области печати принтера;

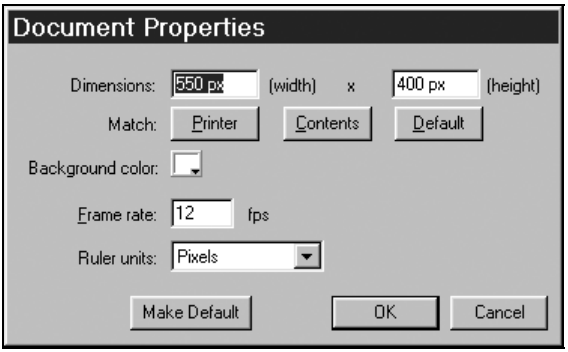


Рис. 1.5. Окно настройки глобальных параметров фильма

- **Contents** — размер рабочей области устанавливается таким образом, что все объекты, расположенные в положительном квадранте (то есть правее и ниже левого верхнего угла рабочей области), оказываются в пределах рабочей области. Для того чтобы точно подогнать размер рабочей области к размеру размещенной на ней композиции, нужно совместить левый верхний угол рамки, ограничивающей изображения, с левым верхним углом рабочей области *и нажать клавишу* **Contents**;
- **Default** — задает значения глобальных параметров документа, используемые по умолчанию;

- **Background color** — выбор цвета фона из текущего каталога цветов. В качестве цвета фона может быть использован *только* сплошной цвет;
- **Frame rate** — скорость воспроизведения монтажной линейки. Измеряется в единицах "кадр в секунду" (frames per second — fps) и задается в диапазоне от 0,01 до 120 кадров в секунду. Наиболее часто используемые значения — 12 или 24 fps;
- **Ruler units** — единицы измерения рабочей среды. Вся информация о размерах отображается в этих единицах;
- **Make default** — сделать текущие настройки настройками, используемыми по умолчанию.

Работа с цветом

В рабочей среде Flash можно синтезировать любой из более чем 16 миллионов оттенков, доступных в цветовой модели RGB. Кроме того, на стадии синтеза цвета можно задать ему определенную степень прозрачности, что позволит получать интересные визуальные эффекты, связанные с перекрыванием частично прозрачных объектов. Имеется возможность загрузки внешних цветовых палитр и создания собственных, которые могут храниться в отдельном файле. При работе над проектом, состоящим из нескольких документов, они могут использовать общие цветовые палитры, что позволяет обеспечить единое цветовое решение различных разделов.

Во Flash существует три типа цветового заполнения:

- сплошной цвет (Solid);
- градиент (Gradient);
- растровая заливка (Bitmap).

Как известно, обводки могут быть окрашены только сплошным цветом, в то время как заливки могут содержать любой из вышеперечисленных типов цветового заполнения.

Для синтеза цвета в рабочей среде Flash используется панель **Color Mixer** (Синтез цвета). Для вызова этой панели нужно выполнить команду главного меню **Window>Design Panels>Color Mixer**. Цвет может быть синтезирован на основе модели RGB или HSB путем задания его точных координат либо выбран "на глаз" с помощью цветового поля, расположенного в нижней части панели. Кроме того, синтезировать сплошной цвет можно "подобрав" его с любой точки экрана. Для того чтобы "подобрать" цвет экрана, необходимо щелкнуть по цветовому образцу в любом блоке управления цветом (в панели инструментов, в панели **Properties** или в панели **Color Mixer**), и когда курсор примет вид пипетки, не отпуская кнопки мыши, увести его в нужную точку. При этом цветовой образец будет фиксировать цвет, соответствующий цвету пиксела экрана, над которым в данный момент находится курсор.

Поле **Alpha** панели **Color Mixer** позволяет задать степень прозрачности для цвета. Прозрачность задается в процентах в диапазоне от 0 до 100. При значении **Alpha** равном нулю, цвет полностью прозрачен, при значении **Alpha**, равном 100, цвет полностью непрозрачен. Все промежуточные значения позволяют задать частичную степень прозрачности.

Градиентная заливка (**Gradient**) представляет собой плавную растяжку (переход) цветов. Основные цвета, входящие в градиент, называются *цветовыми порогами* градиента. Во Flash градиент может содержать не более восьми цветовых порогов. Градиент может быть *линейным* (**Linear**) или *радиальным* (**Radial**). В линейном градиенте переход цветов осуществляется вдоль прямолинейной оси, совпадающей с направлением градиента. Цветовые пороги радиального градиента представляют собой концентрические окружности с общим центром, диаметр которых увеличивается по мере удаления от центра. Цветовой переход в этом случае осуществляется вдоль радиусов.

Синтез градиента, как и синтез сплошного цвета, выполняется в палитре **Color Mixer**. Для его создания необходимо в выпадающем списке, расположенном в центре панели, выбрать требуемый тип градиента, в результате чего в панели будет выведена полоса редактирования градиента. Цветные маркеры, расположенные под этой полосой, соответствуют цветовым порогам градиента. Для добавления нового цветового порога необходимо создать новый маркер. Для этого достаточно щелкнуть под полосой редактирования. Чтобы удалить цветовой порог из градиента, необходимо увести соответствующий ему маркер вниз за пределы области редактирования.

Для редактирования градиентной заливки используется инструмент **Fill Transform**. Данные об основных операциях с цветом представлены в табл. 1.5.

Таблица 1.5. *Операции с цветом*

Операция	Способ
Синтез цвета	Панель **Color Mixer (Window>Design Panels>Color Mixer)**
Хранение цвета	Панель **Color Swatches (Window>Design Panels>Color Swatches)**
Назначение цвета по умолчанию	Кнопка на панели **Color Mixer**; блок управления цветом **Color** панели **Tools** — черная обводка и белая заливка
Назначение цвета заливке	Инструмент **Paint Bucket**; панели **Color Mixer**; **Color Swatches**; **Properties**; блок управления цветом **Color**, системная палитра **Color**
Назначение цвета обводке	Инструмент **Ink Bottle**; панели **Color Mixer**; **Color Swatches**; **Properties**; блок управления цветом **Color**; системная палитра **Color**. Обводка не может быть окрашена в градиентную или растровую заливку
Подбор образца цвета	Инструмент **Eyedropper**

Работа с растровой графикой

Рабочая среда Flash предоставляет достаточно широкие возможности по работе с растровой графикой. Будучи импортированным в рабочую среду, растровое изображение интегрируется в содержимое документа и может быть подвергнуто различным преобразованиям.

Растровые изображения, как правило, занимают больший объем памяти, чем векторные. В связи с этим растровое изображение, включенное в фильм Flash, должно быть обязательно оптимизировано. Эта оптимизация должна включать в себя два этапа.

- Оптимизация в специализированном редакторе растровой графики. Необходимо установить разрешение изображения, не превышающее разрешение экрана монитора (72 dpi или 96 dpi). Размер изображения должен быть уменьшен до величины, которая необходима при использовании изображения в Flash-проекте. Тоновая и цветовая коррекции также выполняются на этапе, предшествующем импорту.
- Оптимизация средствами Flash непосредственно в рабочей среде программы, включающая выбор соответствующего алгоритма сжатия.

Flash MX 2004 позволяет импортировать в рабочую среду растровые изображения большинства наиболее широко используемых форматов. К их числу относятся сле-

дующие (в скобках указаны расширения файлов): GIF (gif); JPEG (jpg); PNG (png); BMP (bmp); TGA (tga); TIFF (tif, tiff) и некоторые другие.

Импортировать файл можно следующими способами:

- с помощью команды главного меню Flash **File>Import**. Команда **Import** предлагает следующий выбор:
  - **Import to Stage** (Импортировать на сцену) — импортированное изображение помещается на сцену, при этом оно автоматически заносится в библиотеку;
  - **Import to Library** (Импортировать в библиотеку) — импортированное изображение помещается в библиотеку без размещения на сцене;
- через **буфер обмена** — во внешнем растровом редакторе векторный объект копируется в буфер и затем вставляется в рабочей среде Flash с помощью команды меню **Edit>Paste** (<Ctrl>+<V>).

В результате импорта растрового изображения оно попадает в библиотеку документа. Для того чтобы включить изображение в фильм, необходимо создать его экземпляр. В документе можно разместить неограниченное количество экземпляров одного и того же изображения, причем на конечный объем документа влияние будет оказывать только объем, занимаемый исходным файлом растрового изображения, расположенным в библиотеке. Экземпляр растрового изображения является объектом наложенного уровня.

Для того чтобы преобразовать экземпляр растрового изображения в объект рабочего уровня, его необходимо разбить с помощью команды **Modify>Break Apart**. Для того чтобы использовать растровое изображение в качестве цветового заполнения векторных форм, в выпадающем списке панели **Color Mixer** следует выбрать значение **Bitmap**. В результате этого действия в панели **Color Mixer** будет выведен каталог, содержащий уменьшенные копии всех растровых изображений, находящихся в библиотеке. Выбранное здесь изображение будет использоваться в качестве растровой заливки, для применения которой можно воспользоваться стандартным инструментарием рисования.

В табл. 1.6 обобщены основные возможности работы с растровыми изображениями.

Таблица 1.6. *Операции с растровой графикой*

Операция		Способ
Наложенный уровень	Импорт	Команда **File>Import...**; выбрать один файл или последовательность
	Оптимизация	Дважды щелкнуть на пиктограмме растрового изображения в окне библиотеки
	Копирование	**1**. Вытащить экземпляр изображения из библиотеки на сцену. **2**. **Edit>Duplicate**. **3**. <Ctrl>+<D>. **4**. Тащить с <Ctrl> или команда <Alt>. **5**. Через буфер обмена
	Удаление	**1**. Команда **Edit>Clear**. **2**. <Delete>. **3**. Удалить из библиотеки растровое изображение
	Трансформация	Инструмент **Free Transform** (масштаб, поворот, скос, зеркальное отражение)

Таблица 1.6 *(окончание)*

Операция		Способ
Наложенный уровень	Трассировка	Команда **Modify>Bitmap>Trace Bitmap...** выполняет автоматическую трассировку изображения
	Замена в фильме	Команда **Modify>Bitmap>Swap Bitmap...**; замена расположенного на сцене экземпляра одного растрового изображения на экземпляр другого
	Анимация	Тип анимации движения **Motion Tween**
	Разбиение (переход к рабочему уровню)	Команда **Modify>Break Apart**; <Ctrl>+<B>
Рабочий уровень	Выделение	*Целиком* — щелчок инструментом **Selection**; *частично* — рамкой инструмента **Selection**; *часть произвольной формы* — инструментом **Lasso**; *области одного цвета* — модификатор **Magic Wand** инструмента **Lasso**
	Изменение формы	Инструменты **Selection**; **Free Transform** (модификаторы **Envelope, Distort**); **Subselection**; **Pen; Eraser**
	Заливка растром	Палитра **Color Mixer**; инструмент **Eyedropper**; инструмент **Paint Bucket** (для заливки всех форм одним растром включить режим **Lock Fill**); инструмент **Brush**
	Редактирование заливки	Инструмент **Fill Transform**
	Анимация	Тип анимации формы **Shape Tween**
	Группировка	Команда **Modify>Group**; <Ctrl>+<G> переход к наложенному уровню

Структура монтажной линейки

Монтажная линейка состоит из двух частей, совмещая в себе палитру слоев, позволяющих структурировать документ, и последовательности кадров, обеспечивающих изменение содержимого сцены во времени.

Слои (layers) можно сравнить с прозрачными листами пленки, на которых могут быть размещены любые объекты рабочей среды. Слои предоставляют возможность изолировать объекты друг от друга, исключая взаимодействие между ними и формируя структуру сцены. Слои Flash прозрачны и размещаются друг над другом в стеке графических объектов. Сквозь области вышележащих слоев, свободных от объектов, можно видеть содержимое слоев, расположенных под ними. Каждый слой содержит свой рабочий и наложенный уровни, в пределах которых действуют все правила,

свойственные иерархии графических объектов Flash. Документ Flash может содержать практически неограниченное число слоев, причем их количество не влияет на размер конечного файла SWF.

Кадры (frames) — это структурные элементы документа, которые представляют собой место фактического размещения любого объекта, включенного в фильм в рабочей среде и в процессе его воспроизведения. Кадры размещаются на монтажной линейке последовательно друг за другом и могут содержать графические объекты, звуковые файлы или сценарии ActionScript. Число кадров в документе практически неограниченно. Следует различать два понятия: *кадр фильма* и *кадр монтажной линейки*.

Кадр фильма — это структурный элемент, объединяющий все объекты сцены, размещенные на различных слоях в определенной временной позиции. Каждый кадр сцены имеет уникальный номер, соответствующий его временной позиции. Номер кадра указан в шкале, расположенной в верхней части монтажной линейки. Здесь же находится указатель текущего кадра, представляющий собой розовый прямоугольник. Перемещая его вдоль монтажной линейки, можно переходить с одного кадра на другой. Длительность экспонирования (просмотра) каждого кадра определяется скоростью воспроизведения монтажной линейки. Так, если скорость воспроизведения монтажной линейки составляет 12 кадров в секунду (frames per second — fps), то при воспроизведении каждый кадр фильма будет виден в течение 1/12 (≈0,08) секунды. Десятисекундный фильм при данной скорости должен содержать 120 кадров. Содержимое каждого кадра фильма может быть распределено по различным слоям.

Кадр монтажной линейки — элемент кадра фильма, расположенный в определенной временной позиции в пределах одного слоя. Любому такому кадру на монтажной линейке соответствует единичная ячейка. На каждом слое может находиться практически неограниченное число кадров монтажной линейки. Кадры монтажной линейки, имеющие один и тот же номер (то есть расположенные в одной и той же временной позиции), формируют кадр фильма. Таким образом, кадр фильма представляет собой срез всех слоев в определенной временной позиции, определяемый скоростью воспроизведения монтажной линейки.

Работа со слоями

Документ Flash может содержать слои различных типов. Каждый тип слоя обладает своими характерными особенностями и применяется для выполнения определенных задач. В палитре слоев каждый тип слоя имеет свою пиктограмму, таким образом, по внешнему виду слоя можно определить его принадлежность к тому или иному типу. Для задания типа слоя применяется диалоговое окно **Layer Properties**, которое можно открыть с помощью команды **Properties...** контекстного меню слоя или двойного щелчка на его пиктограмме.

Новый документ содержит только один слой. Находясь в рабочей среде Flash, можно создавать новые слои, выполнять с ними различные манипуляции и использовать разные режимы отображения и редактирования слоев. Таблица 1.7 обобщает основные возможности работы со слоями.

Таблица 1.7. *Операции со слоями*

Операция	Способ
Создать слой	1. Команда **Insert>Timeline>Layer**. 2. Команда **Insert Layer** контекстного меню слоя. 3. Щелкнуть на пиктограмме  **Insert Layer** в нижней части палитры слоев. 4. Команда **Modify>Timeline>Distribute to Layers** — автоматическое размещение выделенных объектов по слоям
Переименовать	1. Команда **Modify>Timeline>Layer Properties**. 2. Дважды щелкнуть на имени слоя
Выделить слой	1. Щелкнуть на слое в палитре слоев (все кадры слоя выделяются), с <Shift> можно выделить диапазон слоев, с <Ctrl> — несколько слоев вразбивку. 2. Выделить объект на слое. 3. Щелкнуть мышью на кадре
Копировать слой	Команда **Edit>Timeline>Copy Frame**s, затем создать новый слой, вставить кадры командой **Edit>Timeline>Paste Frames**
Изменить порядок расположения слоев	Перетащить слой за его пиктограмму в палитре слоев
Задать свойства слоя	1. Команда **Modify>Timeline>Layer Properties**. 2. Дважды щелкнуть на пиктограмме слоя. 3. Команда **Properties...** контекстного меню слоя. В свойствах задаются тип слоя, имя, режимы отображения и редактирования
Удалить слой	1. Выделить слой и щелкнуть на пиктограмме  мусорной корзины палитры слоев. 2. Команда **Delete Layer** контекстного меню слоя

В табл. 1.8 представлены пиктограммы всех типов слоев и обобщены их основные особенности.

Таблица 1.8. *Типы слоев и их назначение*

Тип и пиктограмма слоя	Назначение
Нормальный **Normal Layer** 	Может содержать любые объекты (статичную графику, анимацию, звук, сценарии, интерактивные элементы), которые будут транслированы в конечный фильм

Таблица 1.8 *(окончание)*

Тип и пиктограмма слоя	Назначение
Направляющий **Guide Layer**	Используется для создания вспомогательных элементов, произвольных линий разметки, макетов, шаблонов, позволяющих позиционировать или создавать объекты на других слоях; содержимое не публикуется в конечный фильм
Слой пути **Motion Guide**	Содержит траекторию движения анимированного объекта (только непрерывный, незамкнутый контур), содержимое не видно в конечном фильме, связан с одним или несколькими ведомыми слоями
Ведомый слой **Guided**	Сгруппирован со слоем пути, содержит автоматическую анимацию движения (Motion Tween), реализующую линейное перемещение объекта из начального положения в конечное. Ведомых слоев может быть несколько
Слой-маска **Mask Layer**	Содержит трафарет (векторная форма, текст, окрашенные сплошным цветом), через который видны нижние подгруппированные маскированные слои. Может содержать анимацию
Маскируемый слой **Masked**	Содержимое маскируемого слоя видно только сквозь векторные формы (области), расположенные на слое-маске. Может содержать анимацию. Маскируемых слоев может быть несколько
Слой-папка **Folder**	Служебный слой, используется для компактной организации слоев в палитре. Может содержать вложенные слои-папки

Работа с кадрами

При создании нового документа Flash его монтажная линейка содержит только один кадр фильма. Остальная область монтажной линейки представляет собой пустые ячейки, в которые могут быть помещены кадры. После добавления новых кадров фильма на монтажную линейку при воспроизведении воспроизводящая головка будет автоматически перемещаться из кадра в кадр. При этом размещенные в них изображения будут сменять друг друга на экране.

Типы кадров

Монтажная линейка может содержать три основных типа кадров, имеющих различное назначение и область применения. К их числу относятся:

- ключевые кадры — **Keyframes**;
- простые (обычные) кадры — **Frames**;
- кадры трансформации — **Tweened frames**.

В табл. 1.9 обобщены сведения о типах и характеристиках кадров.

Таблица 1.9. *Типы и характеристики кадров*

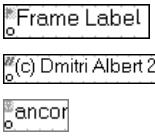
Тип	Вид кадра на монтажной линейке	Назначение и содержание кадра	Способ создания
Ключевой кадр **Keyframe**		Используется для любого изменения содержимого сцены — добавление, удаление объектов из кадра, изменения в анимации	Команда **Insert>Timeline> Keyframe**; <F6>, контекстное меню кадра
Пустой ключевой кадр **Blank Keyframe**		Ключевой кадр без графического содержимого	Команда **Insert>Timeline> Blank Keyframe**; <F7>, контекстное меню кадра
Простой заполненный кадр **Frame**	Светло-серый прямоугольник	Следует после ключевого кадра, отображает содержимое предшествующего ключевого кадра; используется для увеличения времени его экспонирования (просмотра)	Команда **Insert>Timeline> Frame**; <F5>, контекстное меню кадра
Простой пустой кадр **Frame**	Белый прямоугольник	Следует после пустого ключевого кадра	Команда **Edit>Clear** — очистка содержимого предшествующего ключевого кадра
Простой концевой кадр		Простой кадр, используется для регулирования длины диапазона простых кадров	Последний в диапазоне простых кадров
Ключевой кадр, содержащий код AS		Содержит код ActionScript, простые пустые кадры после него также содержат данный сценарий	Команда **Window> Development Panels>Actions**, <F9>
Метка **label**, комментарий **comment**, или якорь **anchor**	Красный флажок с подписью, текстовая строка комментария или значок якоря с названием в заполненном или пустом кадре 	Содержит метку (имя) кадра, используемую для обращения к нему, комментарий автора фильма или якорь, применяемый при перемещении между разделами конечного проекта с помощью кнопок браузера "Вперед", "Назад"	Инспектор свойств, поле **Frame** тип **Label Type**

Таблица 1.9 *(окончание)*

Тип	**Вид кадра на монтажной линейке**	**Назначение и содержание кадра**	**Способ создания**
Кадры трансформации **Motion tween** анимация движения	Ключевые кадры отмечены черной точкой, кадры трансформации — черная стрелка на синем фоне 	Кадры трансформации, единая последовательность кадров. Два ключевых кадра в начале и в конце диапазона анимации задают начальное и конечное состояние объекта, между ними промежуточные кадры, отображающие трансформацию объекта	1. Инспектор свойств, список **Tween** тип **Motion**; 2. Команда **Insert>Timeline> Create Motion Tween** — создание анимации движения
Кадры трансформации **Shape tween** автоматического изменения формы	Ключевые кадры отмечены черной точкой, кадры трансформации — черная стрелка на зеленом фоне 		Инспектор свойств, список **Tween**, тип **Shape**
Ошибка в автоматической анимации	Пунктирная линия 		

Описание операций с кадрами и способов их выполнения представлено в табл. 1.10.

Таблица 1.10. *Операции с кадрами монтажной линейки*

Операция	**Способ**
Создание заполненного ключевого кадра	Команда **Insert>Timeline>Keyframe**, <F6> (автоматически в новый ключевой копируется содержимое предыдущего ключевого кадра); команда контекстного меню кадра **Insert Keyframe**
Создание пустого ключевого кадра	Команда **Insert>Timeline>Blank Keyframe**, <F7>; команда контекстного меню кадра **Insert Blank Keyframe**
Создание простого кадра	Команда **Insert>Timeline>Frame**, <F5>; команда контекстного меню кадра **Insert Blank Keyframe**; если ключевые кадры создаются с определенным интервалом, то промежутки между ними автоматически заполняются простыми кадрами

Таблица 1.10 *(продолжение)*

Операция		Способ
Выделение в покадровом режиме	кадра	Щелчок на кадре монтажной линейки; с <Ctrl> — выделение кадров вразбивку
	диапазона кадров	Выделить первый кадр диапазона, затем с <Shift> щелкнуть на последнем кадре диапазона; 2 щелчка на любом кадре диапазона
	всех кадров	**Edit>Timeline>Select All Frames**, <Ctrl>+<Alt>+<A>, команда контекстного меню кадра **Select All Frames**; щелчок на слое выделяет все кадры слоя
Выделение в интервальном режиме (в меню **Edit>Preferences** флажок **Span based selection)**	кадра	Щелчок на кадре монтажной линейки с <Ctrl>
	диапазона кадров	Щелчок на любом кадре диапазона; сочетание клавиш <Ctrl> и <Shift> позволяет выделить произвольный диапазон последовательно расположенных кадров
	всех кадров	**Edit>Timeline>Select All Frames**, <Ctrl>+<Alt>+<A>, команда контекстного меню кадра **Select All Frames**; щелчок на слое выделяет все кадры слоя
Перемещение кадра		Выделить кадр или несколько кадров и перетащить мышью в требуемое положение
Изменение длины диапазона кадров		В **покадровом** режиме с <Ctrl> переместить концевой или ключевой кадр, изменяя соответствующий диапазон; в **интервальном** режиме просто переместить концевой или ключевой кадр
Копирование кадров		Выделить кадры и выполнить команду **Edit>Timeline>Copy Frames**, <Ctrl>+<Alt>+<C> или команду контекстного меню кадра **Copy Frames**
Вырезание кадров		Выделить кадры и выполнить команду **Edit>Timeline>Cut Frames**, <Ctrl>+<Alt>+<X> или команду контекстного меню кадра **Cut Frames**
Вставка скопированных кадров		Выделить кадр, несколько кадров или пустую ячейку монтажной линейки, соответствующих началу вставляемого диапазона, и выполнить команду меню **Edit>Timeline>Paste Frames**, <Ctrl>+<Alt>+<V> или команду контекстного меню кадра **Paste Frames**
Преобразование ключевого кадра в простой		Выделить ключевые кадры и выполнить команду **Modify>Timeline>Clear Keyframe**, <Shift>+<F6> или команду контекстного меню кадра **Clear Keyframe**
Преобразование простого кадра в заполненный ключевой		Выделить простые кадры и выполнить команду **Modify>Timeline>Convert to Keyframes**, <F6> или команду контекстного меню кадра **Convert to Keyframes**

Таблица 1.10 *(окончание)*

Операция	Способ
Преобразование простого кадра в пустой ключевой	Выделить простые кадры и выполнить команду **Modify>Timeline> Convert to Blank Keyframes**, <F7> или команду контекстного меню кадра **Convert to Blank Keyframes**
Изменение направления анимации	Выделить последовательность кадров, включающую несколько ключевых кадров, и выполнить команду **Modify>Timeline>Reverse Frames** или команду контекстного меню кадра **Reverse Frames**
Очистка содержимого кадра	Выделить кадры и выполнить команду **Edit>Timeline>Clear Frames**, <Alt>+<Backspace> или команду контекстного меню кадра **Clear Frames**
Удаление кадров	Выделить кадры и выполнить команду **Edit>Timeline>Remove Frames**, <Shift>+<F5> или команду контекстного меню кадра **Remove Frames**
Редактирование свойств кадра	Щелкнуть на кадре и открыть инспектор свойств **Window>Properties**, <Ctrl>+ <F3>
Режим калькирования **Onion Skin** и режим множественного редактирования кадров **Edit Multiple Frames**	Использовать пиктограммы в нижней части монтажной линейки . Диапазон калькирования или множественного редактирования устанавливается с помощью маркеров в верхней части монтажной линейки

Символы и экземпляры

Символ (symbol) — это объект рабочей среды, который может быть неоднократно использован в фильме, при этом его многократное применение несущественно влияет на объем конечного файла.

После создания символ автоматически помещается в библиотеку документа. Таким образом, в библиотеке находится эталон (master object), или оригинал объекта. В фильме же используется *экземпляр* (instance) символа, являющийся ссылкой на оригинал. На сцене одновременно может находиться неограниченное количество экземпляров одного и того же символа, причем число этих экземпляров практически не влияет на объем конечного файла. Между символом и его экземплярами существует тесная связь. Любое изменение символа в библиотеке приведет к изменению всех его экземпляров, используемых в фильме. В то же время невозможно изменить базовые свойства экземпляра, не изменяя символа. В связи с этим к экземплярам символа не могут быть применены операции, изменяющие базовые свойства их содержимого. Тем не менее, используя операции трансформации и *цветовые эффекты экземпляров*, можно изменять пропорции и цветовые характеристики отдельных экземпляров, добиваясь визуального разнообразия. Таким образом, можно, например, "посадить" на сцене целый лес деревьев, используя множество экземпля-

ров одного и того же символа. При этом конечный объем файла будет зависеть главным образом от объема изображения дерева, помещенного в символ (рис. 1.6).

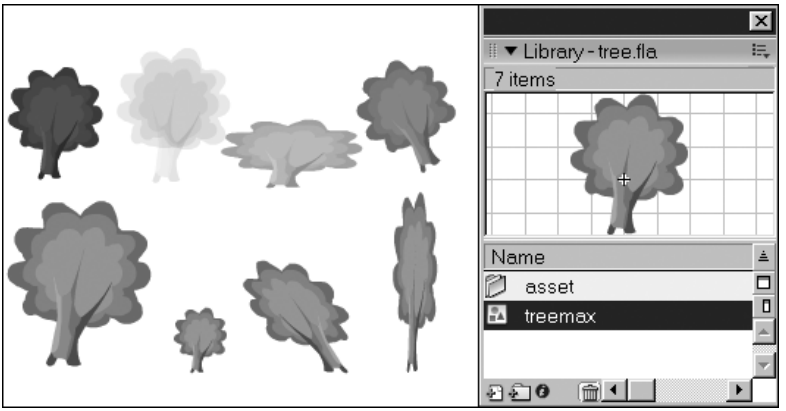


Рис. 1.6. Символ и его экземпляры

Символы могут содержать не только статичные графические объекты, но также анимацию, звук и программные коды, это дает возможность многократного использования в фильме анимированных фрагментов или интерактивных элементов. Любой символ содержит свою собственную монтажную линейку, работа с которой осуществляется точно так же, как и с монтажной линейкой основного фильма.

Типы символов и операции с ними

Во Flash применяются три типа символов, которые имеют определенное назначение и используются для выполнения различных задач. Тип символа характеризуется его *поведением* (behavior), которое задается при создании символа и может быть в дальнейшем переопределено.

Таблица 1.11 обобщает сведения о типах символов, их назначении и характерных особенностях.

Таблица. 1.11. *Типы символов*

Тип символа	Назначение	Содержание				
		Векторная графика, растр, текст	Анимация	Экземпляры символов	Звук	Сценарий AS
Graphic графика	Служит для хранения статичных изображений или анимированных объектов. Монтажная линейка зависит от родительской монтажной линейки	+	+	Графика, Муви-клип, Кнопка	–	–

Таблица. 1.11 *(окончание)*

Тип символа	Назначение	Содержание				
		Векторная графика, растр, текст	**Анимация**	**Экземпляры символов**	**Звук**	**Сценарий AS**
Movie Clip муви-клип 	Служит для создания анимированных интерактивных роликов, многократно используемых в фильме и/или при необходимости программного управления объектом. Монтажная линейка не зависит от родительской монтажной линейки. Обрабатывается программно	+	+	Графика. Муви-клип. Кнопка	+	+
Button кнопка 	Служит для инициирования действий, осуществления навигации и т. д. Монтажная линейка имеет 4 кадра, не зависит от родительской монтажной линейки. Могут быть пустыми (без графики). Обрабатываются программно	+ (маска не работает)	+	Графика, Муви-клип	+	+ (только на экземпляре кнопки, а не на монтажной линейке)

В табл. 1.12 обобщены различные операции с символами и способы их выполнения.

Таблица. 1.12. *Операции с символами*

Операция	Способ
Создание символа	1. Команда **Insert>New Symbol...**, <Ctrl>+<F8> — создать новый пустой символ. 2. Команда **Modify>Convert to Symbol**, <F8> — создать символ из выделенных объектов сцены или среды редактирования другого символа. 3. Пиктограмма  в библиотеке. 4. Команда **New Symbol...** в контекстном меню библиотеки. 5. Перетащить выделенный объект в окно библиотеки

Таблица. 1.12 *(окончание)*

Операция	Способ
Редактирование символа	1. Команда **Edit** в контекстном меню библиотеки или объекта библиотеки. 2. Дважды щелкнуть на пиктограмме символа в библиотеке или по его изображению в окне просмотра. 3. Дважды щелкнуть по копии символа на сцене. 4. Команды **Edit**, **Edit in Place**, **Edit in New Window** контекстного меню экземпляра. 5. Выделить экземпляр символа и выполнить команду **Edit>Edit Symbols**, <Ctrl>+<E>. 6. Выделить экземпляр символа и выполнить команду **Edit>Edit Selected**. 7. Выбрать требуемый символ из общего списка символов с помощью пиктограммы  в полосе редактирования
Выделение в библиотеке	Щелкнуть на пиктограмме символа в окне библиотеки фильма, с <Shift> — выделение диапазона, с <Ctrl> — вразбивку
Переименование	1. Команда **Rename** в контекстном меню библиотеки. 2. Дважды щелкнуть на имени символа в библиотеке. 3. Команда **Rename** в контекстном меню объекта библиотеки, <F2>
Копирование	1. Команда **Duplicate...** в меню библиотеки. 2. Команда **Duplicate** в контекстном меню окна библиотеки. 3. Выделить экземпляр символа на сцене, команда **Duplicate Symbol...** в контекстном меню экземпляра
Изменение типа символа	1. Команда **Properties...** или команда **Type** в контекстном меню окна библиотеки. 2. Команда **Properties...** в меню библиотеки. 3. Пиктограмма  в библиотеке
Удаление	1. Команда **Delete** в меню библиотеки. 2. Щелчок на пиктограмме мусорной корзины в библиотеке. 3. Команда **Delete** в контекстном меню библиотеки

Экземпляр символа и его свойства

Экземпляр (instance) является ссылкой на родительский символ, находящийся в библиотеке. Фильм Flash может включать любое число экземпляров одного и того же символа, которые могут размещаться на сцене или входить в состав других сим-

волов, образуя вложенные символы. Применение символов позволяет существенно оптимизировать объем конечного фильма, поскольку число вхождений экземпляров практически не отражается на размере результирующего файла. В связи с этим при необходимости использования в проекте одинаковых (или похожих) объектов применение символов следует взять за правило.

Таблица 1.13 обобщает различные операции с экземплярами символов и способы их выполнения.

Таблица 1.13. *Операции с экземплярами символов*

Операция	**Способ**
Создание	Перетащить из библиотеки на сцену
Выделение	Щелкнуть инструментом **Selection** или обвести рамкой, добавить к выделенному с <Shift>
Копирование	1. <Ctrl>+<D>. 2. Перетащить инструментом **Selection** с <Alt> или <Ctrl>. 3. Через буфер обмена
Изменение типа поведения **Behavior**	Выбрать другой тип поведения в поле **Symbol Behavior** Инспектора свойств
Трансформация	Инструмент **Free Transform**, возможны все операции трансформации, доступные на наложенном уровне, — масштабирование, поворот, скос, зеркальное отражение
Цветовые эффекты	Назначить эффекты цвета и прозрачность в поле **Color** Инспектора свойств
Замена экземпляра на другой	1. Нажать кнопку **Swap...** Инспектора свойств и выбрать в диалоговом окне **Swap Symbol** другой символ из списка. 2. Команда **Modify>Symbol>Swap Symbol...** 3. Команда **Swap Symbol...** контекстного меню экземпляра
Конвертирование в новый символ	Команда **Convert to Symbol...** контекстного меню экземпляра
Разделение	1. Команда **Modify>Break Apart**; <Ctrl>+<B> приводит к разрыву связи экземпляра и символа, экземпляр становится набором векторных форм рабочего уровня. 2. Команда **Break Apart** контекстного меню экземпляра
Анимация	Тип анимации **Motion Tween**
Удаление	1. Клавиша <Delete>. 2. Команда меню **Edit>Clear**. 3. Удалить символ, по которому создан экземпляр, из библиотеки

Работа с библиотекой

Каждый FLA-документ содержит свою библиотеку (library), в которой хранятся импортированные объекты (растровые изображения, видеоролики, звуки) и символы. С помощью команд библиотеки можно задавать различные параметры содержащихся в ней объектов и упорядочивать их размещение.

Далее представлен перечень операций, которые можно производить над объектами библиотеки, и способы их выполнения.

- **Выделение.** Для того чтобы выделить объект библиотеки, нужно щелкнуть в перечне объектов по любому месту соответствующей строки. Для выделения нескольких объектов, следующих друг за другом, нужно выделить первый объект диапазона, а затем щелкнуть на последнем, удерживая клавишу <Shift>. С помощью клавиши <Ctrl> можно выделить объекты вразбивку.
- **Создание новых объектов.** Для того чтобы создать новый символ, можно воспользоваться пиктограммой **New Symbol...**  в нижней части окна библиотеки или одноименной командой контекстного меню библиотеки. Для создания нового шрифтового символа (Font Symbol) или видеообъекта необходимо воспользоваться командами контекстного меню библиотеки **New Font...** и **New Video...** соответственно.
- **Переименование.** В качестве имени объекта библиотеки может быть использована любая последовательность символов (буквенные, числовые или знаки пунктуации), при этом допустимо использование кириллицы. Имя одного объекта библиотеки не может быть использовано в качестве имени другого ее объекта. Переименовать объект можно одним из следующих способов.
  - Дважды щелкнуть на существующем имени и задать другое имя.
  - Выделить объект и, нажав клавишу <F2>, задать другое имя.
  - Щелкнуть правой кнопкой мыши по соответствующей строке и из контекстного меню объекта выбрать команду **Rename**.
  - Выделить объект и, воспользовавшись контекстным меню библиотеки, выполнить команду **Rename**.
- **Создание копии.** Чтобы скопировать объект, можно выполнить одно из следующих действий.
  - Щелкнуть правой кнопкой мыши на соответствующей строке и из контекстного меню объекта выбрать команду **Duplicate**.
  - Выделить объект и, воспользовавшись контекстным меню библиотеки, выполнить команду **Duplicate**.
- **Сортировка.** Изменить порядок размещения объектов в библиотеке можно с помощью пиктограммы **Toggle Sorting Order** , расположенной над вертикальной полосой прокрутки библиотеки. Для того чтобы отсортировать объекты библиотеки по порядку на основе любого из свойств, указанных в заголовках столбцов библиотеки (Name, Type и т. д.), необходимо щелкнуть на соответствующем заголовке и затем воспользоваться пиктограммой **Toggle Sorting Order**.

- **Организация работы с помощью папок.** Папки выполняют в библиотеке примерно такую же роль, как и каталоги в файловой системе. Они служат для более компактной и упорядоченной организации элементов. Чтобы создать папку можно либо щелкнуть на пиктограмме **New Folder**  в нижней части окна библиотеки, либо выполнить одноименную команду ее контекстного меню. Вновь созданная папка не содержит объектов и отображается пиктограммой . Для того чтобы поместить туда объекты, их нужно выделить и перетащить, "бросив" в требуемую папку. Допускается использование вложенных папок. Папка, наполненная содержимым, отображается пиктограммой . Чтобы раскрыть папку для доступа к ее содержимому, нужно дважды щелкнуть на ее пиктограмме или выполнить команду контекстного меню **Expand Folder**. Раскрытая папка отображается пиктограммой . Объекты, находящиеся внутри раскрытой папки, отображаются с отступом. Для закрытия папки нужно дважды щелкнуть на ее пиктограмме или выполнить команду контекстного меню **Collapse Folder**. Чтобы вытащить объект (или несколько) из папки, необходимо выделить его и переместить за пределы папки.

  Для того чтобы поместить выделенные объекты в автоматически созданную папку, можно выполнить команду **Move to New Folder** контекстного меню объектов или библиотеки. В появившемся диалоговом окне **New Folder** нужно будет задать имя новой папки.

  При работе с большим количеством символов (или импортированных объектов) имеет смысл размещать тематически близкие объекты библиотеки в отдельных папках, присваивая им лаконичные осмысленные имена.

- **Размещение в фильме.** Для того чтобы поместить экземпляр объекта библиотеки в фильм, необходимо перетащить его курсором из списка объектов или окна просмотра библиотеки на сцену или в среду редактирования символа.

- **Изменение свойств объекта.** Для того чтобы изменить свойства объекта библиотеки (изменить тип символа, задать параметры компрессии растровых изображений и звуков и т. д.), можно щелкнуть на пиктограмме **Properties...**  либо выполнить одноименную команду контекстного меню библиотеки или объекта.

- **Удаление.** Для того чтобы удалить один или несколько объектов библиотеки, необходимо выделить их и либо щелкнуть на пиктограмме **Delete** , либо выполнить одноименную команду контекстного меню объекта или библиотеки.

Работа с общими библиотеками (Shared Libraries)

При работе с проектом, включающим в себя несколько файлов SWF, содержащих различные разделы данного проекта, зачастую возникает необходимость использовать одинаковые элементы, включая их в каждый раздел.

Общая библиотека содержит ресурсы (символы, растровые изображения, звуки), которые могут быть использованы любыми файлами проекта. При необходимости использования объекта вместо его встраивания в файл устанавливается связь (linkage) с соответствующим элементом общей библиотеки. При этом файл, содер-

жащий общую библиотеку, должен быть загружен один раз, после чего его содержимое может многократно использоваться другими файлами, причем применение элементов общей библиотеки практически не увеличивает объем файла-получателя.

Использование общих библиотек предполагает выполнение следующих этапов.

- Определение элементов общей библиотеки.
- Включение элементов общей библиотеки в другие файлы проекта.
- Размещение файла SWF, содержащего общую библиотеку, и остальных файлов проекта на сервере.

Для создания общей библиотеки необходимо выполнить следующие действия:

1. Создать новый FLA-документ или открыть имеющийся.
2. Поместить в библиотеку этого документа элементы, используемые в качестве общих ресурсов: символы (графический, муви-клип или кнопка), растровые изображения, звуки, шрифтовые символы *(см. главу 14)*.
3. Для каждого элемента задать параметры связи. Для этого необходимо выделить соответствующий элемент библиотеки документа и, щелкнув на нем правой кнопкой мыши, выбрать из контекстного меню команду **Linkage...**. Далее в появившемся диалоговом окне **Linkage Properties** задать следующие параметры связи:

   - **Identifier** — уникальное имя, используемое для идентификации связанного элемента общей библиотеки. Данное имя должно содержать только символы латинского алфавита и не может включать пробелы;
   - **Export for runtime sharing** — установка этого флажка указывает на то, что данный объект будет экспортироваться в качестве элемента общей библиотеки;
   - **URL** — в это поле необходимо ввести абсолютный или относительный путь к файлу, содержащему общую библиотеку. Относительный путь указывается относительно местоположения файла-получателя. Если файл-получатель размещается на HTML-странице, то в поле **URL** должен быть указан путь к файлу, содержащему общую библиотеку, относительно местоположения HTML-страницы файла-получателя.

     Абсолютный адрес общей библиотеки или путь к файлу, содержащему общую библиотеку, относительно документа-получателя может также быть задан с помощью команды контекстного меню библиотеки **Shared Library Properties**. В результате выполнения этой команды появляется одноименное диалоговое окно, в котором можно задать требуемый путь. Путь, указанный в окне **Shared Library Properties**, автоматически прописывается в поле **URL** окна **Linkage Properties**. Верно и обратное — при изменении значения поля **URL** в окне **Linkage Properties** изменяется путь, заданный в окне **Shared Library Properties**.

Примечание

Флажок **Export in first frame**, устанавливаемый по умолчанию, приводит к тому, что при загрузке файла, содержащего объекты, экспортируемые в качестве объектов общей библиотеки, они будут загружены перед загрузкой содержимого первого кадра фильма. Если документ, содержащий элементы общей библиотеки, используется не только как источник общих ресурсов, но и как самостоятельный раздел проекта, предназначенный для

просмотра, то установка данного флажка приведет к задержке начала воспроизведения документа и, в частности, к задержке выполнения сценариев первого кадра, в котором зачастую содержится предзагрузчик. Если флажок **Export in first frame** снят, то объекты, экспортируемые в качестве общих ресурсов, будут загружены при их первом вхождении в фильм, содержащий общую библиотеку. В этом случае их экземпляры должны быть непосредственно помещены на монтажную линейку документа, содержащего общую библиотеку. Если этого не сделать, элементы общей библиотеки не будут экспортированы в конечный файл SWF. Таким образом, если документ используется исключительно как общая библиотека и не должен демонстрироваться зрителю, флажок **Export in first frame** необходимо установить. Если же документ, содержащий элементы общей библиотеки, выполняет еще и роль определенного раздела проекта и содержит какие-то объекты на своей монтажной линейке, нужно исходить из конкретной ситуации и либо установить флажок (это может привести к задержке воспроизведения), либо снять его (в этом случае элементы общей библиотеки должны быть помещены на монтажную линейку данного документа).

4. Сохранить документ (**File>Save As...**) и опубликовать его (**File>Publish**), присвоив имя в соответствии с именем, указанным в поле **URL** окна **Linkage Properties**.

Чтобы включить элемент общей библиотеки в другой документ Flash, необходимо выполнить следующие действия.

1. Открыть документ, в который должен быть помещен элемент общей библиотеки.
2. Открыть библиотеку документа, содержащего элементы общей библиотеки (**File>Import>Open External Library...**).
3. Перетащить требуемые элементы из внешней библиотеки на сцену или в библиотеку текущего документа. В результате этих действий импортированным объектам в окне **Linkage Properties** автоматически будут заданы следующие параметры связи:
   - **Identifier** — уникальный идентификатор импортированного элемента общей библиотеки. Данное имя должно совпадать с именем, заданным при определении этого объекта в качестве элемента общей библиотеки;
   - **Import for Runtime Sharing** — установка этого флажка указывает на то, что данный объект будет импортирован из общей библиотеки в текущий документ;
   - **URL** — адрес SWF-файла, содержащего общую библиотеку, указанный относительно местоположения текущего документа.
4. Сохранить документ (**File>Save As...**) и опубликовать его (**File>Publish**), разместив конечный файл таким образом, чтобы его местоположение относительно файла, содержащего общую библиотеку, соответствовало пути, указанному в поле **URL** окна **Linkage Properties**.

При размещении конечных файлов на сервер необходимо отправить и все SWF-файлы проекта и файл SWF, содержащий общую библиотеку. При этом важно разместить их друг по отношению к другу таким образом, чтобы все ссылки на общую библиотеку разрешались корректно (взаимное размещение файла-получателя и файла с общей библиотекой должно соответствовать пути, указанному в поле **URL** окна **Linkage Properties**).

Разрешение вопросов политики безопасности при импорте внешних данных из других доменов

С выходом новой версии Flash Player 7 изменились правила политики безопасности, действующие при загрузке внешних данных (общих библиотек, документов XML, текстовых документов и т. д.). В частности, импорт ресурсов общих библиотек может быть непосредственно осуществлен только при условии, что фильм-получатель и фильм, содержащий общую библиотеку, расположены в одном и том же домене. Таким образом, фильм, расположенный по адресу **www.wherever.com**, не сможет загрузить ресурсы общей библиотеки с сайта **www.anywhere.com**.

Для того чтобы получить возможность осуществлять загрузку ресурсов общей библиотеки, расположенной в другом домене, необходимо, чтобы в домене, содержащем общую библиотеку, находился файл с описанием междоменной политики безопасности (cross-domain security). Этот файл представляет собой XML-документ, в котором указано, фильмы с какого домена имеют право осуществлять загрузку данных, хранящихся в настоящем домене. Когда Flash-фильм, расположенный в одном домене, пытается осуществить загрузку данных из другого домена, Flash Player автоматически проверяет наличие файла с политиками в корневом каталоге этого домена. Если домен, с которого осуществляется попытка получить доступ к данным, включен в файл с политиками, данные будут доступны.

Файл с политиками должен быть назван **crossdomain.xml** и помещен в корневой каталог сервера, предоставляющего данные. Файлы с политиками действуют только на серверах, использующих протоколы HTTP, HTTPS или FTP, и только в пределах конкретного порта и протокола.

Документ **crossdomain.xml** должен содержать следующие теги:

- `<allow-access-from>` — тег, содержащий атрибут `domain`, значением которого является точное имя домена, фильмам с которого разрешается получать доступ к ресурсам, хранящимся в домене, содержащем **crossdomain.xml**;
- `<cross-domain-policy>` — внешний тег, содержащий необходимое количество вложенных тегов `<allow-access-from>`.

Ниже приведен пример файла с описанием политики, позволяющей осуществить доступ к данным SWF-файлам, расположенным в доменах **www.anywhere.com** и **www.wherever.com**.

```
<cross-domain-policy>
<allow-access-from domain="*www.anywhere.com"/>
<cross-domain-policy/>
```

Значение атрибута `domain` должно представлять собой точный домен, точный IP-адрес, или может быть задано с применением шаблона "звездочка" (`*`). Звездочка позволяет осуществлять загрузку ресурсов с сервера фильмами со всех поддоменов домена, перед именем которого указана звездочка.

Ниже приведен пример файла с описанием политики, позволяющей осуществить доступ к данным SWF-файлам, расположенным в любом поддомене домена **anywhere.com**.

```
<cross-domain-policy>
<allow-access-from domain="*.anywhere.com"/>
<cross-domain-policy/>
```

Для того чтобы организовать доступ к данным, хранящимся в домене **www.wherever.com** из любого домена, можно в качестве значения атрибута `domain` использовать звездочку. Следующий код позволяет получать доступ к ресурсам сервера фильмам, расположенным в любом домене.

```
<cross-domain-policy>
<allow-access-from domain="*"/>
<cross-domain-policy/>
```

Если файл с политиками не содержит тега `<allow-access-from>`, это равносильно отсутствию данного файла.

Анимация

Flash предоставляет различные способы создания анимации. К их числу можно отнести:

- покадровую анимацию;
- автоматическую анимацию;
- программную анимацию.

Покадровая анимация является наиболее трудоемкой, но в то же время позволяет добиться точной передачи нюансов движения и, как правило, используется для создания художественных детально проработанных эффектов (например, анимация пламени).

Автоматическая анимация (Tweened animation) создается в рабочей среде на монтажной линейке основного фильма или символа (графического типа или типа мувиклип). Автоматическая анимация, в отличие от покадровой, не требует ручной прорисовки фаз. Достаточно задать начальное и конечное состояние анимируемого объекта, а все промежуточные состояния будут рассчитаны автоматически. Диапазон применения автоматической анимации очень широк. С ее помощью можно создавать эффекты, связанные с изменением формы, цвета и прозрачности, перемещением, вращением и масштабированием объектов. Процесс создания автоматической анимации значительно менее трудоемкий, чем покадровой, и во многих случаях позволяет добиться превосходных результатов. Существует два типа автоматической анимации: *анимация формы* (shape tweening) и *анимация движения* (motion tweening). Сравнительные характеристики типов автоматической анимации представлены в табл. 1.14.

Программная анимация создается средствами ActionScript, обладает широкими возможностями и применяется для разработки анимационных эффектов, обладающих сложным характером движения.

Зачастую в фильме используется сочетание всех трех способов, что позволяет решать широкий спектр задач, поставленных перед аниматором.

Таблица 1.14. *Сравнительные характеристики типов автоматической анимации*

Сравнительные характеристики	**Тип автоматической анимации**	
	Анимация формы Shape Tweening	**Анимация движения Motion Tweening**
Способ создания	Панель **Properties**, список **Tween**, тип **Shape**	1. Панель **Properties** список **Tween** тип **Motion**. 2. Команда **Insert> Timeline>CreateMotionTween**. 3. Команда контекстного меню кадра **Create Motion Tween**
Вид на монтажной линейке	 зеленый фон	 голубой фон
Уровень иерархии графических объектов	Рабочий уровень	Наложенный уровень
Типы графических объектов	Векторные формы, контуры, разбитые текст и растровая графика	Группы, экземпляры растровых изображений, текстовые блоки, экземпляры символов, внедренные видеоролики
Количество одновременно анимируемых объектов в одном слое	Не ограничено	Только один объект
Свойства объектов, изменяемые при анимации	Форма, цвет, прозрачность, размер, координаты	Координаты, размер, поворот, скос, цветовые эффекты экземпляров символов **Color Effect** (яркость, цвет и прозрачность)
Возможности	Автоматическое ускорение/замедление анимации, характер перехода (Blend)	Автоматическое ускорение/замедление анимации, автоматическое масштабирование и вращение объекта
Особенности	Использование меток подсказки **Shape Hints**	Возможность анимации по маршруту

Покадровая анимация (Frame by frame animation)

Покадровая, или *ручная* анимация предполагает использование ключевых кадров для размещения в них отдельных фаз движения. Если та или иная фаза должна экспонироваться в течение некоторого промежутка времени, более длительного, чем время просмотра ключевого кадра, то после данного ключевого кадра размещается необходимое число простых кадров. Таким образом, покадровая анимация представляет собой последовательность ключевых и простых кадров. Ключевые кадры содержат основные фазы движения, а простые служат для продления времени их просмотра.

При создании покадровой анимации необходимо придерживаться следующих правил.

- **Разделение статики и динамики** — статичные объекты, не подвергающиеся анимации, должны размещаться на отдельных слоях. Так, например, фоновое изображение, остающееся неподвижным в течение всего фильма, должно быть помещено на отдельный слой, в противном случае его придется перерисовывать из кадра в кадр, что в итоге приведет к запутанности структуры фильма и увеличению объема конечного файла.
- **Разделение одновременных анимационных действий** — если в фильме одновременно используется несколько анимационных процессов, то каждый из них должен быть размещен отдельно на своем собственном слое. Это правило позволит правильно организовать структуру фильма и гибко манипулировать объектами в процессе работы. Так, например, если на сцене одновременно движутся два персонажа, то анимировать каждого из них нужно в отдельном слое, в противном случае можно столкнуться с серьезными трудностями при редактировании анимации, изменении порядка следования планов фильма или одновременном выполнении преобразований содержимого нескольких кадров.
- **Разделение планов фильма** — объекты, принадлежащие разным планам (передний, средний, задний и т. д.), должны быть размещены в разных слоях. Такая организация позволит успешно реализовать впечатление пространственной глубины и легко управлять структурой фильма.

Анимация формы (Shape tweening)

Анимация формы может быть применена исключительно к объектам *рабочего* уровня. С ее помощью можно выполнить следующие виды трансформаций:

- изменение формы;
- изменение цвета и прозрачности;
- перемещение;
- изменение масштаба, скос (наклон).

Для того чтобы создать автоматическую анимацию формы, необходимо выполнить следующие действия.

1. Создать новый слой (**Insert>Timeline>Layer**).
2. В начальном ключевом кадре создать объект *рабочего* уровня (векторную форму и/или контур), например, красный квадрат.

3. Исходя из скорости воспроизведения монтажной линейки, рассчитать необходимое для выполнения анимации число кадров и создать конечный ключевой кадр (**Insert Keyframe** или **Insert Blank Keyframe**). Если анимация должна выполняться за 2 секунды, то при скорости воспроизведения 12 кадров в секунду новый ключевой кадр должен иметь номер 24.
4. Поместить новый объект *рабочего* уровня в пустой ключевой кадр или изменить скопированный из начального кадра объект. В 24-м кадре можно в качестве примера создать синюю звезду. Если координаты объектов в начальном и конечном ключевых кадрах различны, то в ходе анимации будет выполняться перемещение.
5. Задать параметры автоматической анимации. Для этого необходимо выделить *первый* ключевой кадр диапазона и в Инспекторе свойств установить следующие настройки.
   - В выпадающем списке **Tween** выбрать тип анимации **Shape**. При этом начальный и конечный ключевые кадры будут соединены прямой неразрывной линией со стрелкой. Если режим **Tinted Frames** контекстного меню монтажной линейки активен, то диапазон кадров автоматической анимации формы будет выделен зеленоватым цветом. Если в анимации допущена ошибка, то на монтажной линейке появляется пунктирная линия, а при выделении соответствующего кадра в Инспекторе свойств появляется кнопка с пиктограммой восклицательного знака , свидетельствующая об ошибке в анимации.
   - Поле **Easing Value** используется для задания относительного замедления анимации в начале или в конце диапазона. Значение, принимаемое параметром **Ease**, изменяется в диапазоне от —100 до +100 и может быть непосредственно введено вручную либо задано с помощью вертикального слайдера. Отрицательные значения приводят к замедлению анимации в начале (**Easing In**) и, соответственно, к ускорению в конце. Положительные значения приводят к замедлению анимации в конце (**Easing Out**) и к ускорению в начале. При задании отрицательного значения рядом с полем **Easing Value** появляется предлог **In**, а при задании положительного значения — предлог **Out**. Замедление анимации достигается не за счет замедления скорости воспроизведения монтажной линейки, а за счет более плотного размещения фаз анимируемого объекта в начале или в конце диапазона кадров трансформации.
   - Параметр **Blend** позволяет задать тип перехода между начальной и конечной формами. Значение **Angular** приводит к тому, что при переходе у промежуточных форм сохраняются углы и прямолинейные сегменты. Значение **Distributive**, наоборот, сглаживает промежуточные формы. Переход типа **Angular** может быть применен только к формам, содержащим углы и прямолинейные сегменты; в противном случае переход выполняется с использованием значения **Distributive**.
6. Просмотреть результат в рабочей среде (**Control>Play**) или в среде тестирования (**Control>Test Movie**).

Анимация движения (Motion tweening)

Анимация движения может быть применена исключительно к объектам *наложенного* уровня. С ее помощью можно выполнить следующие виды трансформаций:

- перемещение объектов по линейной и произвольной траектории;
- изменение масштаба, скос (наклон), вращение;
- изменение яркости, цвета и прозрачности экземпляров символов с помощью цветовых эффектов (**Color Effect**).

Технология создания автоматической анимации движения аналогична описанной выше процедуре создания анимации формы, однако данный тип анимации имеет ряд характерных только для него параметров настройки. Для создания анимации движения необходимо выполнить следующие действия.

1. В начальный ключевой кадр нового слоя поместить объект наложенного уровня (группу, растровое изображение, видеоролик, текстовый блок, экземпляр символа).
2. В конечный кадр диапазона вставить ключевой кадр (**Insert Keyframe**), поместив в него тот же объект, который содержится в начальном кадре.
3. В последнем кадре диапазона задать конечное состояние объекта, изменив его свойства как объекта наложенного уровня. Для этого можно изменить его координаты, отмасштабировать, повернуть или скосить и, если речь идет об экземпляре символа, применить любой цветовой эффект.
4. Задать параметры автоматической анимации. Для этого необходимо выделить любой кадр диапазона, кроме последнего ключевого, и в Инспекторе свойств установить следующие настройки.
   - В выпадающем списке **Tween** выбрать тип анимации **Motion**. При этом начальный и конечный ключевые кадры будут соединены прямой неразрывной линией со стрелкой. Если режим **Tinted Frames** контекстного меню монтажной линейки активен, то диапазон кадров автоматической анимации формы будет выделен голубоватым цветом. Если в анимации допущена ошибка, на монтажной линейке появляется пунктирная линия, а при выделении соответствующего кадра в Инспекторе свойств появляется кнопка с пиктограммой восклицательного знака , свидетельствующая об ошибке в анимации.
   - Флажок **Scale** — устанавливается при необходимости плавного изменения масштаба объекта на протяжении диапазона анимации. Если размеры объекта в начальном и конечном кадрах отличаются, то автоматическое масштабирование будет выполняться только при установке данного флажка.
   - Поле **Easing Value** используется для задания относительного замедления анимации в начале или в конце диапазона.
   - Список **Rotate** позволяет задать параметры автоматического вращения объекта на протяжении диапазона кадров анимации. Вращение выполняется относительно точки трансформации объекта; смещая точку трансформации, можно изменить характер вращения объекта. Для корректного выполнения анимации точки трансформации у объектов в начальном и конечном кадрах должны на-

ходиться в одном и том же положении. Ниже перечислены параметры вращения объекта:

- ◊ **None** — вращение отключено;
- ◊ **Auto** — автоматическое вращение от угла поворота, заданного в начальном кадре, до угла, заданного в конечном кадре. Вращение выполняется по кратчайшему пути. Так, если в начальном кадре угол поворота составляет 0°, а в конечном —45°, то поворот будет выполняться на 45° против часовой стрелки;
- ◊ **CW** — автоматическое вращение по часовой стрелке (clockwise). Число полных оборотов, выполняемых за один диапазон анимации, задается в поле **Times**;
- ◊ **CCW** — автоматическое вращение против часовой стрелки (counterclockwise).

- Флажок **Sync** — применяется, если начальный и конечный ключевые кадры диапазона анимации движения содержат разные экземпляры одного и того же анимированного символа типа Graphic. Установка данного флажка позволяет синхронизировать первые кадры монтажных линеек данных экземпляров, учитывая выделенный им диапазон кадров трансформации родительской монтажной линейки. Данный режим можно также установить, выделив весь диапазон кадров анимации движения и выполнив команду **Modify>Timeline> Synchronize Symbols** или соответствующую команду контекстного меню кадра.

5. Параметры **Orient to path** и **Snap** используются при создании анимации по заданному маршруту.

Примечание

Необходимо помнить, что в кадре диапазона автоматической анимации движения может находиться только один объект наложенного уровня. Нарушение этого правила приведет к некорректной работе (или прекращению работы) анимации. При необходимости одновременного выполнения нескольких анимационных действий, каждый объект следует помещать на отдельный слой.

6. Просмотреть результат в рабочей среде (**Control>Play**) или в среде тестирования (**Control>Test Movie**).

Использование звука в фильме

Для того чтобы снабдить фильм звуковым сопровождением в рабочей среде, необходимо импортировать соответствующий файл. В рабочую среду Flash можно импортировать звуковые файлы в форматах WAV и MP3.

Для того чтобы импортировать звуковой файл в рабочую среду, необходимо выполнить команду главного меню **File>Import>Import to Library** или команду **File> Import>Import to Stage**. В результате выполнения любой из данных команд звуковой файл помещается в библиотеку текущего документа. Один и тот же звуковой файл, находящийся в библиотеке, может многократно использоваться в фильме. Причем многократное применение экземпляра одного и того же звука не будет приводить к увеличению объема конечного фильма (при использовании определенных типов

синхронизации). Здесь имеет место та же ситуация, что и при работе с растровыми изображениями.

Для того чтобы ввести звук в фильм, необходимо поместить его в ключевой кадр монтажной линейки. Это можно сделать двумя способами.

- Выделить ключевой кадр, в который должен быть помещен звук, и перетащить требуемый звук из библиотеки на сцену, так же как и при создании экземпляра символа.
- Выделить ключевой кадр, в который должен быть помещен звук, и в списке **Sound** панели **Properties**, содержащем все находящиеся в библиотеке текущего документа звуки, выбрать требуемый.

В результате этих действий в соответствующем кадре монтажной линейки появится график цифрового представления звука. Каждый кадр монтажной линейки может содержать только один экземпляр звука. Для того чтобы одновременно использовать несколько разных звуков, необходимо каждый из них размещать на отдельном слое.

В соответствии с назначением звука ему можно задать тип синхронизации, определяющий характерные особенности воспроизведения и обработки звука для каждого применяемого экземпляра. Для того чтобы задать тип синхронизации экземпляра звука, необходимо выделить содержащий его кадр на монтажной линейке и воспользоваться списком **Sync** инспектора свойств.

Flash позволяет задать следующие типы синхронизации звука.

- **Event** (Событие) — данный тип синхронизации используется по умолчанию. Экземпляр звука, использующий тип синхронизации **Event**, будем называть *событийным* звуком. Событийные звуки обладают следующими характерными особенностями.
  - Воспроизведение событийного звука начинается, когда воспроизводящая головка входит в ключевой кадр, содержащий экземпляр данного звука.
  - Событийный звук может быть воспроизведен только после того, как он полностью загрузился. При загрузке фильма по сети с недостаточной скоростью соединения воспроизведение монтажной линейки может быть приостановлено на кадре, содержащем событийный звук, до тех пор, пока данный звук не будет загружен. Этого явления можно избежать с помощью предзагрузчика.
  - Событийный звук не зависит от содержащей его монтажной линейки. Если данный звук запущен, то он прозвучит до конца вне зависимости от того, продолжается ли анимация содержащей его монтажной линейки или нет. Так, если экземпляр пятисекундного звука поместить в первый ключевой кадр монтажной линейки, содержащей 12 кадров (при скорости воспроизведения 12 кадров в секунду), то, будучи запущен, данный звук будет воспроизведен целиком, несмотря на то, что анимация монтажной линейки длится только одну секунду.
  - Событийные звуки *микшируются*, то есть могут накладываться сами на себя. Микширование может иметь место в различных случаях. Например, если событийный звук помещен в первый кадр монтажной линейки, длительность воспроизведения которой меньше продолжительности его звучания, то при зацикливании монтажной линейки звук будет накладываться сам на себя.

Всякий раз при переходе воспроизводящей головки с последнего кадра в первый звук будет запускаться вновь, при этом накладываясь на ранее запущенный экземпляр.

- Многократное использование одного и того же событийного звука практически не приводит к существенному увеличению объема конечного файла. На объем конечного фильма влияет только размер звукового файла, находящегося в библиотеке, а не число его экземпляров. В связи с этим событийные звуки могут быть зациклены, что позволяет использовать их в качестве фонового звукового сопровождения.

- **Start** (Старт) — синхронизация типа **Start** применяется для исключения возможности наложения событийных звуков самих на себя. Экземпляр звука, использующий тип синхронизации **Start**, будем называть звуком типа **Start**. Звук типа **Start** обладает многими свойствами событийного звука, но его характерной особенностью является то, что этот звук будет воспроизводиться только в том случае, если другой экземпляр этого же звука (типа **Event** или **Start**) не воспроизводится в данный момент. Так, если звук типа **Start** помещен в первый кадр монтажной линейки, длительность воспроизведения которой меньше продолжительности его звучания, то при зацикливании монтажной линейки наложения звука не произойдет. При переходе воспроизводящей головки с последнего кадра в первый звук не будет запускаться вновь, если продолжается его воспроизведение, инициированное предыдущим запуском.

- **Stop** (Стоп) — данный тип синхронизации используется для остановки воспроизведения событийного звука. Экземпляр звука, использующий тип синхронизации **Stop**, будем называть звуком типа **Stop**. Как только воспроизводящая головка входит в ключевой кадр монтажной линейки, содержащий звук типа **Stop**, все воспроизводящиеся в этот момент экземпляры того же самого звука будут остановлены. Звук типа **Stop** сам не звучит и применяется исключительно для остановки воспроизведения событийных звуков и звуков типа **Start**. Экземпляр звука типа **Stop** может остановить воспроизведение только экземпляров того же самого звука. На монтажной линейке он отображается знаком паузы (синий прямоугольник).

- **Stream** (Поток) — данный тип используется для синхронизации анимации со звуком. Экземпляр звука, использующий тип синхронизации **Stream**, будем называть *потоковым* звуком. Потоковый звук имеет следующие характерные особенности.

  - Потоковый звук начинает воспроизводиться, как только загрузится достаточное количество информации для начала звучания.

  - Потоковый звук зависит от содержащей его монтажной линейки, то есть он воспроизводится только в тех кадрах, в которых он размещен. Для воспроизведения потокового звука в течение определенного временного интервала после ключевого кадра, содержащего данный звук, необходимо поместить на монтажную линейку соответствующее требуемому интервалу число простых кадров. В отличие от событийного звука, потоковый звук не будет звучать при остановке воспроизведения содержащей его монтажной линейки. Потоковый звук также не будет звучать дольше, чем воспроизводятся кадры, в которых

он размещен. Так, если пятисекундный событийный звук будет помещен на монтажную линейку, содержащую 12 кадров (при скорости воспроизведения 12 кадров в секунду), то при воспроизведении можно будет услышать только первую секунду его звучания.

- Потоковый звук имеет приоритет над визуальным содержимым монтажной линейки. При воспроизведении монтажной линейки Flash стремится сохранить целостность звучания потокового звука и пытается обеспечить синхронное воспроизведение звука и анимации. Однако если из-за низкого быстродействия соединения содержимое некоторых кадров не успевает прорисовываться, они будут пропущены ради нормальной передачи звуковой информации.
- Потоковый звук разбивается на отдельные фрагменты, которые жестко привязываются к содержащим их кадрам монтажной линейки. В связи с этим использование нескольких экземпляров одного и того же потокового звука в разных участках монтажной линейки приведет к пропорциональному увеличению объема конечного файла. Поэтому потоковые звуки не рекомендуется зацикливать.

Таблица 1.15 обобщает основные возможности работы со звуком.

Таблица 1.15. *Операции со звуком*

Операция	**Способ**
Импорт	1. Команда **File>Import>Import to Library**. 2. Команда **File>Import>Open External Library**; <Ctrl>+<Shift>+<O>, перетащить звук из библиотеки другого FLA-документа в библиотеку или на сцену текущего фильма (импорт звука из внешнего документа Flash)
Экспорт	Команда **File> Export >Export Movie...**; <Ctrl>+<Alt>+<Shift>+<S>
Оптимизация (сжатие)	Дважды щелкнуть на пиктограмме звука в библиотеке
Добавление звука в кадр	1. Выделить кадр, перетащить звук из библиотеки на сцену. 2. В Инспекторе свойств выбрать звук из списка
Редактирование длительности	Инспектор свойств — кнопка **Edit**, в окне редактирования звука можно перемещать маркеры на шкале, устанавливая начало и конец звучания
Выбор типа синхронизации	Инспектор свойств — выбрать из поля **Sync** тип звука: • **Event** — звук, управляемый событием (мыши или наступлением кадра на монтажной линейке), должен быть загружен полностью, прежде чем он начнет воспроизводиться и будет звучать, пока его не выключат. Назначение — служит для озвучивания кнопок, для зацикленных музыкальных фрагментов; воспроизводится полностью; помещается только в один кадр;

Таблица 1.15 *(окончание)*

Операция	Способ
	• **Start** — экземпляр звука, использующий данный тип синхронизации, будет воспроизведен только, если другой экземпляр этого же звука не воспроизводится в настоящий момент; • **Stop** — прекращает воспроизведение одноименного звука; • **Stream** — потоковый звук начинает звучать еще до полной загрузки. Служит для синхронизации с анимацией, а также однократного использования звука в фильме, воспроизводится только в кадрах, куда помещен
Управление громкостью	В списке **Effect** Инспектора свойств выбрать один из стандартных эффектов звука или создать пользовательский эффект в окне редактирования звука **Edit Envelope** (кнопка **Edit**)

Публикация исходного документа

Настройка параметров публикации документа осуществляется с помощью диалогового окна **Publish Settings**. Для того чтобы открыть это окно, необходимо выполнить команду главного меню **File>Publish Settings...** или воспользоваться сочетанием клавиш <Ctrl>+<Shift>+<F12>. Столбец **Type** содержит перечень всех форматов файлов, которые могут быть получены в результате публикации. Установка соответствующего флажка приводит к тому, что при выполнении команды **Publish** исходный документ будет опубликован в указанном формате. При установке нескольких флажков исходный документ будет транслирован во все выбранные форматы, в результате чего Flash автоматически сгенерирует требуемые файлы.

По умолчанию файлы, сгенерированные при публикации, получают имя исходного документа и сохраняются в том же каталоге, где был сохранен Flash-документ (с расширением fla). Так, например, если исходный документ был сохранен в каталоге myProject с именем myMovie.fla, то в результате публикации в каталог myProject будут помещены файлы myMovie.swf, myMovie.html, myMovie.gif и т. д.

При необходимости можно сохранить результирующий файл под другим именем, отличным от имени исходного документа. Для этого необходимо воспользоваться полем **Filename**, соответствующим требуемому формату, расположенному в столбце **File**. Для того чтобы сохранить результирующий файл (или несколько файлов) в произвольном каталоге, необходимо задать путь к этому каталогу, воспользовавшись кнопкой  **Select Publish Destination**. Кнопка **Use Default Names** позволяет вновь применить названия по умолчанию.

По мере выбора требуемых форматов публикации в верхней части окна **Publish Settings** появляются ярлыки, позволяющие переместиться на вкладку, содержащую соответствующие параметры настройки. Совокупность настроек, задаваемых в окне **Publish Settings**, называется *профилем публикации* (Publish profile). При сохранении документа профиль публикации сохраняется вместе с ним. Один документ может

содержать несколько различных профилей. Flash MX 2004 позволяет сохранять профиль во внешнем файле и импортировать в другие документы. Это дает возможность создания набора стандартных настроек, которые могут быть многократно использованы при работе над различными документами.

Название текущего профиля публикации указывается в списке **Current profile**, расположенном в верхней части окна **Publish Settings**. По умолчанию новый документ использует профиль **Default**, содержащий настройки публикации в форматы Flash (SWF) и HTML. Кнопки, расположенные справа от списка **Current profile**, позволяют выполнить ряд операций с профилями публикации.

- Для того чтобы создать новый профиль, нужно щелкнуть на кнопке **Create New Profile** () и в появившемся диалоговом окне **Create New Profile** указать имя нового профиля. Созданный профиль будет добавлен в список **Current profile** и автоматически сделан текущим.
- Для того чтобы создать копию существующего профиля, нужно воспользоваться кнопкой **Duplicate Profile** ().
- Для того чтобы изменить профиль, нужно выбрать его в списке **Current profile** и произвести все необходимые изменения. Новые установки будут автоматически сохранены в данном профиле.
- Кнопка **Profile Properties** () позволяет переименовать существующий профиль.
- Для того чтобы получить возможность использования профиля публикации текущего документа в других документах Flash, данный профиль необходимо экспортировать и сохранить во внешнем файле XML. Это можно сделать с помощью кнопки **Import/Export Profile** (). В выпадающем меню нужно выбрать команду **Export...**, а в появившемся вслед за этим окне **Export Profile** указать имя сохраняемого профиля. По умолчанию все экспортируемые профили сохраняются в каталоге Publish Profiles. При работе с операционной системой Windows 2000 или Windows XP данный каталог расположен по адресу: <drive>/Documents and Settings/<username>/Local Settings/Application Data/Macromedia/Flash MX 2004/en/Configuration/Publish Profiles. При работе с операционной системой Windows 98 он расположен по адресу: <drive>/Windows/Application Data\Macromedia/Flash MX 2004/en/ Configuration/Publish Profiles.
- Для того чтобы импортировать профиль, нужно щелкнуть на кнопке **Import/Export Profile** и в выпадающем меню выбрать команду **Import...**. Далее в диалоговом окне **Import Profile** нужно найти требуемый XML-файл. При импорте профиля Flash автоматически открывает каталог Publish Profiles.
- Удалить профиль из документа можно с помощью кнопки **Delete Profile** (), при этом соответствующий XML-файл остается в каталоге Publish Profiles.

После выполнения всех настроек документ можно опубликовать. Для этого необходимо выполнить одно из следующих действий:

- щелкнуть на кнопке **Publish** в диалоговом окне **Publish Settings**;
- выполнить команду главного меню **File>Publish** или воспользоваться ее клавиатурным эквивалентом <Shift>+<F12>;

- для того чтобы просмотреть результат публикации в один из выбранных форматов, нужно выполнить команду главного меню **File>Publish Preview**. В выпадающем списке можно выбрать требуемый формат. Выполнение данной команды приводит к публикации документа только в одном выбранном формате;
- при тестировании проекта (**Control>Test Movie**, **Control>Test Scene**) выполняется публикация документа в формат SWF.

Формат Flash (SWF)

Файл SWF включает в себя всю функциональность, которой исходный документ был снабжен на стадии разработки. В результате публикации происходят реорганизация и оптимизация данных, в результате чего SWF-файл занимает значительно меньший объем, чем исходный авторский документ. Формат SWF используется для размещения Flash-фильма в сети Интернет. Для воспроизведения фильма в формате SWF применяется проигрыватель Flash Player. Для просмотра Flash-содержимого с помощью Web-браузера в зависимости от его типа необходимо наличие соответствующей версии элемента управления Flash Player ActiveX (для браузера Internet Explorer) или встраиваемого модуля расширения Flash Player (для браузера Netscape Navigator). Локально размещенный Flash-фильм может быть воспроизведен с помощью автономного проигрывателя Flash (Stand Alone Player), который называется проектор (projector) и поставляется вместе с Flash MX 2004. Его можно найти в каталоге Players, расположенном в подкаталоге, куда были установлены файлы Flash-приложения. Файл проектора называется SAFlashPlayer.exe. Файл SWF также может быть воспроизведен в приложении Macromedia Director при наличии соответствующего расширения (Flash Asset Xtra).

Вкладка **Flash** окна **Publish Settings** позволяет установить следующие параметры публикации (рис. 1.7).

- **Version** — данный список позволяет указать версию проигрывателя Flash Player, для которой будет выполнена публикация. Следует иметь в виду, что фильм, опубликованный для более поздней версии проигрывателя, не может быть воспроизведен более ранней версией. При публикации для более ранней версии (ниже Flash Player 7) часть функциональности проекта, связанная с новыми возможностями Flash, может быть утрачена.
- **Load order** — данный список позволяет указать порядок загрузки элементов первого кадра фильма по слоям: **Bottom up** — снизу вверх или **Top down** — сверху вниз. При медленном соединении бывает важно, чтобы содержимое слоев загружалось в определенном порядке.
- **ActionScript version** — данный список позволяет указать версию языка сценариев ActionScript, которая используется при написании сценариев в документе. В связи с тем, что во второй версии ActionScript появилось несколько существенных отличий (строгая типизация данных, объявление классов во внешних файлах и др.), сценарии в духе ActionScript 1.0 могут не работать или выполняться некорректно. Для того чтобы реализовать обратную совместимость, нужно выбрать первую версию ActionScript 1.0. Если в документе используется ActionScript 2.0, становится доступной клавиша **Settings**, приводящая к появлению диалогового окна **ActionScript Settings**. Здесь можно задать кадр, в котором будет выполняться

включение кодов пользовательских классов, сохраненных во внешних файлах, а также путь (Classpath) к каталогу, содержащему файлы пользовательских классов.

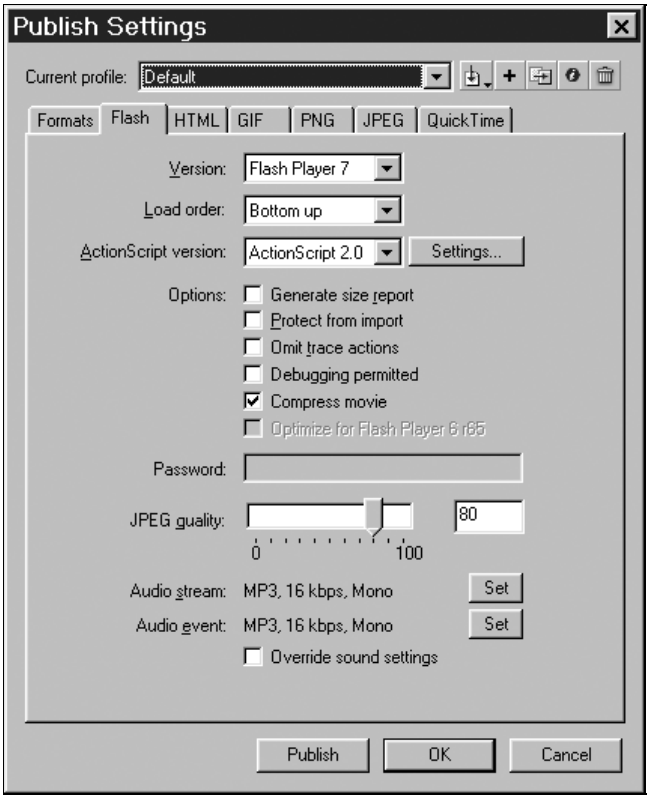


Рис. 1.7. Настройка параметров публикации в формат SWF (Flash)

- **Options** — данный раздел содержит ряд опций, используемых при публикации.
  - **Generate size report** — создать отчет о размере всех элементов фильма. Отчет будет сохранен в формате TXT в том же каталоге, где находится исходный документ.
  - **Protect from import** — установка данного флажка приведет к тому, что сгенерированный при публикации SWF-файл не сможет быть импортирован обратно в рабочую среду. При попытке импортировать или даже открыть защищенный файл в рабочей среде будет выведено сообщение о том, что данная операция не может быть выполнена. При установке данного флажка становится доступным поле **Password**, в которое можно ввести пароль, позволяющий осуществить импорт защищенного файла. Следует иметь в виду, что данная защита предусматривает использование легальных средств и может быть достаточно легко разблокирована.

- **Omit trace actions** — установка данного флажка приведет к тому, что при тестировании фильма все вызовы функции `trace()` будут проигнорированы. Функция `trace()` используется при отладке сценария ActionScript и позволяет в среде тестирования выводить в окно **Output** информацию, помогающую проанализировать ход выполнения программы (сообщения о выполнении определенных условий, результаты вычисления выражений, принимаемые переменными значения и т. д.).
- **Debugging permitted** — установка данного флажка позволяет осуществлять отладку сценария Flash-фильма, размещенного на сервере. Для того чтобы воспользоваться отладчиком сценариев (Debugger), необходимо использовать проигрыватель Flash Debug Player, который устанавливается автоматически при установке Flash MX 2004. При установке данного флажка становится доступным поле **Password**, позволяющее задать пароль для удаленной отладки *(см. главу 28)*.
- **Compress movie** — установка данного флажка позволяет применить к SWF-файлу компрессию и таким образом уменьшить его размер. Эта возможность может быть использована только при работе с проигрывателем Flash Player 6 или выше. Наиболее успешно сжатие может быть применено к фильму, содержащему большие объемы текста и/или сценариев ActionScript.
- **Optimize for Flash Player 6r65** — эта опция становится доступной только при публикации документа для шестой версии Flash Player. Установка данного флажка позволяет оптимизировать фильм для промежуточной версии проигрывателя.

- **JPEG quality** — этот параметр позволяет задать степень сжатия всех используемых в фильме растровых изображений, которые не были оптимизированы с помощью библиотеки, т. е. для которых в окне **Bitmap Properties** был установлен флажок **Use document default quality**. Требуемое значение может быть установлено с помощью слайдера или введено вручную в поле, расположенное справа от него. Значение параметра **quality** задается в диапазоне от 0 до 100 и характеризует собой соотношение качества изображения и размера занимаемого им файла. Чем выше значение качества, тем больший объем будет занимать файл данного изображения.
- **Audio Stream/Audio Event** — кнопки **Set** позволяют задать тип и параметры компрессии для используемых в фильме событийных и потоковых звуков в отдельности. Данные настройки компрессии применяются только к тем звуковым файлам, которые не были индивидуально оптимизированы с помощью библиотеки, т. е. для которых в списке **Compression** окна **Sound Properties** было установлено значение **Default**. Нажатие кнопки **Set** приводит к появлению окна **Sound Settings**, где в списке **Compression** можно выбрать требуемую схему сжатия. Типы компрессии и параметры настройки аналогичны соответствующим параметрам окна **Sound Properties**. При выборе значения **Disable** звук не экспортируется в конечный фильм.
- **Override sound settings** — установка этого флажка позволяет переопределить параметры оптимизации звуков, заданные в библиотеке, текущими настройками окна **Publish Settings**.

Формат HTML

Для распространения Flash-фильма через Интернет на сервер необходимо отправить сам SWF-файл и документ HTML, содержащий набор тегов, необходимых для описания параметров размещения и воспроизведения Flash-фильма. Flash позволяет автоматически сгенерировать HTML-страницу на основе одного из стандартных шаблонов, поставляемых вместе с Flash MX 2004.

При публикации в формате HTML автоматически выделяется формат Flash. Взаимное расположение HTML-страницы и Flash-фильма в результате публикации определяется значениями, заданными в полях **Filename** вкладки **Formats**. На основе этого значения в соответствующих тегах HTML-страницы указывается путь к отображаемому на ней файлу SWF. Если после публикации взаимное расположение данных файлов будет изменено, HTML-код необходимо отредактировать вручную, в противном случае Flash-фильм не сможет быть отображен на странице.

Вкладка **HTML** окна **Publish Settings** позволяет выполнить ряд настроек, на основании которых будет сгенерирована HTML-страница, содержащая соответствующий код для описания параметров размещения и воспроизведения Flash-фильма (рис. 1.8).

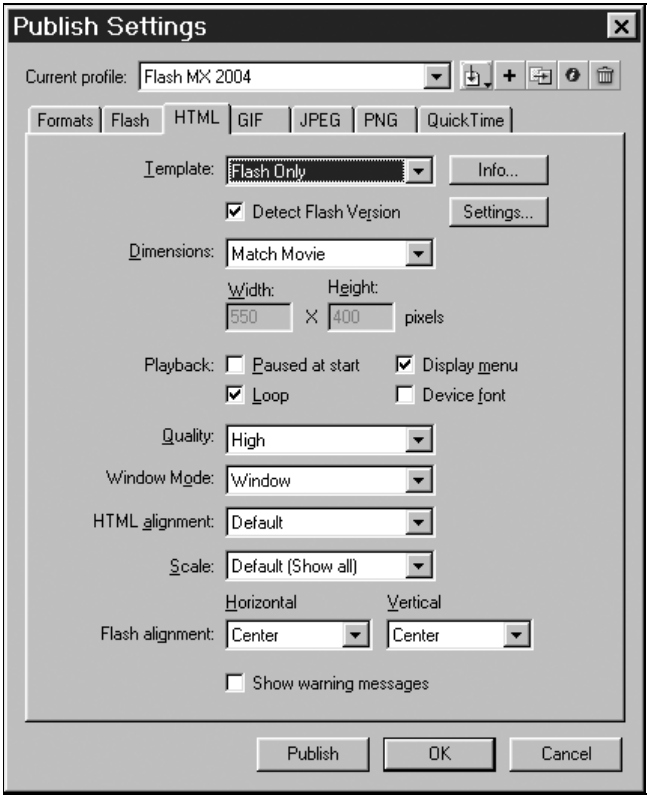


Рис. 1.8. Настройка параметров публикации в формат HTML

Вкладка **HTML** включает следующие параметры настройки:

- Выпадающий список **Template** позволяет выбрать один из предлагаемых шаблонов, поставляемых вместе с Flash MX 2004, на основе которого будет сгенерирована HTML-страница. Код, создаваемый в результате публикации на основе поставляемых шаблонов, соответствует стандарту XHTML. Список содержит следующий набор шаблонов.
  - **Flash Only** — данный шаблон используется по умолчанию и генерирует все необходимые теги для размещения Flash-фильма на странице. Шаблон **Flash Only** можно использовать, если предполагается создание Web-страницы исключительно средствами Flash. Если на странице кроме Flash-содержимого должны находиться другие элементы HTML, данный шаблон удобно использовать для получения исходного кода, описывающего параметры размещения файла SWF, который в дальнейшем может быть отредактирован вручную и включен в соответствующий фрагмент кода конечной страницы.
  - **Flash For Pocket PC 2003** — данный шаблон используется для размещения Flash-содержимого на HTML-странице, предназначенной для просмотра в браузере Pocket Internet Explorer на платформе Pocket PC. Страница, сгенерированная на основе данного шаблона, может быть также просмотрена в обычных браузерах IE и Netscape Navigator.
  - **Flash HTTPS** — данный шаблон позволяет сгенерировать HTML-страницу и разместить на ней Flash-фильм. Если при загрузке данной страницы на компьютере пользователя будет отсутствовать проигрыватель Flash Player, он будет автоматически загружен с сайта компании Macromedia по защищенному каналу с использованием протокола HTTPS.
  - **Flash With AICC Tracking** — данный шаблон используется при разработке интерактивных обучающих программ (e-learning) на основе поставляемых вместе с Flash MX 2004 шаблонов (Quiz) и элементов общих библиотек (Learning Interactions). Данные элементы отслеживают выполняемые пользователем действия и отправляют соответствующую информацию управляющей системе (LMS — learning management system), расположенной на сервере и поддерживающей протокол AICC (Aviation Industry CBT Committee protocol). Информация об использовании поставляемых вместе с Flash интерактивных обучающих элементов содержится в документации к Flash MX 2004.
  - **Flash With FSCommand** — данный шаблон помещает на страницу сценарий, позволяющий организовать взаимодействие Flash-фильма со сценарием Java Script, расположенным на данной странице с помощью функции `fscommand()`;
  - **Flash With Named Anchors** — данный шаблон необходимо использовать для того, чтобы получить возможность перемещаться между кадрами фильма, содержащими указатели или якоря (named anchors) с помощью кнопок браузера Forward (Вперед) и Back (Назад). Эта возможность позволяет пользователю переходить из одного раздела Flash-проекта в другой так же, как если бы это были просмотренные Web-страницы. При этом, правда, все разделы должны находиться в пределах одного фильма. При публикации Flash автоматически сгенерирует HTML-страницу, содержащую сценарий JavaScript (для браузера Netscape Navigator), и якоря HTML, соответствующие указателям Flash. Ис-

пользование якорей поддерживается проигрывателем Flash, только начиная с версии Flash Player 6.

- **Flash With SCORM Tracking** — данный шаблон, так же как и Flash With AICC Tracking, используется при разработке интерактивных обучающих программ. Применение этого шаблона предполагает использование стандарта SCORM (Shareable Content Object Reference Model).
- **Image Map** — данный шаблон используется для создания карты изображения. Карта изображения представляет собой растровое изображение, размещаемое на HTML-странице, содержащее набор активных областей произвольной формы, выполняющих функции гиперссылок. Каждой из таких областей сопоставляется адрес интернет-ресурса; при попадании курсора в активную область он принимает вид руки; нажатие левой кнопки мыши позволяет перейти к соответствующему ресурсу. Карта изображения генерируется на основе содержимого кадра, помеченного меткой **#Static**. В данном кадре необходимо разместить кнопки и назначить им сценарии с использованием функции загрузки сетевого ресурса `getURL()`. Следует принять во внимание, что созданные активные области будут иметь прямоугольную форму, а не форму области реагирования кнопки Flash. При публикации кроме форматов Flash и HTML необходимо выбрать один из предлагаемых графических форматов GIF, JPEG или PNG — тот, который будет использован для создания изображения. При использовании изображения в формате JPEG воспроизводящая головка в момент публикации должна находиться в кадре, играющем роль изображения. В результате публикации будет создана HTML-страница, содержащая не Flash-фильм, а карту изображения.
- **Quick Time** — данный шаблон позволяет разместить на HTML-странице фильм в формате Quick Time.
- **Detect Flash Version** — установка данного флажка позволит активизировать механизм автоматической проверки наличия у пользователя необходимой для просмотра публикуемого SWF-файла версии проигрывателя Flash Player. Использование этой опции приведет к тому, что Flash автоматически сгенерирует и поместит на HTML-страницу (Detection File) SWF-файл (Flash_detection.swf), содержащий сценарий ActionScript, который проверяет номер версии проигрывателя Flash Player, установленного в системе у пользователя. Сценарий использует синтаксис ActionScript версии Flash 4, благодаря чему данный файл может быть воспроизведен у большинства пользователей Интернета, поскольку для его работы достаточно наличия проигрывателя Flash Player 4. Страница, содержащая файл-определитель, должна быть загружена в первую очередь. В этом случае, если проверка покажет, что у пользователя установлена более ранняя версия Flash Player, то произойдет переход на альтернативную страницу (Alternate File), содержащую, например, HTML-версию сайта. Если проверка покажет, что номер версии Flash Player пользователя равен или выше версии, для которой выполнялась публикация, то будет автоматически выполнен переход к основной странице (Content File). Для настройки параметров определения версии необходимо воспользоваться кнопкой **Settings...**, нажатие которой приведет к появлению диалогового окна **Version Detection Settings**.

- **Dimensions** — данный список позволяет задать размер окна Flash Player, в котором будет воспроизводиться фильм. Окно Flash Player представляет собой прямоугольную область без границ — ее можно условно сравнить с киноэкраном. Фильм, который воспроизводится в данном окне, можно сравнить с пятном света, проецируемым на экран. Список **Dimensions** содержит три варианта задания размеров окна:
  - **Match Movie** — размер окна Flash Player задается равным размеру рабочей области документа. В этом случае фильм будет занимать все окно целиком;
  - **Pixels** — произвольный размер окна Flash Player в пикселах. В этом случае размер окна и размер фильма могут отличаться;
  - **Percent** — произвольный размер окна Flash Player в процентах от размера окна браузера. В этом случае размер окна и размер фильма могут отличаться. При изменении размера окна браузера будет происходить соответствующее масштабирование окна Flash Player.
- **Playback —** параметры воспроизведения включают следующие установки:
  - **Paused at start** — при установке этого флажка воспроизведение фильма автоматически не начинается. Запуск монтажной линейки основного фильма происходит в результате выполнения определенного действия (запуска фильма пользователем с помощью команды **Play** контекстного меню проигрывателя Flash Player, нажатия на спрограммированную кнопку, расположенную в фильме, выполнения сценария ActionScript и т. д.). Анимация монтажных линеек муви-клипов, расположенных в первом кадре основной монтажной линейки, будет выполняться вне зависимости от того, установлен данный флажок или нет;
  - **Loop** — при установке этого флажка воспроизведение основной монтажной линейки фильма будет выполняться циклически;
  - **Display menu** — в результате установки этого флажка при щелчке правой кнопкой мыши внутри окна Flash Player будет выведено контекстное меню, содержащее ряд команд, с помощью которых можно управлять воспроизведением основной монтажной линейки фильма и изменять масштаб просмотра содержимого данного окна. Если использование этих возможностей является нежелательным, флажок необходимо снять. Правда, при этом контекстное меню все равно будет появляться, но в нем будет присутствовать только один пункт — About Macromedia Flash Player 7;
  - **Device font** — установка данного флажка приводит к замене отсутствующих в системе пользователя шрифтов системными шрифтами. Данный параметр воздействует только на статический горизонтальный текст, для которого в рабочей среде был установлен режим **Use device fonts**.
- **Quality** — данный параметр позволяет задать соотношение между быстродействием и качеством прорисовки кадров. При включенном сглаживании одновременное выполнение нескольких сложных анимационных заданий на недостаточно мощном компьютере может привести к снижению скорости воспроизведения фильма. Данный параметр может принимать одно из следующих значений.
  - **Low** — приоритет полностью отдается быстродействию. В результате достигается максимальная скорость воспроизведения, но границы форм выглядят ступенчатыми и неровными вследствие полного отключения сглаживания.

- **Auto Low** — приоритет на стороне быстродействия, но, по возможности, качество прорисовки поддерживается на приемлемом уровне. В начале воспроизведения сглаживание отключено, однако если ресурсов процессора оказывается достаточно для более качественной прорисовки, сглаживание может быть включено.
- **Auto High** — в начале воспроизведения устанавливается равное соотношение между скоростью воспроизведения и качеством прорисовки кадров; сглаживание включено. Однако если ресурсов процессора недостаточно для поддержания требуемой скорости воспроизведения, сглаживание отключается.
- **Medium** — частичное сглаживание графики. Растровые изображения не сглаживаются вообще.
- **High** — приоритет отдается качеству прорисовки, сглаживание включено всегда. Если фильм не содержит анимации, растровые изображения сглаживаются; при наличии анимации сглаживание растровых изображений отключено.
- **Best** — приоритет полностью отдается качеству прорисовки. Полное сглаживание включено всегда вне зависимости от того, обеспечивается ли требуемая скорость воспроизведения или нет.

- **Window Mode** — данный список позволяет задать параметры размещения Flash-фильма на странице HTML относительно ее содержимого.
  - **Window** — Flash-фильм воспроизводится в своем собственном прямоугольном окне.
  - **Opaque Windowless** — окно Flash-фильма имеет непрозрачный фон и перекрывает другие элементы HTML-страницы.
  - **Transparent Windowless** — окно Flash-фильма имеет прозрачный фон, сквозь который просвечивает содержимое нижележащих элементов HTML. Использование данного параметра в сочетании с DHTML-слоями позволяет реализовать достаточно интересные анимационные и интерактивные эффекты, связанные с тем, что Flash-содержимое "плавает" над элементами HTML.

  Безоконный режим (Windowless) поддерживается только следующими версиями браузеров: Internet Explorer 5.0 и выше, Netscape Navigator 7.0 и выше, Opera 6 и выше, Mozilla 1.0 и выше, AOL/Compuserve.
- **HTML alignment** — данный список предназначен для задания параметров выравнивания окна Flash Player относительно границ окна браузера. Позиционирование Flash-фильма на странице имеет смысл осуществлять в соответствующем HTML-редакторе (например, Dreamweaver) или вручную, поскольку данный параметр не позволяет добиться желаемых результатов.
- **Scale** — данный список позволяет указать параметры масштабирования фильма внутри окна Flash Player. Различные варианты масштабирования имеют смысл в ситуации, когда размер окна и размер фильма не совпадают. Предлагаются следующие варианты масштабирования.
  - **Default (Show all)** — этот параметр используется по умолчанию. Фильм масштабируется пропорционально таким образом, чтобы все содержимое рабочей области отображалось в окне. При этом возможна ситуация, когда будет видна часть вспомогательной области.

- **No Border** — фильм масштабируется пропорционально таким образом, чтобы полностью заполнить все окно Flash Player. При этом возможна ситуация, когда часть фильма будет срезана границами окна Flash Player.
- **Exact Fit** — фильм масштабируется таким образом, чтобы полностью заполнить все окно Flash Player. Размер фильма подгоняется к размеру окна, при этом, если рабочая область фильма и окно имеют разные пропорции, фильм будет сжат или растянут.
- **No Scale** — масштабирования фильма не происходит; он отображается в окне Flash Player с теми же размерами, которые были заданы в рабочей среде.

- **Flash alignment** — два данных списка позволяют задать параметры выравнивания фильма внутри окна Flash Player по вертикали и по горизонтали соответственно.
- **Show warning messages** — вывод сообщений об ошибках, возникающих вследствие конфликта заданных параметров публикации.

Формат Windows Projector (exe)

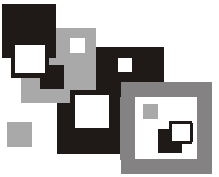
Проектор представляет собой исполняемый файл (с расширением exe), который включает в себя Flash-фильм, содержащий полную функциональность и автономный проигрыватель Flash Player, позволяющий воспроизвести данный фильм, даже если у пользователя не установлено программное обеспечение для просмотра Flash-содержимого. Размер проектора всегда больше размера соответствующего SWF-файла на тот объем, который занимает автономный проигрыватель. Для седьмой версии он составляет 980 Кбайт. Применение проектора чрезвычайно удобно для распространения Flash-проектов, предназначенных для локального просмотра, на электронных носителях или через сеть интранет (intranet). При работе с проектором существует возможность контролировать параметры воспроизведения фильма (качество, режим экрана и т. д.) и выполнять различные действия (закрывать проектор, запускать другой исполняемый файл, находящийся в системе и пр.) с помощью сценария ActionScript.

Публикация в данный формат не требует задания дополнительных параметров, необходимо только установить соответствующий флажок на вкладке **Formats**. Для создания проектора, предназначенного для просмотра на платформе Macintosh, необходимо установить флажок **Macintosh Projector**.

Примечание

Проектор также может быть создан непосредственно при воспроизведении SWF-файла из проигрывателя Flash Player с помощью команды **File>Create Projector**....

Глава 2



Новые возможности ActionScript

Машина должна работать, человек — думать.

Принцип IBM

В этой главе приведен краткий обзор основных нововведений, появившихся в ActionScript с выходом Flash MX 2004. Подробное описание перечисленных здесь элементов ActionScript и возможностей работы читатель сможет найти далее в соответствующих главах.

ActionScript 2.0

С выходом Flash MX 2004 язык сценариев ActionScript приобрел ряд новых возможностей, которые существенно повлияли на стиль программирования и подходы к структурированию исходного программного кода. Новая редакция ActionScript официально получила название ActionScript 2.0. Обновление ActionScript не привело к коренным изменениям языка, как это, например, произошло при выходе Flash 5, а просто обогатило его функциональными возможностями, обеспечивающими более полное соответствие стандарту ECMA 262 и практике объектно-ориентированного программирования в стиле Java. Несмотря на веские аргументы в пользу применения ActionScript 2.0 разработчик оставляет за собой право выбора стиля программирования. Flash MX 2004 позволяет указывать, какая версия языка используется при создании программного кода, включаемого в документ.

Зависимость от регистра

В предыдущих версиях ActionScript зависимость от регистра соблюдалась лишь частично. Так, идентификаторы `myVar` и `MyVar` были идентичны. ActionScript 2.0 является *регистрозависимым* языком. Это означает, что строчные и прописные (заглавные) буквы, используемые в идентификаторах, различаются.

Зависимость от регистра имеет место при публикации документа для проигрывателя Flash Player 7 (и выше) независимо от того, применяется ли ActionScript 1.0 или ActionScript 2.0. Для того чтобы исключить зависимость от регистра, можно публиковать документ для Flash Player 6 или ниже.

Строгий контроль типа данных

В ActionScript 2.0 появилась возможность строгой типизации переменных, параметров функции и возвращаемого функцией значения. Использование строгой типизации переменных приводит к тому, что переменная в момент создания получает определенный тип и в дальнейшем может содержать данные только этого фиксированного типа. Попытка поместить в строго типизированную переменную данные, тип которых отличен от типа переменной, приводит к ошибке. Ошибка несоответствия типа генерируется во время компиляции и может быть обнаружена при автоматической проверке синтаксиса.

Типизация формальных параметров функции позволяет на этапе компиляции обнаружить ошибки несоответствия типа, связанные с передачей функции фактических параметров, тип которых не соответствует типу, указанному в объявлении функции.

Типизация возвращаемого функцией результата позволяет обнаружить ошибку, возникающую в случае, когда функция возвращает значение неверного типа.

Применение строгой типизации не является обязательным, тем не менее это позволяет существенно снизить вероятность возникновения ошибок и делает программный код более ясным. Кроме того, применение строго типизированных идентификаторов позволяет в процессе создания программы с помощью встроенного редактора ActionScript воспользоваться всплывающими подсказками по коду, что значительно облегчает процедуру набора инструкций и позволяет контролировать правильность синтаксиса.

Формальный синтаксис классов

До появления ActionScript 2.0 пользовательские классы создавались непосредственно в FLA-файлах (или должны были включаться в них с помощью директивы `#include`) в виде конструкторов. Для реализации наследования необходимо было явным образом ссылаться на прототип конструктора.

В ActionScript 2.0 пользовательские классы создаются исключительно во внешних AS-файлах с применением нового формального синтаксиса. Так, например, объявление класса выполняется с помощью ключевого слова `class`, для создания подкласса применяется ключевое слово `extends`, статические члены класса создаются с помощью ключевого слова `static` и т. д. Файлы классов могут объединяться в пакеты, что позволяет исключить возможность конфликта имен и лучше структурировать систему классов. Формальный синтаксис классов позволяет реализовать различные режимы доступа к членам класса, которые могут быть объявлены как открытые или защищенные. Еще одним важным нововведением является возможность создания интерфейсов.

Несмотря на то, что ActionScript 2.0 предоставляет принципиально новый подход к созданию классов, внутренний механизм их функционирования остался прежним. В процессе публикации программные коды пользовательских классов включаются в результирующий SWF-документ и преобразуются к виду, в точности соответствующему коду, созданному в стиле ActionScript 1.0. Таким образом, классы ActionScript 1.0 и ActionScript 2.0 отличаются только на этапе разработки. Тем не менее новый подход значительно лучше соответствует практике объектно-ориентированного

программирования. И, несмотря на то, что он, возможно, не обладает всей гибкостью, которую дает стиль ActionScript 1.0, его использование делает процесс разработки более легким, ясным и логичным.

Новая архитектура компонентов

С выходом Flash MX 2004 технология создания компонентов претерпела существенные изменения. Отметим основные особенности новой архитектуры компонентов.

- Управляющие классы компонента создаются во внешних AS-файлах с применением формального синтаксиса классов.
- Определение параметров компонента осуществляется автоматически на основании тегов метаданных, включаемых разработчиком непосредственно в код управляющего класса компонента.
- Возможность использования режима предварительного просмотра (Live preview) теперь не требует создания специального SWF-документа и реализуется для любого экземпляра компонента автоматически.
- Компоненты хранятся в библиотеке в виде компилированных клипов, то есть SWF-документов, включающих в свой состав элементы, которые обеспечивают их функциональность и визуальное представление. Тот факт, что компонент, по сути, является SWF-документом, исключает возможность проникновения в него из рабочей среды, как это можно легко сделать при работе с компонентом, созданным на основе архитектуры Flash 6. Таким образом, все подробности внутренней реализации и графические элементы компонента оказываются скрытыми от пользователя.
- Готовые компоненты распространяются в виде архива в формате SWC, в состав которого входит сам компилированный компонент (SWF-файл) и ряд дополнительных файлов, используемых рабочей средой Flash для получения данных, описывающих характеристики компонента при его интегрировании в Flash-документ. Наряду с применением формата SWC для распространения компонентов также может быть использован формат MXP, применяемый для хранения расширений в различных продуктах Macromedia.

Массивы

Раньше методы `sort()` и `sortOn()` при вызове без параметров выполняли сортировку элементов массива только на основании их строкового представления и в возрастающем направлении. Для реализации других опций сортировки необходимо было использовать пользовательскую функцию сортировки. Теперь методы `sort()` и `sortOn()` поддерживают несколько встроенных опций сортировки. Для этих целей им передаются дополнительные числовые параметры. При сортировке может одновременно использоваться несколько опций (например, сортировка строк без учета регистра по убывающей). В классе `Array` появился ряд статических свойств, хранящих числовые значения, соответствующие опциям сортировки. Эти свойства могут передаваться в качестве параметров методам `sort()` и `sortOn()` вместо непосредст-

венной передачи чисел. Пользовательская функция сортировки может использоваться одновременно с применением встроенных опций сортировки.

Работа с клипами

В Flash MX 2004 появилось несколько новых возможностей для работы с клипами. Некоторые из них реализованы на уровне среды разработки, другие представляют собой новые программные инструменты класса `MovieClip`.

Связывание символа с классом

При использовании ActionScript 2.0 связывание символа с пользовательским классом может выполняться непосредственно в рабочей среде с помощью диалогового окна **Linkage Properties**. Для того чтобы связать символ с классом, теперь достаточно в диалоговом окне **Linkage Properties** указать идентификатор связи символа и путь к управляющему классу. После этого любой вновь созданный экземпляр этого клипа будет наследовать поведение, реализованное в связанном с ним пользовательском классе (так, как если бы он был порожден этим классом). Данная возможность позволяет реализовать функциональность пользовательского класса в экземпляре клипа без непосредственного применения метода `Object.registerClass()` (хотя именно он лежит в основе внутреннего механизма связывания клипа с классом с помощью окна **Linkage Properties**). Таким образом, исходный FLA-документ теоретически может быть вообще полностью лишен программного кода, и при этом результирующий SWF-документ будет обладать интерактивностью, обеспечиваемой только внешними пользовательскими классами, код которых будет включен в него на этапе компиляции.

Компилированные клипы

Теперь клипы в библиотеке можно хранить в скомпилированном виде. Скомпилированный клип представляет собой SWF-документ, включающий в свой состав элементы, которые обеспечивают его функциональность и визуальное представление. Тот факт, что компилированный клип является SWF-документом, исключает возможность доступа к его содержимому, как это можно легко сделать при работе с обычным символом. Компилированные клипы, в частности, используются для распространения компонентов. Поэтому все подробности внутренней реализации и графические элементы компонента скрыты от конечного пользователя.

Контроль заполнения уровней стека

В классе `MovieClip` появились два новых метода `getInstanceAtDepth()` и `getNextHighestDepth()`, связанных с контролем заполнения стека графических объектов.

Метод `getInstanceAtDepth()` позволяет определить, содержит ли заданный уровень стека экземпляр клипа, и, если уровень оказывается занят, возвращает ссылку на этот клип.

Метод `getNextHighestDepth()` позволяет определить ближайший уровень стека, на который должен быть помещен клип, для того, чтобы он располагался на переднем плане. Применение этого метода избавляет от необходимости рассчитывать номера уровней при динамическом создании экземпляров.

Свойство *_lockroot*

Еще одним полезным нововведением является свойство `_lockroot`, появившееся в классе `MovieClip`. Это свойство применяется при динамической загрузке внешних SWF-документов в клип и позволяет указывать, должен ли идентификатор `_root`, используемый в загружаемом документе, ссылаться на основную монтажную линейку загружаемого документа или же на основную монтажную линейку принимающего документа.

Метод *getSWFVersion()*

Новый метод, позволяющий определить номер версии проигрывателя Flash Player, для которой выполнялась публикация документа, содержащего вызвавший этот метод клип.

Работа с текстом

Существенные нововведения произошли в области работы с текстом. ActionScript обогатился новыми возможностями, позволяющими значительно рационализировать динамическую обработку, вывод и обновление текстового наполнения Flash-проекта.

Поддержка CSS (Cascading Style Sheets)

В Flash Player 7 включена частичная поддержка первой версии каскадных таблиц стилей — CSS 1. Это позволяет создавать таблицу стилей, представляющую собой набор правил, применяемых для форматирования текста, выводимого в динамическом (или пользовательском) текстовом блоке. Проигрыватель Flash Player 7 поддерживает лишь малую часть возможностей, предоставляемых спецификацией CSS1, однако даже эти весьма ограниченные возможности позволяют существенно оптимизировать процедуру форматирования и, что особенно важно, значительно упростить обновление и изменение параметров форматирования содержимого текстовых блоков.

Создание таблицы стилей и выполнение различных манипуляций с ее содержимым осуществляется с помощью нового встроенного класса `StyleSheet`. Экземпляр класса `StyleSheet` представляет таблицу стилей и хранит информацию о наборе правил форматирования, определяющих различные параметры форматирования текста, такие как, например, шрифт, его размер, цвет, начертание и др. Для связывания таблицы стилей с текстовым блоком используется новое свойство `textField` класса `TextField`.

Таблица стилей может формироваться непосредственно средствами ActionScript и включаться в исходный программный код FLA-документа или же создаваться как внешний CSS-документ, который динамически загружается в SWF-фильм в процессе его воспроизведения. Использование внешних таблиц стилей позволяет хранить всю информацию, описывающую форматирование текста вне Flash-документа, что позволяет чрезвычайно просто обновлять его внешний вид без необходимости редактирования исходных файлов. Применение CSS в сочетании с динамической загрузкой структурированного текста в формате XML существенно упрощает процесс разработки информационно-насыщенных Flash-проектов, требующих регулярного обновления текстового содержимого, поскольку дает возможность вынести все текстовое наполнение и правила его форматирования за пределы Flash-документа, который только исполняет роль текстового контейнера.

Возможность загрузки изображений и внешних SWF-документов в текстовый блок

Еще одной новой особенностью, связанной с применением динамического (и пользовательского) текста, является возможность загрузки в текстовый блок экземпляров символов из библиотеки документа, внешних файлов растровых изображений в формате JPEG и внешних SWF-документов. Для реализации этих возможностей в Flash Player включена поддержка нового HTML-тега `<img>`. По окончании загрузки изображение или фильм встраиваются прямо в текстовый блок, при этом автоматически выполняется обтекание внедренного объекта текстом.

Прокрутка текста с помощью колеса мыши

В Flash Player 7 для Windows включена возможность выполнения прокрутки содержимого выделяемого текстового блока с помощью колеса мыши. В классе `TextField` появилось встроенное свойство `mouseWheelEnabled`, позволяющее разрешить или запретить возможность прокрутки выделяемого текста с помощью колеса мыши.

Для контроля выполнения прокрутки колеса мыши используется обработчик события `onMouseWheel` класса `Mouse`, который также позволяет определить скорость выполнения прокрутки и целевой объект, над которым во время выполнения прокрутки находится курсор мыши.

Контроль статического текста

В седьмой версии Flash появились новые возможности, позволяющие контролировать статический текст, расположенный на монтажной линейке клипа. Эти возможности реализуются с помощью нового встроенного класса `TextSnapshot`. Созданный на основе данного класса объект `TextSnapshot` представляет собой "снимок" всего статического текста, расположенного на монтажной линейке клипа. Класс `TextSnapshot` содержит набор методов, позволяющих осуществлять поиск требуемого сочетания символов в статическом тексте, а также осуществлять выделение статического текста заданным цветом. Все операции, связанные с программным

контролем статического текста, не дают возможности его редактирования или управления параметрами форматирования, а предоставляют доступ исключительно для чтения.

Класс *MovieClipLoader*. Организация процесса загрузки

До появления седьмой версии Flash загрузка внешних SWF- и JPEG-файлов могла быть реализована исключительно с помощью методов `loadMovie()` и `unloadMovie()`, позволяющих загружать внешние документы на монтажные линейки клипов и удалять их оттуда, а также глобальной функции `loadMovieNum()`, выполняющей загрузку внешнего файла на заданный уровень стека документов. Эти средства вполне эффективны и часто применяются на практике, однако их использование в ряде случаев приводит к существенным трудностям, связанным с отсутствием адекватных возможностей контроля хода загрузки.

Новый встроенный класс `MovieClipLoader` предоставляет чрезвычайно удобные возможности детального контроля процесса загрузки внешних файлов в форматах SWF и JPEG. Важным преимуществом класса `MovieClipLoader` является наличие в нем набора обработчиков событий, позволяющих контролировать этапы и состояние процесса загрузки. Встроенные методы класса `MovieClipLoader` позволяют инициировать или прекратить процесс загрузки, получить сведения об ее состоянии, а также зарегистрировать объект-приемник на прием сообщений о событиях от объекта `MovieClipLoader` и исключить его из списка объектов-приемников. Еще одной полезной особенностью класса `MovieClipLoader` является возможность обнаружения ошибки, возникшей в процессе загрузки и препятствующей ее завершению.

Один и тот же объект `MovieClipLoader` может инициировать загрузку любого количества внешних документов. При этом ход загрузки для каждого из них можно контролировать с помощью объектов-приемников, получающих сведения от осуществляющего загрузку объекта `MovieClipLoader`. Это дает возможность создания универсального предзагрузчика, используя для его реализации единый программный модуль. Вместе с тем при необходимости можно для загрузки каждого документа использовать индивидуальный объект `MovieClipLoader`.

Класс *Sound*. Поддержка ID3-тегов второй версии

В Flash Player 7 включена поддержка ID3-тегов версий 2.3 и 2.4. Поскольку ID3-теги второй версии располагаются в начале файла, это дает возможность вывести их еще до окончания загрузки файла. Для контроля тегов второй версии в объект `id3`, представляющий собой свойство встроенного класса `Sound`, был добавлен целый ряд новых свойств. Кроме того, в классе `Sound` появился новый обработчик события `onID3()`, вызов которого выполняется сразу же после окончания загрузки каждого набора тегов, включенных в mp3-файл.

Динамическая загрузка видео

С выходом Flash MX 2004 у разработчиков появилась возможность осуществлять динамическую загрузку (progressive download) видео в формате FLV (Flash Video) в SWF-документ с помощью ActionScript. Для этого в SWF-документе создается видеообъект (экземпляр класса `Video`), представляющий собой контейнер, в который уже непосредственно в процессе воспроизведения документа загружается внешний FLV-файл. Открытие соединения для выполнения загрузки выполняется с помощью метода `connect()` класса `NetConnection`. Процесс загрузки и управление воспроизведением контролируются с помощью класса `NetStream`. Оба эти класса поддерживались предыдущей версией проигрывателя Flash Player 6 только при работе с Flash Communication Server. В Flash Player 7 теперь можно непосредственно использовать их для реализации динамической загрузки FLV-документов. Программный интерфейс класса `NetStream` позволяет контролировать состояние загрузки, регулировать размер буфера и степень его заполненности, а также управлять воспроизведением FLV-файла. Использование динамической загрузки видео имеет ряд существенных преимуществ по сравнению с его внедрением в исходный документ на стадии разработки.

- Объем SWF-документа определяется только его содержимым и никак не зависит от объема видеофайла, который будет в него загружаться.
- Воспроизведение динамически загружаемого видеофайла начинается, как только на диск будет записана порция информации, необходимая для начала воспроизведения.
- Внешний FLV-файл загружается, кэшируется и затем воспроизводится с локального диска пользователя. Поэтому для запуска воспроизведения не требуется, чтобы весь видеофайл целиком был загружен в память компьютера.
- Качество динамически загруженного видео выше, чем внедренного. Это объясняется тем, что кодек, осуществляющий импорт видео в Flash, обеспечивает более низкое качество по сравнению с кодеком, используемым при работе с динамически загруженным видео.

Управление контекстным меню проигрывателя

С выходом Flash MX 2004 у разработчиков появились новые инструменты, позволяющие очень гибко управлять контекстным меню проигрывателя Flash Player. Такими инструментами являются два встроенных класса `ContextMenu` и `ContextMenuItem`. Эти классы обеспечивают возможности включения или отключения отображения произвольного сочетания стандартных элементов меню, добавления в него пользовательских элементов и, что особенно ценно, связывания контекстного меню, включающего заданный набор команд (стандартных и/или пользовательских) с конкретным объектом, расположенным на сцене.

Для хранения данных о структуре контекстного меню применяется объект `ContextMenu`, порожденный классом `ContextMenu`. Пользовательское контекстное меню можно ассоциировать с любым экземпляром клипа, кнопки или текстового блока.

Связывание объекта ContextMenu (то есть конкретного меню с заданным набором элементов) с "материальным" объектом, расположенным на сцене, выполняется динамически с помощью нового свойства menu, которое добавлено в классы MovieClip, Button и TextField. Это обеспечивает большую гибкость, поскольку позволяет использовать для нескольких объектов фильма одинаковые контекстные меню или, наоборот, при необходимости выводить разные контекстные меню для одного и того же объекта.

Класс ContextMenuItem предоставляет возможности создания пользовательских элементов контекстного меню. Это позволяет вынести в контекстное меню некоторые функциональные элементы интерфейса, специфичные для данного проекта (например, добавить в контекстное меню ссылку на адрес электронной почты или команды управления звуковым сопровождением или анимацией конкретного объекта). При создании проекта для русскоязычной аудитории нелишним может оказаться замена стандартных элементов меню проигрывателя их эквивалентами на русском языке.

К сожалению, даже новые возможности управления контекстным меню не позволяют полностью запретить его вывод или исключить из него "любимые всеми" элементы **Settings...** и **About Macromedia Flash Player...**.

Объект *System.capabilities*

В объект System.capabilities был добавлен целый ряд новых свойств, позволяющих получить дополнительные сведения о системе пользователя.

Печать из фильма

В Flash MX 2004 появился новый класс PrintJob, обеспечивающий широкие и значительно более удобные возможности печати содержимого Flash-фильма, чем используемые для этих целей ранее функции print(), printAsBitmap(), printNum() и printAsBitmapNum().

При использовании данного класса пользователь задает все параметры печати в стандартном диалоговом окне свойств установленного у него в системе принтера, после чего эти параметры передаются объекту PrintJob, представляющему собой задание на печать. Используя сведения о произведенных пользователем настройках, можно при необходимости выполнить масштабирование и позиционирование отправляемых на печать графических элементов, чтобы лучше закомпоновать их на листе.

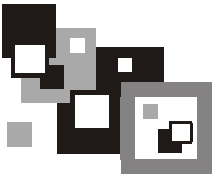
На печать одновременно может быть отправлено несколько кадров основной монтажной линейки или клипа. Причем для выполнения печати нескольких кадров не требуется помещать в них метки #p, как это необходимо было делать при работе с функциями print(), printAsBitmap(), printNum() и printAsBitmapNum(). При отправке кадра на печать можно задать печатаемую область и режим печати (растровый или векторный), используя предоставляемый классом PrintJob программный интерфейс.

Политика безопасности

В новой версии проигрывателя Flash Player 7 действуют более жесткие правила политики безопасности. Теперь для реализации взаимодействия между фильмами и для загрузки в фильм внешних данных требуется строгое соответствие доменов и протоколов. В предыдущей версии Flash Player 6 для реализации подобного взаимодействия достаточно было идентичности родительских доменов, а ограничения на соответствие протоколов вообще не существовало. Новые правила политики безопасности выражаются в следующих характерных особенностях.

- По умолчанию один документ может получить доступ к объектам и переменным другого документа только в том случае, если оба документа загружены с одного и того же домена. Для того чтобы разрешить доступ к объектам и переменным документа, расположенного в одном домене, со стороны фильма, загруженного из другого домена, необходимо воспользоваться методом `System.security.allowDomain()`.
- По умолчанию один документ может получить доступ к объектам и переменным другого документа только в том случае, если оба документа загружены с применением одного и того же протокола (HTTP или HTTPS). Для того чтобы разрешить доступ к файлам, использующим защищенный протокол HTTPS, документам, использующим незащищенный протокол HTTP, необходимо использовать новый метод `System.security.allowInsecureDomain()`.
- Локальное взаимодействие между документами по умолчанию разрешается только при условии полной идентичности их доменов. Для того чтобы разрешить взаимодействие между файлами, расположенными в разных доменах, необходимо воспользоваться обработчиком события `allowDomain` класса `LocalConnection`.
- Файл, загруженный с сайта, использующего протокол HTTP, не сможет взаимодействовать с файлом, загруженным с сайта по защищенному протоколу HTTPS. Для преодоления этого ограничения можно воспользоваться обработчиком события `allowInsecureDomain` класса `LocalConnection`.
- Импорт внешних данных (XML-документов, текстовых документов, ресурсов общих библиотек) может быть непосредственно осуществлен только при условии, что фильм-получатель и файл, содержащий импортируемые данные, находятся в одном и том же домене. Для того чтобы получить возможность осуществлять загрузку внешних ресурсов, расположенных в другом домене, необходимо, чтобы в этом домене находился специальный XML-документ с описанием междоменной политики безопасности, определяющий, каким доменам может быть предоставлена такая возможность.

Глава 3



Введение в ActionScript. Среда разработки

Как только вы испробуете все возможные способы решения
и не найдете подходящего, тут же найдется решение простое
и очевидное для всех других людей.

Уравнение Смэйфу

Представление о сценариях ActionScript

ActionScript представляет собой объектно-ориентированный язык программирования, интегрированный в среду Macromedia Flash, который предназначен для решения многообразных задач, встающих перед разработчиком в процессе создания Flash-проектов. Этот язык был разработан на основе стандарта ECMA — 262 и с появлением новых версий Flash постепенно модернизировался, обогащался новыми возможностями, направленными на более близкое соответствие данному стандарту. Разработчики, имеющие опыт работы с языком JavaScript, несомненно, обнаружат, что ActionScript очень близок к своему "старшему брату". Тем не менее ActionScript имеет ряд специфических особенностей, которые обусловлены его прикладной функцией. Это инструмент, адаптированный для применения в среде Flash, который позволяет снабдить Flash-фильм интеллектом, заставить его взаимодействовать с пользователем и обеспечить его динамический контроль.

Одной из наиболее специфичных особенностей ActionScript является возможное рассредоточение программного кода по различным структурным элементам документа. Это зачастую приводит к тому, что весь программный код невозможно представить в виде единого листинга, поскольку его различные фрагменты тесно связаны и функционируют в пределах области их размещения. Объединение таких разрозненных фрагментов в единый текст, скорее всего, приведет к утрате требуемой функциональности и зачастую к потере логики программы. Это можно сделать только в целях наглядности (например, при демонстрации кода в учебных целях) и обязательно снабдив соответствующие фрагменты пояснительными комментариями, указывающими их местоположение в структуре документа. Таким образом, характерной чертой сценариев ActionScript является их *тесная интеграция в структуру документа*.

Программный код, как и обычный текст, состоит из предложений. Для создания сценария можно использовать встроенный редактор ActionScript, который кроме возможности непосредственного ввода текста программы, предоставляет удобный сервис в работе с программным кодом. При необходимости для создания сценария также можно воспользоваться любым внешним текстовым редактором или специализированным редактором, например, Macromedia Dreamviewer.

Сценарий может быть непосредственно помещен в исходный документ FLA, либо его можно оформить в виде отдельного файла с расширением as. Файл AS представляет собой текстовый документ, использующий Unicode, кодировку UTF-8. Такой документ может быть просмотрен и отредактирован в любом текстовом редакторе, работающем с кодированным текстом.

При публикации исходного FLA-документа код из внешнего AS-файла встраивается в него и компилируется в результирующий SWF-документ.

Использование редактора ActionScript для создания сценариев

Для создания сценариев и работы с ними непосредственно в рабочей среде FLA-документа Flash служит панель **Actions**. Кроме предоставления возможности набора текста программы панель **Actions** позволяет автоматизировать процесс набора некоторых команд, выполнять форматирование программного кода, осуществлять автоматическую проверку синтаксиса, поиск и замену символов, быстро переключаться между фрагментами кода, распределенными по разным элементам структуры Flash-фильма. Эти многочисленные возможности делают процесс создания сценария более комфортным и контролируемым.

В случае создания файла AS (ActionScript File) работа ведется в автономном редакторе ActionScript, интегрированном в рабочую среду Flash и представляющем собой усеченный вариант панели **Actions**. Автономный редактор сценариев доступен только во Flash MX 2004 Professional. При работе с базовой версией (Flash MX 2004) для создания внешнего AS-файла можно воспользоваться обычным текстовым редактором или экспортировать сценарий из панели **Actions**.

С выходом седьмой версии Flash язык ActionScript также претерпел определенные изменения, нашедшие отражение в его новой редакции ActionScript 2.0. ActionScript, используемый в предыдущих версиях, теперь официально называется ActionScript 1.0. В рабочей среде Macromedia Flash MX 2004 можно работать, используя как ActionScript 1.0, так и ActionScript 2.0. Для того чтобы указать версию, применяемую в документе, необходимо выполнить одно из следующих действий.

- Выполнить команду главного меню **File>Publish Settings...** и во вкладке **Flash** одноименного диалогового окна выбрать используемую версию языка в списке **ActionScript version**.
- Снять выделение со всех объектов сцены и нажать кнопку **Settings...**, расположенную в панели **Properties**. Затем во вкладке **Flash** диалогового окна **Publish Settings** выбрать используемую версию языка в списке **ActionScript version**.

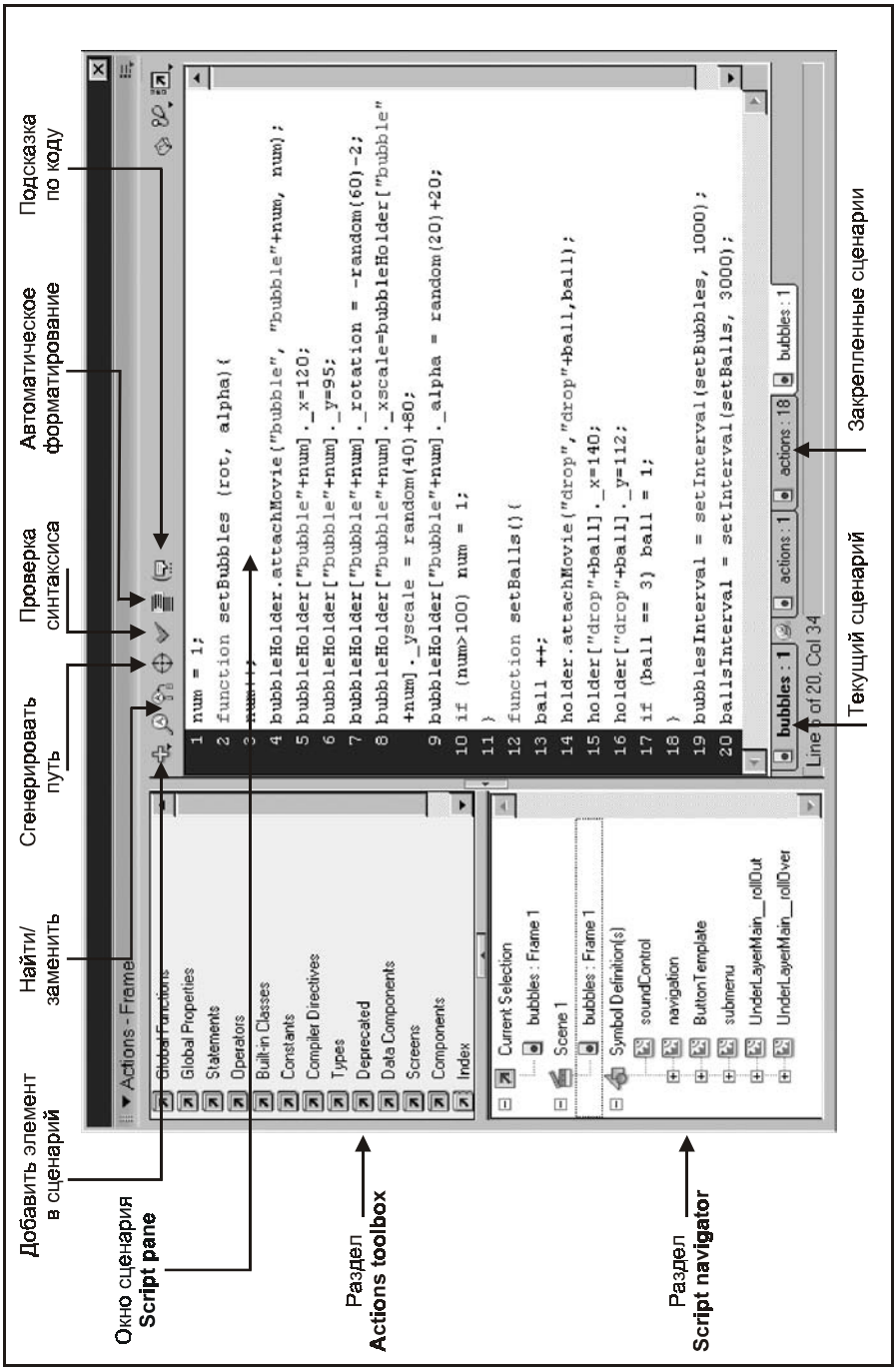


Рис. 3.1. Панель редактора ActionScript **Actions**

Используемая в документе версия языка указывается в панели **Properties** при отсутствии выделенных объектов. Несмотря на то, что ActionScript 2.0 появился только с выходом Flash MX 2004 и Flash Player 7, его также можно использовать и при публикации проекта для более ранней версии проигрывателя Flash Player 6. Это связано с тем, что различия между ActionScript 1.0 и ActionScript 2.0 существуют только на этапе разработки. После компиляции код, созданный с помощью ActionScript 2.0, ничем не отличается от кода, созданного с помощью ActionScript 1.0.

Для вызова панели **Actions** необходимо выполнить команду главного меню **Window>Development Panels>Actions** или воспользоваться ее клавиатурным эквивалентом — клавишей <F9>. Внешний вид панели **Actions** представлен на рис. 3.1.

Панель **Actions** состоит из трех разделов.

- **Script pane** (Окно сценария) — окно, расположенное в правой части панели **Actions**, предназначенное для ввода и отображения сценария.
- **Actions toolbox** — данный раздел расположен слева в верхней части панели **Actions** и содержит перечень всех встроенных классов, задокументированных методов, свойств, функций, предложений, операторов и т. д., то есть элементов языка ActionScript. Все элементы разбиты на категории и иерархически упорядочены с помощью папок.
- **Script navigator** — данный раздел расположен слева в нижней части панели **Actions** и содержит схематичное отображение элементов структуры документа, содержащих сценарии.

Для того чтобы скрыть левую часть, содержащую разделы **Actions toolbox** и **Script navigator**, для увеличения размера окна сценария, нужно щелкнуть на пиктограмме треугольника, расположенной посередине на вертикальной линии, разделяющей левую и правую части панели **Actions**. Аналогичным образом можно скрыть раздел **Actions toolbox**, оставив при этом разделы **Script navigator** и **Script pane**.

Раздел *Script pane* (Окно сценария)

Работа с текстом программного кода в окне сценария осуществляется так же, как и с обычным текстом в текстовом редакторе. Для настройки параметров редактора сценариев служит вкладка **ActionScript** диалогового окна **Preferences**, вызвать которое можно либо выполнив команду главного меню **Edit>Preferences...**, либо с помощью команды **Preferences...** контекстного меню самой панели **Actions** (рис. 3.2).

Работа с текстом программы

Шрифт и кегль, с помощью которого отображаются команды сценария, задаются в разделе **Text**. По умолчанию используется шрифт Courier New. Такой выбор является наиболее предпочтительным, поскольку применение моноширинного шрифта повышает читабельность текста.

Фрагменты сценария могут быть выделены, скопированы, вставлены из буфера или удалены. Эти действия выполняются с помощью команд контекстного меню, которое появляется при щелчке правой кнопкой мыши внутри раздела **Script pane**, или с помощью стандартных клавиатурных сочетаний (<Ctrl>+<A>, <Ctrl>+<C>, <Ctrl>+<V>, <Ctrl>+<X>, <Delete>). Для того чтобы выделить целую строку сцена-

рия, можно щелкнуть на синей полосе слева от требуемой строки, удерживая при этом клавишу <Ctrl>. Для того чтобы выделить сразу несколько строк или многострочный фрагмент кода, можно воспользоваться клавишей <Shift>.

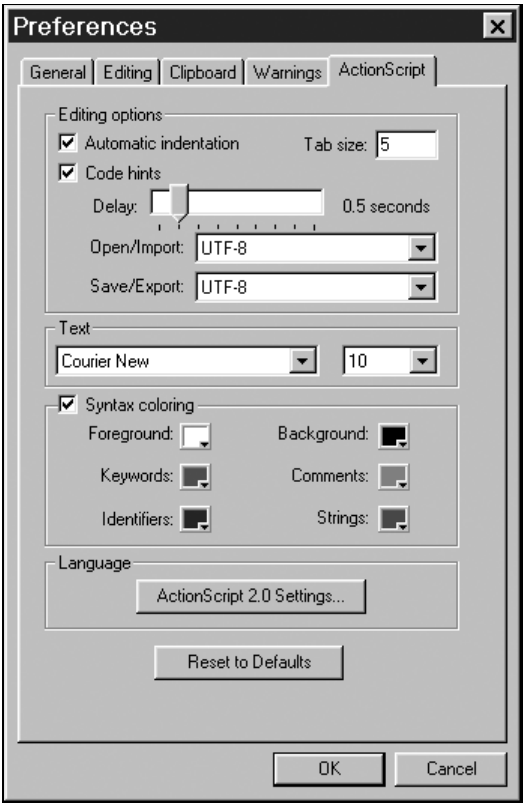


Рис. 3.2. Вкладка **ActionScript** диалогового окна **Preferences**

При необходимости увеличения отступа абзаца сразу для нескольких последовательно расположенных строк кода нужно выделить эти строки и нажать клавишу <Tab>. Для того чтобы сместить несколько последовательно расположенных строк кода влево, уменьшая отступ абзаца, нужно выделить эти строки и нажать сочетание клавиш <Shift>+<Tab>. Величина абзацного отступа задается на вкладке **ActionScript** диалогового окна **Preferences** в поле **Tab size**.

Установка флажка **Automatic indentation** приводит к тому, что текст, который набирается с новой строки после символов (или {, располагается с отступом. Эта опция включена по умолчанию. Ее использование позволяет визуально упорядочить код для лучшей читабельности. Используя отступ, можно зрительно отделить, например, тело цикла или блок условного предложения от остального набора команд сценария.

Зачастую предложения бывают достаточно длинными. Для того чтобы исключить необходимость горизонтальной прокрутки содержимого окна сценариев, можно использовать автоматический перенос слов. Для включения этого режима можно выполнить одно из следующих действий.

- Выполнить команду контекстного меню панели **Actions Word Wrap**.
- Щелкнуть на пиктограмме , расположенной справа над окном сценариев, и из всплывающего меню выбрать команду **Word Wrap**.
- Нажать сочетание клавиш <Ctrl>+<Shift>+<W>.
- При работе в автономном редакторе ActionScript выполнить команду главного меню **View>Word Wrap**.

Важно иметь в виду, что видимый перенос части строки вследствие использования режима **Word Wrap** не приводит к реальному переносу. Таким образом, применение данного режима не разбивает исходное предложение.

Если текст программы состоит из большого числа предложений, бывает удобно включить нумерацию строк. Для того чтобы это сделать, можно выполнить одно из следующих действий.

- Выполнить команду **View Line Numbers** контекстного меню панели **Actions**.
- Щелкнуть на пиктограмме , расположенной справа над окном сценариев, и из всплывающего меню выбрать команду **View Line Numbers**.
- Нажать сочетание клавиш <Ctrl>+<Shift>+<L>.
- При работе в автономном редакторе ActionScript выполнить команду главного меню **View>View Line Numbers**.

Для того чтобы быстро переместиться к требуемой строке кода, можно выполнить команду **Go to Line...** контекстного меню панели **Actions** или воспользоваться сочетанием клавиш <Ctrl>+<G>. При работе в автономном редакторе используется команда главного меню **View>Go to Line...**.

Редактор ActionScript обладает функциями поиска и/или замены требуемого символа или сочетания символов. Для их реализации можно воспользоваться следующими возможностями.

- Для поиска требуемого сочетания символов можно выполнить команду **Find** (Найти) контекстного меню панели **Actions** или щелкнуть на пиктограмме . Также можно воспользоваться клавиатурным эквивалентом данной команды <Ctrl>+<F>. В появившемся диалоговом окне необходимо указать искомую строку. Флажок **Match case** позволяет осуществлять поиск с учетом регистра.
- Для повторного поиска сочетания символов, заданного при предыдущем поиске, нужно выполнить команду **Find Again** (Найти снова) контекстного меню панели **Actions** или воспользоваться ее клавиатурным эквивалентом <F3>.
- Для поиска заданного сочетания символов и замены его на другое служит команда **Replace** (Заменить), ее клавиатурный эквивалент <Ctrl>+<H> или пиктограмма , расположенная слева над окном сценариев.
- Редактор ActionScript предоставляет возможности экспорта и импорта сценария.

- Для того чтобы экспортировать текущий сценарий, отображаемый в панели **Actions**, нужно выполнить команду контекстного меню **Export Script...** или воспользоваться ее клавиатурным эквивалентом <Ctrl>+<Shift>+<X>. Сценарий будет сохранен во внешнем файле с расширением as.
- Для осуществления импорта сценария из внешнего AS-файла в текущий код, необходимо выполнить команду **Import Script...** контекстного меню панели **Actions** или воспользоваться ее клавиатурным эквивалентом <Ctrl>+<Shift>+<I>. Импортированный фрагмент кода будет вставлен в текущий сценарий, начиная с позиции, в которой находится курсор. При этом исходный код в окне сценариев, остается без изменений.
- Кодировка, используемая при импорте и экспорте сценариев, может быть указана во вкладке **ActionScript** диалогового окна **Preferences** (рис. 3.2).
- Команда **Print** (Печать) контекстного меню панели **Actions** позволяет вывести сценарий на печать.

Подсветка синтаксиса

Одной из удобных возможностей редактора ActionScript является режим **Syntax coloring** (Подсветка синтаксиса). Применение данного режима позволяет разметить элементы программного кода разными цветами, что существенно облегчает его восприятие и ориентацию в нем. Для включения этого режима необходимо установить флажок **Syntax coloring** на вкладке **ActionScript** диалогового окна **Preferences** (рис. 3.2). Здесь же можно указать, каким именно цветом должны отображаться предлагаемые элементы:

- **Background** — цвет фона (по умолчанию — белый);
- **Keywords** — ключевые слова; это группа идентификаторов, зарезервированных для использования компилятором (по умолчанию — синий). К ключевым словам относятся, например, `var` (предложение объявления переменной), `if` (условное предложение), `for` (предложение цикла), `function` (предложение объявления функции) и т. д.;
- **Comments** — комментарии (по умолчанию — серый). Любой набор символов, находящийся после сочетания `//`, или заключенный между символами `/*` и `*/`;
- **Identifiers** — идентификаторы (по умолчанию — синий). К их числу относятся все встроенные глобальные функции (например, `getURL()`), классы (например, `MovieClip`), их методы (например, `gotoAndStop()`) и свойства (например, `_alpha`);
- **Strings** — строки (по умолчанию — голубой). Строкой является произвольный набор символов, заключенный в одинарные или двойные кавычки (например, `"Dmitri I. Albert"` или `'Elena. E. Albert'`);
- **Foreground** (по умолчанию — черный) — все остальные слова и символы. Сюда относятся операторы, пользовательские переменные, функции, классы, методы и свойства.

Применение режима **Syntax Coloring** позволяет не только легче ориентироваться в программном коде, но также способствует снижению количества синтаксических

ошибок. Если при вводе стандартной команды, принадлежащей к одной из подсвечиваемых категорий, перечисленных ранее, она не меняет цвет, это свидетельствует об ошибке в написании. Следует помнить, что ActionScript 2.0 — регистрозависимый язык, поэтому редактор "узнает" стандартный метод `createEmptyMovieClip()` и подсветит его цветом, заданным для категории **Identifiers**, но не сможет "опознать" пользовательский метод `CreateEmptyMovieclip()`, который будет подсвечен цветом, заданным для категории **Foreground**.

Автоматическое форматирование

Редактор ActionScript позволяет выполнить автоматическое форматирование текста программы. Эта процедура может быть выполнена, только если код не содержит синтаксических ошибок. Для автоматического форматирования можно выполнить одно из следующих действий.

- Выполнить команду контекстного меню панели **Actions Auto Format**.
- Щелкнуть на пиктограмме , расположенной слева над окном сценариев.
- Нажать сочетание клавиш <Ctrl>+<Shift>+<F>.
- При работе в автономном редакторе ActionScript выполнить команду главного меню **Tools>Auto Format**.

В результате применения автоматического форматирования весь текст сценария будет единообразно отформатирован согласно текущим установкам режима **Auto Format**. Для настройки параметров режима автоматического форматирования нужно выполнить команду контекстного меню панели **Actions Auto Format Options...**. При работе в автономном редакторе ActionScript нужно выполнить команду главного меню **Edit>Auto Format Options...**. Появившееся вслед за этим диалоговое окно **Auto Format Options** позволяет определить, как будут форматироваться представленные здесь конструкции (рис. 3.3). Результат можно сразу же оценить на примере кода, представленного в окне **Preview**. Диалоговое окно **Auto Format Options** предлагает следующие возможности автоматического форматирования.

- **'{' on line after if, for, switch, while, etc** — переносить открывающую фигурную скобку в условных предложениях, циклах, предложениях `with` на следующую отдельную строку.
- **'{' on line after function, class and interface keywords** — переносить открывающую фигурную скобку в объявлениях функций, классов и интерфейсов на следующую отдельную строку.
- **Don't cuddle '}' and else** — не размещать закрывающую фигурную скобку в одной строке с предложением `else`. При установке данной опции закрывающая фигурная скобка размещается отдельной строкой.
- **Space after function name in function calls** — отделять пробелом имя функции от ее аргументов, заключенных в круглые скобки, при вызове функции.
- **Spaces around operators** — размещать символ пробела слева и справа от операторов.

Кнопка **Reset to Defaults** восстанавливает настройки автоформатирования, используемые по умолчанию. При этом все флажки снимаются.

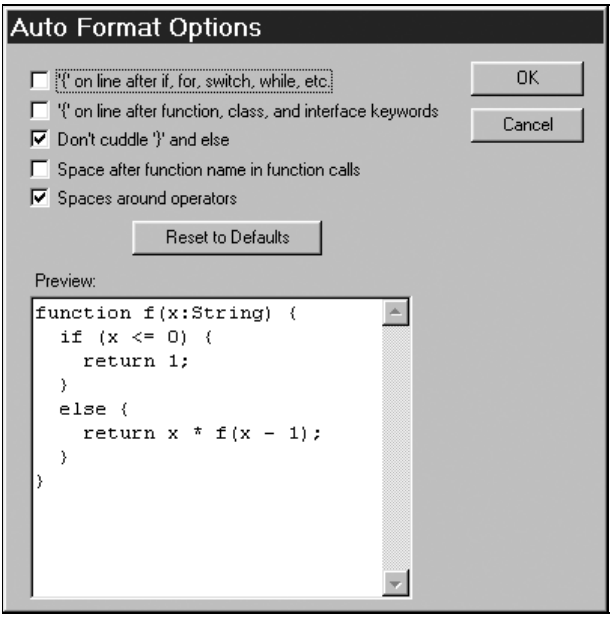


Рис. 3.3. Диалоговое окно **Auto Format Options**

Автоматическая проверка синтаксиса

Чрезвычайно удобной возможностью редактора ActionScript является автоматическая проверка синтаксиса. Эта опция действует наподобие функции проверки орфографии, поддерживаемой многими текстовыми редакторами (и, в частности, включенной в пакет Macromedia Flash MX 2004 Professional). Проверка синтаксиса позволяет обнаружить ошибки, возникающие вследствие неправильного набора команд, отсутствия требуемых символов или операторов (например, несовпадения количества открывающих и закрывающих скобок), указания неправильного количества или типа аргументов, несоответствия типов данных и др. При этом важно иметь в виду, что автоматическая проверка обнаруживает только синтаксические ошибки и не способна выявить ошибки в логике или алгоритме программы.

Для выполнения проверки синтаксиса можно выполнить одно из следующих действий.

- Выполнить команду контекстного меню панели **Actions Check Syntax**.
- Щелкнуть на пиктограмме , расположенной слева над окном сценариев.
- Нажать сочетание клавиш <Ctrl>+<T>.
- При работе в автономном редакторе ActionScript выполнить команду главного меню **Tools>Check Syntax**.

Если программный код не содержит синтаксических ошибок, то в результате проверки выводится сообщение **The script contains no errors** (Сценарий не содержит

ошибок). Если же проверка обнаруживает ошибку, то при этом автоматически открывается панель **Output**, в которой указывается адрес и содержание ошибки (рис. 3.4). Сообщение о каждой встреченной ошибке начинается со слова `**Error**`. Далее следует адрес, то есть местоположение кода в документе (адрес включает указание сцены, слоя и кадра), и строка, содержащая ошибку, например: `Scene=Scene 1, layer=Layer 1, frame=1:Line 1`. Далее следует описание ошибки и приводится фрагмент кода, в котором встретилась ошибка. Сообщение о каждой следующей ошибке также начинается со слова `**Error**` и выводится отдельной строкой. В конце отчета об ошибках приводится информация об их общем количестве. Зачастую первая ошибка может повлечь за собой целую цепь последующих ошибок, вследствие чего отчет может состоять из большого числа записей. В связи с этим, анализировать сообщение об обнаруженных в тексте программы ошибках необходимо, начиная с самого верха. Так, например, ошибочное использование запятых вместо точки с запятой в заголовке цикла, представленном далее, приводит к трем сообщениям об ошибках (см. рис. 3.4). В связи с тем, что заголовок цикла оформлен неверно, предложение, помещенное в тело цикла, интерпретируется как ошибочное, и закрывающая фигурная скобка также вызывает ошибку, как не имеющая открывающей пары.

```
for (i=1, i<10, i++){
this["x0"+i]=i*5;
}
```

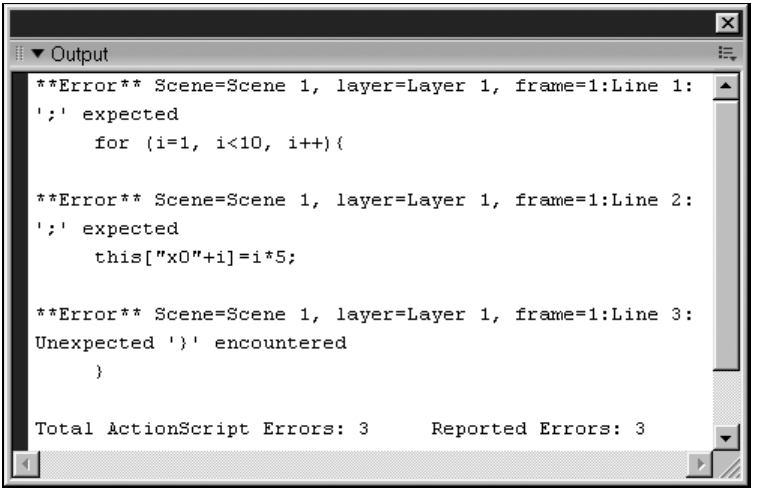


Рис. 3.4. Панель **Output**. Сообщение об обнаружении ошибок

Этот простой пример иллюстрирует механизм автоматической проверки синтаксиса. Очень часто исправление самой первой встреченной ошибки приводит к устранению всех остальных ошибок цепи.

Примечание

При написании большого фрагмента кода частой причиной ошибок может быть несоответствие количества открывающих и закрывающих скобок (круглых, квадратных или фигурных). Для того чтобы проверить, имеет ли открывающая скобка соответствующую ей закрывающую, можно воспользоваться следующим методом. Необходимо поместить курсор внутрь скобок, соответствие которых должно быть проверено, и нажать сочетание клавиш <Ctrl>+<'>. В результате этого действия весь текст, находящийся внутри скобок, будет выделен, что позволит оценить правильность их расстановки.

Всплывающие подсказки

Еще одной удобной возможностью редактора ActionScript являются всплывающие подсказки по коду (code hints). Подсказки автоматически появляются в процессе ввода текста сценария (если данная опция не отключена) и избавляют разработчика от необходимости вводить вручную определенные элементы команд. Кроме того, подсказки указывают синтаксис возможных вариантов применения стандартных команд непосредственно в процессе их ввода.

Редактор ActionScript выводит два типа подсказок: *всплывающая* подсказка по синтаксису используемой языковой конструкции или элемента (tooltip style) и подсказка *в виде меню* (menu style). Подсказка по синтаксису появляется при вводе открывающей круглой скобки, размещенной после элемента сценария, требующего наличия круглых скобок (условное предложение, предложение цикла, функция и т. д.). Всплывающая подсказка указывает возможные синтаксические варианты применения данного элемента. При наличии нескольких вариантов подсказка содержит стрелки, позволяющие просмотреть различные схемы его использования (рис. 3.5). Такая возможность бывает очень удобна при необходимости применения малознакомых или редко используемых команд.

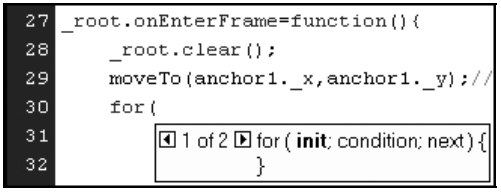


Рис. 3.5. Всплывающая подсказка по синтаксису

Подсказка в виде меню появляется при вводе точки после имени переменной, тип которой был явно указан при ее объявлении, при вводе точки после имени стандартного объекта, после ввода открывающей круглой скобки обработчика событий, после двоеточия в объявлении переменной или функции. Подсказка в виде меню позволяет выбрать из предлагаемого списка один из требуемых в данном контексте элементов языка (рис. 3.6). Это может быть тип данных, доступный для данного объекта метод или свойство, событие, контролируемое используемым обработчиком. Подсказка в виде меню позволяет автоматически вставить требуемый элемент в код и сократить время набора элементов стандартных команд.

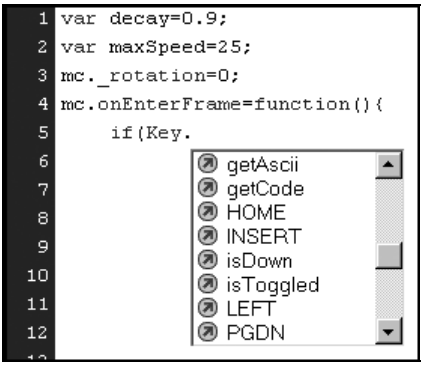


Рис. 3.6. Подсказка по коду в виде меню

Подсказки обоих типов выводятся только в том случае, если используемый синтаксис соответствует задокументированному (в частности, подсказки чувствительны к регистру). Настройка параметров отображения подсказок осуществляется во вкладке **ActionScript** окна **Preferences** (см. рис. 3.2).

- **Code hints** — установка этого флажка включает всплывающие подсказки. Если флажок снят, всплывающие подсказки не отображаются.
- **Delay** — этот параметр позволяет указать время, в течение которого будет выведена всплывающая подсказка. Диапазон значений изменяется от 0 до 4 секунд. При установке значения 0 подсказка появляется мгновенно после ввода команды, предполагающей появление подсказки.

Для того чтобы скрыть всплывающую подсказку по синтаксису, можно выполнить одно из следующих действий:

- закрыть круглую скобку;
- щелкнуть на другом участке кода;
- нажать клавишу <Escape>.

Для того чтобы скрыть всплывающую подсказку в виде меню, можно выполнить одно из следующих действий:

- выбрать одно из предлагаемых в меню значений;
- щелкнуть на другом участке кода;
- если подсказка появилась после открытия круглой скобки (например, в обработчике событий), закрыть круглую скобку;
- нажать клавишу <Escape>.

Для того чтобы отобразить подсказку по требованию, необходимо поместить курсор в соответствующий участок кода и щелкнуть на пиктограмме , расположенной слева над окном сценариев.

Контекстная справка

Macromedia Flash MX 2004 Professional поставляется с достаточно полной справочной системой, содержащей множество полезных сведений по работе с ActionScript. Справочная информация регулярно пополняется и обновляется. Обновление доступно через Интернет.

Редактор ActionScript имеет возможность вывода справки по требуемому элементу сценария. Для этого необходимо щелкнуть по нему правой кнопкой мыши и в контекстном меню выбрать команду **View Help** либо воспользоваться пиктограммой , расположенной справа над окном сценариев.

Для вызова справочной системы по языку можно воспользоваться командой главного меню **Help>ActionScript Dictionary**.

Раздел *Actions toolbox*

Здесь представлены все описанные в стандартной документации элементы языка ActionScript 2.0. Они разделены на тематические группы с помощью папок. Папки отображаются с помощью пиктограммы , а расположенные в них элементы с помощью пиктограммы . Щелчок по папке раскрывает ее, открывая доступ к содержимому. Зачастую во внешней папке могут находиться вложенные папки, открыть которые, в свою очередь, можно, щелкнув по ним. Раздел **Actions toolbox** включает следующие категории элементов.

- **Global Properties** — глобальные свойства и идентификаторы. К их числу относятся, например, следующие: `_global`, `_root`, `_parent`, `this` `_quality` и др.
- **Statements** — предложения, используемые при создании классов (`import`, `class`, `extends`, `private`, `public`, `static` и др.), предложения циклов (`for`, `while` и др.), условные предложения (`if`, `else`, `switch`), предложения, используемые в объявлениях функций (`function`, `return`) и переменных (`var`, `set()` и др.), и др.
- **Operators** — операторы. Сюда входят операторы сравнения, арифметические, логические, побитовые и другие операторы.
- **Built-in Classes** — встроенные классы. Это один из наиболее важных и объемных разделов. Здесь представлены все встроенные классы и определяемые в них методы, свойства и события, доступные для экземпляров этих классов.
- **Constants** — константы. Сюда относятся зарезервированные слова, используемые для обозначения определенного состояния или значения (`true`, `false`, `undefined`, `infinity` и др.).
- **Compiler Directives** — ключевые слова, указывающие компилятору на необходимость загрузки внешнего кода или управляющие последовательностью выполнения различных блоков кода (`#include`, `#initclip`, `#endinitclip` и др.).
- **Types** — перечень встроенных типов данных, которые могут быть использованы при указании типа объявляемой переменной, типа возвращаемого функцией значения и т. д.
- **Deprecated** — устаревшие элементы языка, используемые в более ранних версиях Flash. Их применение может быть оправдано при необходимости обеспечения

обратной совместимости проекта при публикации для более ранних версий проигрывателя Flash Player и в некоторых случаях для увеличения скорости работы проигрывателя. В остальных случаях использование вышедших из употребления элементов не рекомендовано.

- **Data Components** — перечень методов, свойств и событий компонентов, обеспечивающих контроль и обработку данных (Data Components).
- **Screens** — перечень методов, свойств и событий, применяемых при работе со слайдами (Slides) и формами (Forms).
- **Components** — перечень методов, свойств и событий компонентов пользовательского интерфейса (UI Components) и медиа-компонентов (Media Components).
- **Index** — глобальный перечень всех элементов языка, перечисленных в алфавитном порядке. Для того чтобы определить, к какой категории относится тот или иной элемент из глобального перечня **Index**, нужно щелкнуть по нему правой кнопкой мыши и выполнить команду контекстного меню **Show original**.

Для получения справки по элементу, открытому в разделе **Actions toolbox**, можно щелкнуть на нем правой кнопкой мыши и выполнить команду контекстного меню элемента **View Help**.

Двойной щелчок по элементу в разделе **Actions toolbox** перемещает его в окно сценария. Этот результат можно получить и с помощью контекстного меню элемента **Add to script**.

Для быстрого ввода некоторых инструкций можно воспользоваться соответствующим им сочетанием клавиш с использованием клавиши <Esc>. Для того чтобы отобразить соответствующие клавиатурные эквиваленты в разделе **Actions toolbox**, можно воспользоваться командой **View Esc Shortcut Keys**, вызвать которую можно, щелкнув на пиктограмме , расположенной над окном сценариев или с помощью контекстного меню панели **Actions**. Так, например, в результате нажатия сочетания клавиш <Esc>+<I>+<F> в окне сценариев будет сгенерировано условное предложение:

```
if () {
}
```

При этом курсор будет помещен внутрь круглых скобок. Нажатие сочетания клавиш <Esc>+<F>+<R> поместит в окно сценариев предложение цикла:

```
for () {
}
```

При этом курсор будет помещен внутрь круглых скобок заголовка цикла. Естественно, далеко не все команды имеют клавиатурный эквивалент, однако использование горячих клавиш для наиболее часто используемых конструкций позволяет сократить время при наборе текста программы. Перечень горячих клавиш для инструкций ActionScript содержится в *приложении 4*.

В седьмой версии пакета Macromedia Flash MX 2004 редактор ActionScript больше не имеет режима **Normal**, позволяющего формировать текст программы, не набирая его целиком вручную. Такой подход был чрезвычайно долгим, неэффективным и не годился для работы над более или менее сложными и большими сценариями. Вме-

сто этого теперь во Flash появилась новая концепция использования готовых фрагментов кода — *поведений* (behaviors), позволяющих реализовать стандартные элементы функциональности. Тем не менее при необходимости любой элемент языка может быть автоматически помещен в текст программы с помощью кнопки , расположенной слева над окном сценариев. Щелчок на этой кнопке приводит к появлению списка, содержащего перечень категорий, представленных в разделе **Actions toolbox**. Щелчок на элементе автоматически генерирует соответствующий фрагмент кода и помещает его в текст сценария.

Раздел *Script navigator*

Данный раздел напоминает проводник фильма **Movie Explorer**. Здесь в виде иерархического дерева графически представлена структура размещения сценариев в пределах текущего документа. Поскольку сценарии ActionScript могут быть рассредоточены по различным сценам, слоям, кадрам и объектам, поиск фрагмента кода, требующего редактирования, и переход к нему иногда могут вызывать определенные трудности. **Script navigator** позволяет зрительно представить структуру программного кода и, что особенно важно, быстро переместиться к любому его фрагменту без необходимости добираться до него вручную.

Информация, представленная в разделе **Script navigator**, подразделяется на три категории:

- **Current Selection**  — отображает сценарий текущего выделенного элемента (кадра, клипа или кнопки). Если выделенный объект не может являться носителем сценария, выводится сообщение **Current selection cannot have actions applied to it**.
- **Scene**  — содержит перечень всех объектов текущей сцены, снабженных сценариями. Здесь также приводятся атрибуты этих объектов. Для кадров указываются название слоя, номер и, если имеется, метка. Для клипов и кнопок указываются имя символа и, если задано, имя экземпляра.
- **Symbol Definition(s)**  — содержит древовидный перечень всех клипов, включающих элементы, в которых имеются сценарии.

Один щелчок на элементе в разделе **Script navigator** позволяет отобразить его сценарий в разделе **Script pane**.

Носители сценариев AS

Любой сценарий ActionScript, существующий в пределах FLA-документа и создаваемый с помощью панели **Actions**, обязательно прикрепляется к определенному объекту рабочей среды. Такой объект называется *носителем сценария*. Носителем сценария может быть только один из четырех типов объектов рабочей среды, к числу которых относятся:

- ключевой кадр (Keyframe) основной монтажной линейки (`_root`) или ключевой кадр монтажной линейки символа типа Movie Clip. Сценарий, прикрепленный к

ключевому кадру, в дальнейшем будем называть *сценарием кадра*. Кадр, содержащий сценарий, на монтажной линейке имеет специальную метку "a" ;

- экземпляр (Instance) символа типа Button (Кнопка). Сценарий, прикрепленный к экземпляру кнопки, в дальнейшем будем называть *сценарием кнопки*;
- экземпляр (Instance) символа типа Movie Clip (Клип). Сценарий, прикрепленный к экземпляру клипа, в дальнейшем будем называть *сценарием клипа*;
- экземпляр (Instance) компилированного клипа (Compiled Clip). Компилированные клипы используются, в частности, при работе с компонентами. Сценарий, прикрепленный к экземпляру компонента, в дальнейшем будем называть *сценарием компонента*.

Никакие другие графические и не имеющие визуального выражения элементы исходного документа не могут являться носителями сценариев. Нельзя, например, назначить сценарий группе, текстовому блоку или экземпляру звука.

Местоположение сценария в структуре документа является чрезвычайно важным для его корректного функционирования. Это связано с двумя основными особенностями:

- Существуют жесткие требования к оформлению сценария в зависимости от того, какой именно объект является его носителем. Так, сценарии клипов или кнопок *всегда* помещаются внутрь блока обработчика событий (event handler). К тому же клипы могут использовать обработчик, применение которого для кнопок недопустимо (в то время как обработчик событий кнопок может быть применен и к клипам). Сценарии кадра, напротив, не требуют использования обработчика событий. Частой ошибкой начинающих является прикрепление сценария, предназначенного одному носителю (например, клипу), к другому (например, к кадру). Такая ошибка всегда обнаруживается при проверке синтаксиса (команда **Check Syntax**) и на этапе компиляции, выполняемой в процессе тестирования проекта (команда главного меню **Control>Test Movie**). Адрес и суть ошибки выводятся в сообщении панели **Output**.
- Область действия или *видимости* (scope) сценария ограничена объектом, к которому он прикреплен. Так, например, сценарий, помещенный в кадр, действует только в пределах монтажной линейки, содержащей данный кадр, и не распространяется непосредственно на остальные монтажные линейки (клипов) документа. Для преодоления этих ограничений необходимо использовать *адресацию*, то есть явно указывать целевой объект, к которому относятся выполняемые действия. Вопросы области видимости достаточно многогранны и будут подробно рассмотрены на страницах этой книги в соответствующих разделах.

Для того чтобы назначить или "прикрепить" сценарий к конкретному объекту, необходимо выделить этот объект, явно передав ему фокус выделения. Так, например, если выделить кадр, содержащий экземпляр клипа, то выделенными окажутся и кадр, и все его содержимое на сцене (то есть клип); тем не менее фокус выделения находится именно на кадре, и именно кадр в этом случае будет являться носителем сценария, который будет введен в окно сценариев панели **Actions**.

Чрезвычайно важно перед вводом текста сценария убедиться, что требуемый носитель находится в фокусе и что сценарий будет назначен именно тому объекту, для

которого он предназначается. На объект, находящийся в фокусе в данный момент, в панели **Actions** указывают следующие признаки:

- в заголовке панели указывается текущий носитель **Actions — Frame** (Кадр), **Actions — Button** (Кнопка), **Actions — Movie Clip** (Клип). В остальных случаях заголовок панели состоит из одного слова — **Actions**;
- в категории **Current Selection** раздела **Script navigator** указывается текущий носитель. Если текущий выделенный объект не является носителем, здесь появляется сообщение **Current selection cannot have actions applied to it**;
- информация из категории **Current Selection** раздела **Script navigator** также отображается в закладке, расположенной слева под окном сценариев **Script pane**.

Для переключения между сценариями, прикрепленными к различным элементам документа, необходимо либо вручную перенести фокус выделения на соответствующий носитель, либо воспользоваться разделом **Script navigator** панели **Actions** или проводником фильма **Movie Explorer**. Зачастую возникает необходимость часто вносить изменения сразу в несколько сценариев. Для того чтобы каждый раз при необходимости работы со сценарием не приходилось "добираться" до него вручную, можно "закрепить" (pin) требуемый сценарий в панели **Actions**. При закреплении сценария в нижней части окна сценариев справа от закладки с текущим (активным) сценарием появляется дополнительная закладка с указанием местоположения закрепленного сценария и пиктограмма с изображением воткнутой кнопки . Для того чтобы отобразить в окне сценариев закрепленный сценарий, достаточно просто щелкнуть по его закладке, при этом закладка подсвечивается белым цветом. Если в окне сценариев отображается закрепленный сценарий (его закладка расположена справа от пиктограммы ), то переключение фокуса выделения никак не отражается на содержимом окна сценариев. В этом случае перейти к сценарию другого объекта можно либо открепив сценарий, либо щелкнув по закладке текущего сценария, либо воспользовавшись разделом **Script navigator**.

Для того чтобы закрепить сценарий, можно выполнить одно из следующих действий.

- Для закрепления текущего сценария можно щелкнуть на пиктограмме с изображением кнопки  в нижней части окна сценариев. При нажатии на нее кнопка "воткнется" и сценарий будет закреплен.
- Для закрепления текущего сценария можно щелкнуть на его закладке правой кнопкой мыши и из контекстного меню выбрать команду **Pin Script**.
- Для закрепления текущего сценария также можно выполнить команду **Pin Script** контекстного меню панели **Actions** или нажать сочетание кнопок <Ctrl>+<+>.
- Для закрепления любого сценария можно дважды щелкнуть по содержащему его элементу в разделе **Script navigator**.

Одновременно можно закрепить любое количество сценариев. Если закладки сценариев не умещаются в панели по ширине, справа от них появляется пиктограмма со стрелками, щелчок по которой выводит список всех не уместившихся закладок.

Для того чтобы открепить сценарий, можно выполнить одно из следующих действий.

- Для открепления текущего закрепленного сценария (его закладка отображается белым цветом) можно щелкнуть на пиктограмме с изображением воткнутой кнопки . При этом кнопка будет "вытащена", а сценарий откреплен, и его закладка удалена из панели **Actions**.
- Для открепления текущего закрепленного сценария можно щелкнуть на его закладке правой кнопкой мыши и из контекстного меню выбрать команду **Close Script**.
- Для открепления текущего сценария также можно выполнить команду **Close Script** контекстного меню панели **Actions** или нажать сочетание клавиш <Ctrl>+<_>.
- Для открепления всех закрепленных сценариев можно выполнить команду **Close All Scripts** контекстного меню панели **Actions** (<Ctrl>+<Shift>+<_>) или соответствующую команду контекстного меню закладки закрепленного или текущего сценария.

Синтаксические нормы и правила

Синтаксис языка программирования — это совокупность правил и соглашений, описывающих форму записи последовательности символов в программе. В данном разделе представлены некоторые сведения, описывающие наиболее общие синтаксические правила, принятые в ActionScript. Подробная информация о синтаксисе различных элементов языка содержится далее, в главах, описывающих данные элементы.

Предложения

Текст программы, как и текст естественного языка (например, русского), состоит из набора строк, включающих произвольную последовательность символов. Строки программы, состоящие из символов языка, называются *предложениями* или утверждениями (statements). Таким образом, синтаксические правила описывают, как должны выглядеть предложения языка, для того чтобы они могли быть корректно интерпретированы.

Для разделения предложений в ActionScript используется точка с запятой (`;`). В соответствии с этим точка с запятой ставится в конце каждого предложения. Каждое предложение начинается с новой строки. Вот пример кода, состоящего из нескольких предложений:

```
ball ++;
holder.attachMovie("drop", "drop" + ball, ball);
holder["drop" + ball]._x = 140;
holder["drop" + ball]._y = 112;
ballsInterval = setInterval(setBalls, 1000);
```

В действительности точка с запятой в конце предложения может быть опущена. Если следующее предложение начинается с новой строки, это не вызовет ошибку, и сценарий будет откомпилирован корректно. Перевод строки автоматически указывает интерпретатору на начало нового предложения. Тем не менее использование точки с запятой в конце предложения является желательным, поскольку позволяет сделать текст программы более читаемым и, таким образом, избежать возможных ошибок.

При условии использования точки с запятой в конце предложения несколько коротких предложений можно разместить на одной строке для экономии места и смысловой группировки. Так, например, следующий фрагмент сценария не содержит ошибки:

```
holder["drop" + ball]._x = 140; holder["drop" + ball]._y = 112;
```

Однако записывать несколько предложений в одну строку не рекомендуется, поскольку это затрудняет чтение и снижает читабельность кода.

Комментарии

Комментарии — это текст внутри программы, который игнорируется интерпретатором и служит для пояснения назначения фрагментов программного кода. Комментарии дают разработчику возможность создания пояснительных записей непосредственно в тексте программы. Это позволяет сделать программный код более "говорящим" и для самого автора, и для остальных членов команды разработчиков. Применение комментариев является чрезвычайно полезной практикой. При работе с длинными и довольно сложными программами необходимо контролировать большое количество переменных, объектов и хорошо ориентироваться в коде. При наличии комментариев делать это значительно проще. Кроме того, многочисленные важные подробности реализации программы со временем имеют тенденцию забываться. Если спустя некоторое время после создания сценария требуется внести в него какие-то изменения, отсутствие детальных и грамотных комментариев может привести к тому, что автору придется заново мучительно разбираться со своим "творением".

Существует два типа комментариев. Комментарии могут быть *однострочными* или *многострочными*.

Для создания однострочных комментариев используются две наклонные черты (//). Все символы, помещенные между символами (//) и концом строки, являются комментарием. При переходе на следующую строку такой комментарий заканчивается. Ниже приводится несколько примеров однострочных комментариев:

```
//Это однострочный комментарий

//Это многострочный комментарий, который
//создан с помощью нескольких однострочных
```

Для создания многострочного комментария необходимо в начале такого комментария разместить символ /*, а в конце символ */. Многострочный комментарий может занимать неограниченное количество строк. Все символы и предложения, рас-

положенные между /* и */, являются комментариями. Многострочный комментарий может включать однострочные комментарии, однако не может содержать другой многострочный комментарий.

```
/* Это
образец
многострочного
комментария*/
```

При использовании режима **Syntax coloring** комментарии подсвечиваются установленным цветом (по умолчанию — серым).

Имена

Имя (name) или *идентификатор* (identifier) — это совокупность символов, которая используется для идентификации некоторой сущности в сценарии. Имена служат для идентификации переменных (variables), массивов (arrays), функций (functions), методов (methods), свойств (properties), классов (classes) и объектов (objects). Имя дает возможность обратиться к сущности, извлечь или изменить хранимое в ней значение, вызвать определенную функцию и т. д.

Правила именования

Важно строго соблюдать правила именования. Все идентификаторы должны удовлетворять следующим требованиям.

- Состоять исключительно из буквенных, числовых символов и символов (_), ($). Это, в частности, означает, что в именах недопустимы пробельные символы.
- Начинаться с буквы, символа (_) или ($).
- Быть уникальными в пределах своей области видимости. Это, например, означает, что в один и тот же кадр нельзя поместить два экземпляра клипов с одинаковыми именами. Такая ситуация приведет к некорректной работе сценариев.
- Не совпадать с ключевыми, зарезервированными словами и встроенными элементами ActionScript. Несоблюдение этого правила может привести к ошибкам или неожиданным результатам. Застраховаться от таких ошибок достаточно просто, поскольку все ключевые и зарезервированные слова, как и встроенные элементы, подсвечиваются в окне сценариев в случае использования режима **Syntax coloring**.

При именовании встроенных идентификаторов в ActionScript используется подход, в соответствии с которым, в составных именах (именах, состоящих из нескольких слов) начальная буква каждого слова, начиная со второго, является прописной. Вот несколько примеров:

```
gotoAndPlay("Logo", "Start");
createEmptyMovieClip("mc", 1);
removeMovieClip(this);
```

Выделение с помощью заглавной буквы делает имена более читаемыми и позволяет лучше и быстрее ориентироваться в программном коде. Однако первая буква, как правило, всегда является строчной. С прописной (заглавной) буквы обычно начинаются только имена классов. При создании пользовательских идентификаторов можно использовать другую систему именования или разработать свою, однако на страницах этой книги авторы будут придерживаться описанного подхода.

Зависимость от регистра

ActionScript 2.0 является *регистрозависимым* языком. Это означает, что строчные и прописные (заглавные) буквы различаются. Это важное обстоятельство необходимо иметь в виду, поскольку теоретически в программе могут быть использованы несколько внешне очень похожих идентификаторов, отличающихся только регистром.

```
initNum
initNUM
initnum
```

Несмотря на внешнее сходство эти идентификаторы, тем не менее, являются различными и обозначают разные объекты.

Примечание

Во избежание возможных ошибок в работе программы необходимо придерживаться единого стиля именования идентификаторов. Так, для отделения слов в составных именах можно либо выделять каждое последующее слово прописной буквой, либо использовать символ нижнего подчеркивания. Однако желательно последовательно применять принятую систему именования.

В ActionScript 1.0, используемом в предыдущих версиях Macromedia Flash, поддерживалась лишь частичная зависимость от регистра. Так, например, переменные с именами `initNum` и `InitNum` обозначали одно и то же значение. Зависимость от регистра имеет место при публикации документа для проигрывателя Flash Player 7 (и выше) независимо от того, применяется ли ActionScript 1.0 или ActionScript 2.0. Для того чтобы исключить зависимость от регистра, можно публиковать документ для Flash Player 6 или ниже. Однако при этом может быть утрачена часть функциональности, связанная с использованием новых возможностей, которые не поддерживаются более ранними версиями проигрывателя.

Функции

Функция (function) или *процедура* (procedure) — это фрагмент программы, состоящий из объявлений данных и предложений и выполняющий определенные вычисления и действия. Он может быть многократно вызван из других частей программы или других элементов документа. При необходимости многократного выполнения однотипных вычислений или действий их удобно оформить в виде функции. В этом случае для выполнения требуемых вычислений или действий не требуется всякий раз повторять все реализующие их предложения, достаточно просто вызвать соответствующую функцию. Это позволяет существенно повысить эффективность программы за счет многократного использования одного и того же программного кода.

При вызове функции ей могут быть переданы определенные значения, называемые *параметрами* (parameters) или *аргументами* (arguments). Параметры служат для реализации различных вариантов выполнения функции. Так, например, функция, служащая для вычисления площади прямоугольника, в качестве параметров может принимать длины его сторон. В зависимости от значений длин сторон результат, возвращаемый функцией, будет разным, однако сама процедура вычисления площади, помещенная в функцию, остается неизменной.

Для вызова функции в ActionScript используется следующий синтаксис:

```
functionName (arg1, arg2, ..., argN);
```

Здесь:

`functionName` — имя функции;

`arg1, arg2, ..., argN` — аргументы функции.

Важно запомнить, что после имени функции всегда следуют открывающая и закрывающая круглые скобки. Круглые скобки указывают интерпретатору на то, что данный элемент является функцией. Если опустить скобки, `functionName` будет интерпретирован как идентификатор, то есть как ссылка на переменную, свойство или объект. В этом случае вместо вызова функции будет выполнено обращение к соответствующему объекту, что естественно даст совершенно иной результат.

Внутри круглых скобок через запятую указываются параметры, передаваемые функции для выполнения вычислений. Число параметров, принимаемых функцией, в общем случае не ограничено. Если функция не принимает параметров, то круглые скобки остаются пустыми, тем не менее их наличие необходимо.

ActionScript включает большой набор встроенных или *предопределенных* функций, которые выполняют разнообразные действия, широко используемые для реализации базовых задач программирования. Кроме того, разработчики могут создавать собственные *пользовательские* функции для реализации специфических вычислений и действий, при этом зачастую используя в них предопределенные функции. Ниже представлены предложения, содержащие вызовы предопределенных функций:

```
play() //Нет параметров
getURL ("mailto:book_flash@mail.ru"); //Один параметр
startDrag(this, true, 0, 0, 550, 400); //Несколько параметров
```

Функция *trace()*

Функция `trace()` — это скромный с виду, но чрезвычайно важный и полезный инструмент в работе со сценариями. Описание функции `trace()` вынесено в эту главу в связи с тем, что эта функция представляет собой базовое средство отладки программы и будет в дальнейшем повсеместно использоваться в примерах, приводимых на страницах этой книги. Функция `trace()` применяется исключительно для работы в среде тестирования. Она позволяет в среде тестирования отобразить на экране панель **Output** и вывести в нее любое сообщение, переданное функции в качестве аргумента. Применение функции `trace()` обеспечивает возможность передачи данных для отображения в панели **Output** из любого фрагмента и точки кода. Это по-

зволяет в нужный момент оценить значения переменных, состояние операторов или последовательность выполнения блоков программы.

Функция `trace()` имеет следующий синтаксис:

`trace(`*`expression`*`)`

В качестве аргумента функция `trace()` может принимать выражение *`expression`*. Во время выполнения программы это выражение вычисляется, и строковое представление результата вычисления отображается в панели **Output**. Приведем несколько примеров:

```
trace ("Привет !") // Выводит Привет !
var x = 5; // Создаем переменную x со значением 5
var y = 2; // Создаем переменную y со значением 2
trace (x * y); // Выводит результат вычисления 10
trace (x > y); // Выводит результат вычисления true
trace (x + "*" + y + "=" + x * y); // Выводит строку 5 * 2 = 10
var date = new Date(); // Создаем экземпляр класса Date
trace (date); // Выводит текущую дату и время
```

Если в первый ключевой кадр нового документа поместить этот сценарий, то в результате выполнения команды **Control>Test Movie** в среде тестирования будет выведена панель **Output** с соответствующей информацией. Каждое новое сообщение, передаваемое вызовом функции `trace()`, выводится в панели **Output** с новой строки.

Применение функции `trace()` может существенно снизить быстродействие проигрывателя в среде тестирования. Если требуется отключить действие этой функции вместо того, чтобы вручную удалять ее из программного кода (это может отнять много времени, кроме того, впоследствии ее применение может снова оказаться необходимым), можно установить флажок **Omit trace actions** в диалоговом окне **Publish Settings** (**File>Publish Settings...**). В результате этого действия все вызовы функции `trace()` при компиляции не будут включены в программный код конечного SWF-документа (при этом они по-прежнему будут присутствовать в сценарии исходного FLA-файла).

Инициирование и последовательность выполнения сценариев. Системные и пользовательские события

Как уже было отмечено, специфической чертой сценариев ActionScript является возможность их рассредоточения по различным структурным элементам документа. При этом следует иметь в виду, что все монтажные линейки документа воспроизводятся синхронно и, соответственно, происходит непрерывная смена кадров. Возникает вопрос, каким образом и в какой последовательности выполняются сценарии, прикрепленные к различным кадрам, клипам и кнопкам в пределах одного и того же документа? Сначала рассмотрим, каким образом происходит выполнение сценария кадра.

Последовательность выполнения сценариев кадров

Сценарий кадра выполняется сразу при входе в него воспроизводящей головки. После выполнения сценария кадра происходит обновление изображения на экране и воспроизведение звука, размещенного в этом кадре. Таким образом, последовательность выполнения сценариев, прикрепленных к кадрам, расположенным в пределах одной и той же монтажной линейки, определяется взаимным расположением этих кадров. Так, вначале выполняется сценарий первого кадра, затем воспроизводящая головка входит во второй кадр, в этот момент выполняется сценарий кадра номер два, после чего происходит переход к третьему кадру и т. д. Такое выполнение сценариев называется *синхронным*. При синхронном выполнении сценариев моментом их выполнения можно управлять путем выбора требуемой временной позиции ключевого кадра. Так, если сценарий должен быть выполнен в начале третьей секунды анимации, его можно поместить в 25-й кадр монтажной линейки основного фильма (при скорости воспроизведения 12 fps).

Существенное внимание следует уделить вопросу последовательности выполнения сценариев, прикрепленных к кадрам различных монтажных линеек, а также к кадрам, расположенным в разных слоях.

Если в одном и том же кадре фильма имеется несколько сценариев, расположенных в разных слоях, то их выполнение всегда происходит сверху вниз. Так, пусть на монтажной линейке имеется три слоя `actions1`, `actions2` и `actions3`, расположенных последовательно друг над другом так, что слой `actions1` находится в самом низу. Если в первом кадре каждого из этих слоев находится свой собственный сценарий, то порядок выполнения этих трех сценариев будет следующим: сначала выполнится сценарий первого кадра слоя `actions3`, затем сценарий первого кадра слоя `actions2`, затем сценарий первого кадра `actions1`.

Более сложная ситуация возникает, когда кадр монтажной линейки, к которому прикреплен сценарий, содержит экземпляр клипа, в свою очередь также содержащий сценарий на своей линейке. В этом случае порядок выполнения сценариев определяется в соответствии со следующим правилом.

В кадре, где клип появляется впервые, сначала выполняется сценарий кадра основной монтажной линейки, затем сценарий кадра монтажной линейки этого клипа. В последующих кадрах (на протяжении времени существования клипа) вначале выполняется сценарий кадра монтажной линейки клипа, а затем сценарий кадра основной монтажной линейки, содержащего данный клип.

Еще более запутанная ситуация имеет место в случае, когда в пределах одного слоя размещается несколько различных клипов, содержащих сценарии на своих монтажных линейках. В этом случае порядок выполнения сценариев определяется взаимным расположением клипов в стеке (стопке), а также параметром **Load order** (Порядок загрузки), который задается в диалоговом окне **Publish Settings** (**File>Publish Settings**). Взаимным размещением объектов в стеке слоя в рабочей среде можно управлять с помощью команд главного меню **Modify>Arrange**. Для определе-

ния порядка выполнения сценариев клипов, расположенных в одном кадре в пределах одного слоя необходимо запомнить следующее правило.

Для кадра, в котором клипы появляются впервые, установка параметра ***Load order*** *в значение* ***Bottom up*** *(Снизу вверх) означает, что в первую очередь будет выполнен сценарий самого нижнего клипа на слое, затем клипа, расположенного над ним, и т. д. Установка значения* ***Top down*** *(Сверху вниз) означает, что сценарии монтажных линеек клипов выполняются по нисходящей, начиная с самого верхнего.*

Для остальных кадров, в которых присутствуют клипы, используется обратный порядок выполнения сценариев. Установка значения ***Bottom up*** *приводит к тому, что сценарии клипов в этих кадрах выполняются сверху вниз, начиная с самого верхнего (а не с нижнего). Установка значения* ***Top down*** *приводит к тому, что сценарии монтажных линеек клипов в этих кадрах выполняются снизу вверх, начиная с самого нижнего.*

Так, пусть в одном кадре монтажной линейки находятся три клипа с именами `mc1`, `mc2`, `mc3`, причем `mc1` находится в самом низу, над ним расположен `mc2`, а `mc3` помещен на передний план. Пусть в первом кадре монтажной линейки каждого из этих клипов находится сценарий. Если параметр **Load order** установлен в значение **Bottom up**, то порядок выполнения этих трех сценариев будет следующим. В первом кадре основной монтажной линейки сначала будет выполнен сценарий монтажной линейки клипа `mc1`, затем сценарий монтажной линейки клипа `mc2` и, наконец, сценарий монтажной линейки клипа `mc3`. В последующих кадрах вплоть до удаления этих клипов сначала будет выполняться сценарий монтажной линейки клипа `mc3`, затем сценарий монтажной линейки клипа `mc2` и, наконец, сценарий монтажной линейки клипа `mc1`. Если же параметр **Load order** установлен в значение **Top down**, то будет использоваться обратный порядок.

Если клипы, содержащие сценарии, находятся в одном кадре фильма, но в разных слоях, порядок выполнения этих сценариев также определяется параметром **Load order**, при этом имеют место те же самые закономерности.

Примечание

Во избежание возможных проблем, связанных с порядком выполнения сценариев кадров, в пределах одной монтажной линейки рекомендуется размещать все сценарии на одном слое. Этот слой обычно помещается на самый верх и ему дается название `actions`.

Сценарии кнопок и клипов

В отличие от сценариев, прикрепленных к кадрам, инициирование выполнения сценариев, прикрепленных непосредственно к экземплярам кнопок или клипов, не связано напрямую с положением воспроизводящей головки. Любой сценарий кнопки или клипа выполняется только при наступлении определенного события. *Событие* (event) является реакцией либо на действия пользователя, либо на процесс, связанный с ходом загрузки и воспроизведения SWF-документа. Наступление события называется *генерированием* события. Генерирование события может инициировать выполнение программного кода. Так, например, при нажатии кнопки генерируется событие `press` (нажатие). Результатом обнаружения данного события может быть выполнение сценария, прикрепленного к соответствующей кнопке.

Все события можно подразделить на две группы: *системные* (system) и *пользовательские* (user) события.

- Системные события генерируются проигрывателем Flash Player в результате протекания процессов, связанных с обработкой данных и воспроизведением SWF-документа. Примерами системных событий могут служить, например, окончание загрузки информации из внешнего файла или истечение времени экспонирования кадра (при данной скорости воспроизведения).
- Пользовательские события генерируются в результате определенных действий пользователя. Примерами таких событий являются, например, перемещение курсора мыши, нажатие клавиатурной клавиши или наведение на кнопку.

Встроенные классы ActionScript содержат определение ряда обработчиков событий, которые могут быть использованы объектами для инициирования выполнения сценариев. Различные объекты могут отслеживать и контролировать наступление событий, определенных в породившем их классе. Кроме того, существует возможность уведомлять объекты, не умеющие отслеживать наступление определенных событий, об их обнаружении другими объектами.

В ходе воспроизведения фильма может генерироваться множество событий, однако далеко не каждое из них приводит к выполнению каких-то действий. Например, если пользователь нажимает кнопку, при этом объектом `Button` генерируется событие `onPress`, однако в отсутствие сценария, связанного с данной кнопкой, нажатие не вызовет никакой реакции. Для того чтобы наступление события инициировало выполнение определенного сценария, необходимо использовать так называемый *обработчик событий* (event handler). Обработчик событий — это программная инструкция, синтаксически похожая на функцию. Обработчик событий определяет, наступление какого события повлечет за собой выполнение сценария. Так, например, для обработки событий клипов можно использовать обработчик `onClipEvent`. Он имеет следующий синтаксис:

```
onClipEvent (событие) {
блок кода
...
}
```

Здесь `событие` — событие, наступление которого инициирует выполнение блока кода, расположенного в фигурных скобках. Важно запомнить, что любой сценарий, прикрепленный к экземпляру кнопки или клипа, должен быть помещен в блок обработчика. Несоблюдение этого правила приведет к ошибке при компиляции.

Обработчики событий могут использоваться различным образом. В частности, сценарий клипа или кнопки не обязательно прикреплять к самому экземпляру. Его можно разместить в кадре. При этом обработчик событий определяется как метод объекта. Подробно вопросы использования обработчиков событий будут рассмотрены в *главах 10 и 11*.

Представление об объектно-ориентированном программировании

Объектно-ориентированное программирование (ООП) — это подход к структурированию и организации программы, при котором для описания объектов реального мира или для описания абстрактных объектов создаются программные модули, содержащие все данные и команды, необходимые для представления одной категории или *класса* объектов. Объектно-ориентированный подход заключается в идентификации объекта (реального или абстрактного), выявлении характерных особенностей его представления и функционирования и последующего моделирования этого объекта с учетом его внутренних процессов и возможности реализации связей с другими объектами. Такой подход может быть реализован не только для моделирования предметов реального мира, но и для создания многократно используемых программных компонентов.

Программный модуль, реализующий абстрактный тип данных и содержащий программное представление моделируемого объекта, называется *классом* (class). Класс — это строительный блок, лежащий в основе ООП. Так, например, для хранения данных о клиентах можно создать класс `Client`, содержащий данные о характеристиках клиентов (профессия, стаж, возраст, семейное положение и т. д.) и набор действий, позволяющих обрабатывать эти данные (например, выводить информацию о клиенте, добавлять информацию о новых клиентах, удалять записи и т. д.).

ActionScript содержит ряд встроенных классов, использование которых лежит в основе любого сценария. К числу таких классов относятся, например, `Number`, `String`, `Button`, `MovieClip`, `TextField`, `Math`, `Date`, `Color` и др. Эти классы позволяют реализовать широкую функциональность, при этом пользователю не требуется знать внутренние подробности функционирования класса, поскольку взаимодействие с ним осуществляется с помощью внешнего интерфейса.

Класс содержит объявления *свойств* и *методов*. Свойства — это контейнеры для хранения данных (например, координат объекта или его цвета). Методы — это функции, определяющие операции, производимые с объектами, порождаемыми классом. Так, например, встроенный класс `MovieClip` определяет свойства `_x`, `_y`, позволяющие контролировать координаты клипа, и методы `stop()` и `play()`, дающие возможность останавливать и запускать анимацию клипа.

С выходом седьмой версии Flash и появлением новой редакции языка ActionScript 2.0 создание пользовательских классов теперь осуществляется только во внешних файлах, имеющих расширение as. Предопределенные классы Flash реализованы разработчиками Macromedia на уровне проигрывателя Flash Player. Эти классы предоставляют базовый программный инструментарий для создания сценариев ActionScript. Создание пользовательских классов выполняется с целью решения конкретных практических задач, встающих перед разработчиком. Для этого необходимо использовать автономный редактор ActionScript или любой текстовый редактор. При публикации исходного FLA-документа внешние файлы классов автоматически включаются в конечный SWF-фильм.

Класс представляет собой шаблон, описывающий поведение и представление объекта. Для реализации заложенной в нем функциональности необходимо создать *экзем-*

пляр (instance) этого класса, то есть конкретный объект. В дальнейшем термины *объект* и *экземпляр класса* будут использоваться как синонимы. Важно понимать, что класс создается в отдельном AS-файле (в случае применения AS 2.0), а его экземпляры могут быть использованы в исходном FLA-документе. Экземпляр класса получает доступ к свойствам и методам, определенным в классе. При этом внутренние подробности реализации *инкапсулируются*, то есть остаются скрытыми. Вызывая метод `play()`, мы запускаем анимацию клипа, однако механизм этого действия скрыт. Инкапсуляция позволяет абстрагироваться от деталей реализации и обеспечивает точную и логичную организацию программы. Это позволяет существенно оптимизировать программу, повысить ее эффективность, сделать более устойчивой и переносимой.

Один класс может породить любое количество объектов (экземпляров). При этом все они обладают одинаковым поведением, и функционируют единообразно согласно механизму, реализованному в классе. Тем не менее фактические проявления разных объектов могут отличаться, поскольку каждый объект может иметь разные значения свойств и вызывать различные методы. Например, два клипа, порожденные классом `MovieClip`, могут иметь разные значения координат, разные размеры. Анимация одного из них может быть остановлена, а у другого продолжать воспроизводиться. Это напоминает принцип использования символов в рабочей среде документа. Сам символ, содержащий графическое и иное представление объекта, находится в библиотеке и служит шаблоном для создания экземпляров. Экземпляры внешне и функционально повторяют все детали родительского символа, однако могут быть видоизменены путем задания различных параметров.

Создание экземпляров класса, как правило, выполняется с помощью оператора `new` и *функции конструктора* (constructor function). При создании объекта он обычно сохраняется в переменной, с помощью которой в дальнейшем можно обращаться к свойствам и вызывать методы класса. Переменная, содержащая объект, имеет абстрактный тип, совпадающий с именем класса. Этот тип может быть явно указан при создании переменной.

```
var obj:Object = new Object(); /* Создаем экземпляр класса Object
и сохраняем его в переменной obj. Переменная obj имеет тип Object */

var myArray:Array = new Array(); /* Создаем массив и сохраняем его
в переменной myArray. Переменная имеет тип Array */

var date:Date = new Date(); /* Создаем экземпляр класса Date и помещаем
его в переменную date. Переменная имеет тип Date */
```

Существует ряд встроенных классов, для которых создание экземпляра осуществляется без использования оператора `new` и функции конструктора. К числу таких классов относятся: `Button`, `MovieClip` и `TextField`. Создание экземпляров этих классов осуществляется непосредственно в рабочей среде исходного документа. Размещение на сцене экземпляра кнопки, клипа или динамического или пользовательского текстового блока приводит к автоматическому созданию экземпляров классов.

После создания экземпляра класса можно осуществлять обращение к свойствам объекта и вызывать его методы.

```
trace(myArray.length); /*Выводит в окно Output количество элементов
массива*/
```

```
myArray.push(25); /* Добавляет в конец массива новый элемент со значением
25 */
date.getFullYear(); // Возвращает текущий год
```

Создание сценария во внешнем файле

Внешние файлы сценариев ActionScript сохраняются в формате AS с расширением as и применяются для создания пользовательских классов или для хранения программных кодов (например, пользовательских функций), автоматически включаемых в проект при публикации с помощью директивы `#include`. Для создания файла сценария можно выполнить одно из следующих действий.

- Выполнить команду главного меню рабочей среды **File>New...** и в разделе **Type** диалогового окна **New Document** выбрать **ActionScript File**. В результате будет создан новый файл с расширением as, который будет открыт в окне автономного редактора ActionScript. Автономный редактор сценариев доступен только во Flash MX 2004 Professional.
- Экспортировать сценарий из панели **Actions** во внешний AS-файл с помощью команды **Export Script...** контекстного меню этой панели.
- Воспользоваться любым текстовым редактором и сохранить файл с расширением as.

Во время публикации сценарии из внешних файлов включаются в результирующий SWF-документ. Таким образом, при распространении готового проекта файлы сценариев не требуются. При этом важно иметь в виду, что любые изменения сценариев во внешних файлах требуют повторной публикации, для того чтобы сделанные изменения могли быть внесены в конечный фильм.

Применение директивы *#include*

В активе разработчика, как правило, имеется набор программных модулей, используемых в различных проектах для решения типовых задач. Для того чтобы многократно не вставлять типовые фрагменты кода в исходный документ вручную, можно разместить их во внешних файлах (с расширением as) и при необходимости включать их с помощью директивы `#include`. Это позволит сделать программный код в исходном документе менее громоздким. Директива `#include` имеет следующий синтаксис:

```
#include "путь"
```

Здесь *путь* — это абсолютный или относительный путь к внешнему файлу со сценарием. Путь представляет собой строковый литерал и всегда заключается в кавычки. Следует отметить, что перед названием директивы указывается обязательный префикс #, а в конце предложения, содержащего данную директиву, отсутствует точка с запятой.

Директива `#include` загружает сценарий из внешнего файла, помещая его непосредственно в тот участок кода, где она находится. Загрузка внешнего сценария в документ выполняется при публикации, то есть на стадии компиляции, в результате чего программный код из внешнего файла включается в исходный документ, как

если бы он изначально был в нем создан. После публикации для функционирования конечного документа внешний сценарий не требуется, поскольку его код был полностью включен в исходный документ. Однако при необходимости внесения изменений во внешний сценарий следует заново выполнить публикацию исходного документа, чтобы эти изменения были включены в конечный документ.

При выполнении автоматической проверки синтаксиса (в частности, во время тестирования) вместе с кодом исходного документа анализируется также код всех внешних файлов сценариев, включаемых в документ. При обнаружении синтаксических ошибок во внешних файлах выводится соответствующее сообщение.

При указании пути к внешнему файлу со сценарием используется стандартный синтаксис адресации файлов в файловой системе. Для обращения к вышележащему каталогу используется две точки (`..`), для обращения к нижележащему каталогу используется наклонная черта (`/`). Следует иметь в виду, что относительные ссылки разрешаются относительно местоположения исходного файла FLA, в котором используется директива `#include`. Вместо относительной адресации можно использовать абсолютную адресацию. В этом случае путь начинается с указания диска. Приведем несколько примеров:

```
#include "code.as" // файл code.as находится в том же каталоге,
что и исходный документ

#include "..actions/code.as" // файл code.as находится в каталоге
actions, который находится на одном уровне с каталогом, содержащем
исходный документ

#include "C:/actions/code.as" // Абсолютный путь в Windows
```

Проекты. Панель *Project*

Зачастую сложные Flash-проекты могут включать в себя большой набор различных файлов. Чаще всего приходится иметь дело с форматами FLA, SWF, AS, HTML, XML, TXT, JPEG, MP3 и FLV. При работе с такими комплексными проектами для внесения изменений приходится часто переключаться между разными документами, входящими в состав проекта. С помощью главного меню рабочей среды нельзя открыть многие из перечисленных выше форматов. Это создает определенные неудобства при необходимости внесения изменений и при тестировании документа.

Macromedia Flash MX 2004 Professional включает в себя новый и чрезвычайно удобный инструмент организации файлов проекта — панель **Project**. С помощью этой панели можно группировать файлы проекта, быстро переключаться между ними, легко осуществлять тестирование проекта, а также избегать возможности случайного редактирования файла или замены его новой версии более ранней редакцией.

Работа с проектами

Для того чтобы открыть панель **Project**, необходимо выполнить команду главного меню **Window>Project** или нажать сочетание клавиш <Shift>+<F8>. Внешний вид этой панели представлен на рис. 3.7.

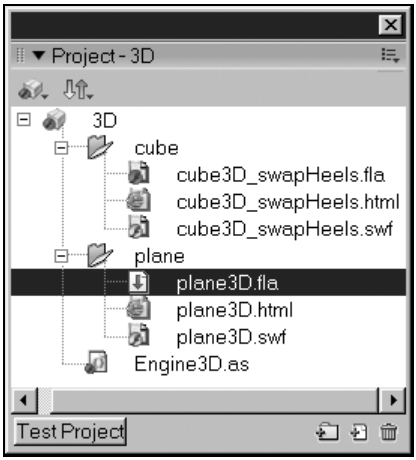


Рис. 3.7. Панель **Project**

В панели **Project** отображается иерархическая структура файлов, входящих в состав проекта. С помощью этой панели в проект могут быть добавлены как файлы рабочей среды (FLA, SWF, AS), так и любые другие внешние файлы (например, HTML, XML, TXT, JPEG), а также другие проекты. Возможности панели **Project** позволяют удалять файлы из проекта (при этом они физически не удаляются с диска или сервера), упорядочивать их внутри панели в группы с помощью папок, а также блокировать их для редактирования. *Проект* (project) представляет собой XML-документ с расширением flp, содержащий перечень файлов, входящих в состав проекта. Все изменения, вносимые в панели **Project**, автоматически отражаются в FLP-документе проекта, поэтому специально сохранять файл проекта не требуется. При необходимости файл проекта может быть создан вручную без помощи панели **Project**, при этом необходимо помнить, что FLP-файлы используют кодировку UTF-8. Листинг 3.1 содержит образец XML-документа проекта, представленного на рис. 3.7.

Листинг 3.1. Образец файла проекта

```
<flash_project name="3D" version="1">
  <project_folder name="plane" expanded="true">
    <project_file path="./plane3D.fla" filetype="fla" profile="" />
    <project_file path="./plane3D.swf" filetype="swf" />
    <project_file path="./plane3D.html" filetype="html" />
  </project_folder>
  <project_folder name="cube" expanded="true">
    <project_file path="./cube3D_swapHeels.html" filetype="html" />
    <project_file path="./cube3D_swapHeels.fla" filetype="fla"↳
profile="" />
```

```
    <project_file path="./cube3D_swapHeels.swf" filetype="swf" />
  </project_folder>
  <project_file path="../Classes/my3D/Engine3D.as" filetype="as" />
</flash_project>
```

Создание нового проекта

Для создания нового проекта можно выполнить одно из следующих действий.

- Выполнить команду главного меню **File>New...** и в разделе **Type** диалогового окна **New Document** выбрать **Flash Project**.
- Если другой проект еще не открыт в рабочей среде Flash, открыть панель **Project** и щелкнуть по надписи **new project** в самой панели.
- Если другой проект еще не открыт в рабочей среде Flash, щелкнуть правой кнопкой мыши в пределах сцены или в окне файла ActionScript и из контекстного меню выбрать команду **Add to New Project**. При этом текущий документ будет автоматически помещен в новый проект.
- Если в рабочей среде открыт другой проект, щелкнуть на пиктограмме  в панели **Project** и из всплывающего меню выбрать команду **New Project...**. При этом будет выведено сообщение о том, что текущий проект уже открыт. Нажатие кнопки **OK** приведет к созданию нового проекта, при этом текущий проект будет закрыт.

После выполнения команды создания нового проекта в диалоговом окне **New Project** необходимо задать имя проекта и указать его местоположение на диске.

Открытие существующего проекта

В рабочей среде Flash одновременно может быть открыт только один проект. Если один проект уже открыт, то открытие другого приведет к закрытию первого проекта. Для того чтобы открыть существующий проект, можно выполнить одно из следующих действий.

- Выполнить команду главного меню **File>Open...** и в разделе **Type** диалогового окна **New Document** выбрать **Flash Project**.
- Если другой проект еще не открыт в рабочей среде Flash, открыть панель **Project** и щелкнуть по надписи **open** в самой панели.
- В панели **Project** щелкнуть на пиктограмме  и из всплывающего меню выбрать команду **Open Project...**.

Для того чтобы переименовать проект, необходимо выделить его пиктограмму в панели **Project** и, щелкнув на пиктограмме , из всплывающего меню выбрать команду **Rename...**, либо выполнить одноименную команду контекстного меню самого проекта, щелкнув на его пиктограмме правой кнопкой мыши.

Добавление файлов в проект

В проект могут быть добавлены файлы любого типа (не только форматы рабочей среды Flash). В проект также может быть добавлен другой проект. Добавление файлов в проект не приводит к изменениям в файловой структуре, то есть исходное местоположение файлов на диске не изменяется. Файл может быть добавлен в проект только один раз. Повторное добавление файла, уже включенного в проект, приводит к появлению сообщения об ошибке. Для того чтобы добавить файл в проект, можно выполнить одно из следующих действий.

- В панели **Project** щелкнуть на пиктограмме  и из всплывающего меню выбрать команду **Add File...**.
- Правой кнопкой мыши щелкнуть на любом элементе в панели **Project** и из контекстного меню выбрать команду **Add File...**.
- В окне открытого FLA- или AS-документа щелкнуть правой кнопкой мыши, в контекстном меню выбрать команду **Add to Project**. В результате текущий документ будет добавлен в открытый проект.
- Щелкнуть на пиктограмме  слева в нижней части панели **Project**.

Организация файлов проекта с помощью папок

Для того чтобы упорядочить файлы внутри панели **Project**, можно использовать папки, примерно так же, как это происходит в библиотеке (Library) для организации расположенных в ней элементов. Группировка файлов с помощью папок в панели **Project** не приводит к их перемещению в файловой системе. Папки должны иметь уникальные имена в пределах одной и той же структурной ветви. Для того чтобы создать папку, можно выполнить одно из следующих действий.

- В панели **Project** щелкнуть на пиктограмме **Add Folder...**.
- Правой кнопкой мыши щелкнуть на любом элементе в панели **Project** и из контекстного меню выбрать команду **Add Folder...**.
- Щелкнуть на пиктограмме  слева в нижней части панели **Project**.

Для того чтобы поместить файл в папку, нужно взяться за него и просто перетащить, бросив на пиктограмму папки. Аналогичным образом можно перемещать целые папки с вложенными файлами.

Для того чтобы переименовать папку, необходимо выделить ее и, щелкнув на пиктограмме , из всплывающего меню выбрать команду **Rename...** либо выполнить одноименную команду контекстного меню самой папки.

Открытие файлов проекта

С помощью панели **Project** можно быстро открыть любой входящий в состав проекта документ. Для того чтобы это сделать, достаточно дважды щелкнуть на требуемом файле в панели **Project**. Если формат открываемого файла поддерживается Flash, он будет открыт в рабочей среде. Если формат не поддерживается, то он будет открыт в приложении, используемом для обработки файлов данного формата.

Удаление из проекта файлов и папок

Для того чтобы удалить из проекта папку или файл, необходимо выделить требуемый элемент в панели **Projects** и выполнить одно из следующих действий.

- Щелкнуть на пиктограмме  и из всплывающего меню выбрать команду **Remove**.
- Щелкнуть на элементе правой кнопкой мыши и из контекстного меню выбрать команду **Remove**.
- Нажать клавишу <Delete>.
- Щелкнуть на пиктограмме  слева в нижней части панели **Project**.

Удаление файлов из проекта не приводит к их удалению из файловой структуры.

Закрытие проекта

Для того чтобы закрыть проект, нужно щелкнуть на пиктограмме  и из всплывающего меню выбрать команду **Close Project** либо правой кнопкой мыши щелкнуть на пиктограмме самого проекта и выбрать эту же команду из контекстного меню.

Тестирование и публикация файлов проекта

С помощью панели **Project** можно выполнять тестирование и публикацию как отдельных FLA-файлов, входящих в состав проекта, так и всего проекта целиком.

Публикация проекта

Публикация файлов проекта осуществляется на основе профиля публикации (publish profile), используемого каждым FLA-документом проекта. Для того чтобы указать профиль публикации для документа, нужно выделить его в панели **Project** и выполнить одно из следующих действий.

- В панели **Project** щелкнуть на пиктограмме  и из всплывающего меню выбрать команду **Settings...**.
- В панели **Project** щелкнуть на любом файле правой кнопкой мыши и из контекстного меню выбрать команду **Settings...**.

После этого в разделе **Publish** диалогового окна **Project Settings** нужно выбрать требуемый FLA-документ и в списке **Profile** выбрать любой из доступных профилей публикации.

Для публикации отдельного FLA-документа проекта нужно в панели **Project** щелкнуть на нем правой кнопкой мыши и из контекстного меню выбрать команду **Publish**.

Для одновременной публикации всех FLA-документов, входящих в состав проекта, нужно щелкнуть на пиктограмме  и из всплывающего меню выбрать команду **Publish Project** либо щелкнуть правой кнопкой мыши на пиктограмме самого проекта в корне структурного дерева и выполнить команду контекстного меню **Publish**. Файлы, сгенерированные в результате публикации, перезапишут свои предыдущие версии, если они не блокированы. Если существующие конечные документы блокированы, при публикации будет выведено сообщение об ошибке и публикация не будет выполнена.

Тестирование проекта

Для того чтобы протестировать отдельный FLA-документ проекта, нужно щелкнуть на нем правой кнопкой мыши и из контекстного меню выбрать команду **Test**. Это позволит оценить функционирование конечного SWF-документа в среде тестирования.

Для того чтобы протестировать все файлы проекта, необходимо предварительно указать документ, используемый по умолчанию. В качестве такого документа можно указать FLA-документ или HTML-документ, который будет отображен в результате выполнения команды тестирования.

Для того чтобы указать документ по умолчанию, необходимо в панели **Project** щелкнуть на нем правой кнопкой мыши и из контекстного меню выбрать команду **Make Default Document**. Документ по умолчанию отображается в панели **Project** с помощью пиктограммы . Проект может содержать только один документ, используемый по умолчанию.

Для тестирования проекта можно щелкнуть на кнопке **Test Project**, расположенной в нижней части панели **Project**, или, щелкнув на пиктограмме , выбрать из всплывающего меню одноименную команду.

Во время тестирования проекта выполняется публикация всех документов проекта. При этом если в качестве документа по умолчанию используется файл FLA, в среде тестирования будет открыт результирующий SWF-документ. Если же в качестве документа по умолчанию используется файл HTML, сгенерированная при публикации HTML-страница с размещенным на ней SWF-документом будет выведена в окне браузера.

Примечание

Для того чтобы файлы проекта автоматически сохранялись, всякий раз при тестировании или публикации проекта необходимо выполнить команду главного меню **Edit>Preferences** и в появившемся диалоговом окне **Preferences** на вкладке **Editing** установить флажок **Save project files on test project or publish project**.

Управление версиями файлов

Еще одной возможностью панели **Project** является контроль версии файла, входящего в состав проекта (version control). Контроль версии позволяет блокировать файлы проекта, хранящиеся локально, а также обновлять локальные версии более новыми удаленными версиями файлов. Для реализации этих возможностей необходимо включить файлы проекта в сайт и определить (define) или зарегистрировать этот сайт в панели **Project**. Если файл проекта не входит в состав зарегистрированного (определенного) сайта, к нему не может быть применен механизм управления версиями.

Определение сайта

Для того чтобы определить сайт в панели **Project,** необходимо выполнить следующие действия.

1. Создать проект и включить в его состав требуемые файлы.
2. Выполнить команду главного меню **File>Edit Sites...** или щелкнуть на пиктограмме  и из контекстного меню выбрать команду **Edit Sites...**.

3. В диалоговом окне **Edit Sites** нажать кнопку **New...**.
4. В диалоговом окне **Site Definition** заполнить следующие поля (рис. 3.8.):
   - **Site Name** — имя сайта;
   - **Local Root** — локальный адрес корневой папки с файлами сайта;
   - **Email** — почтовый адрес автора;
   - **Check Out Name** — имя пользователя, используемое при снятии блокировки с файла;

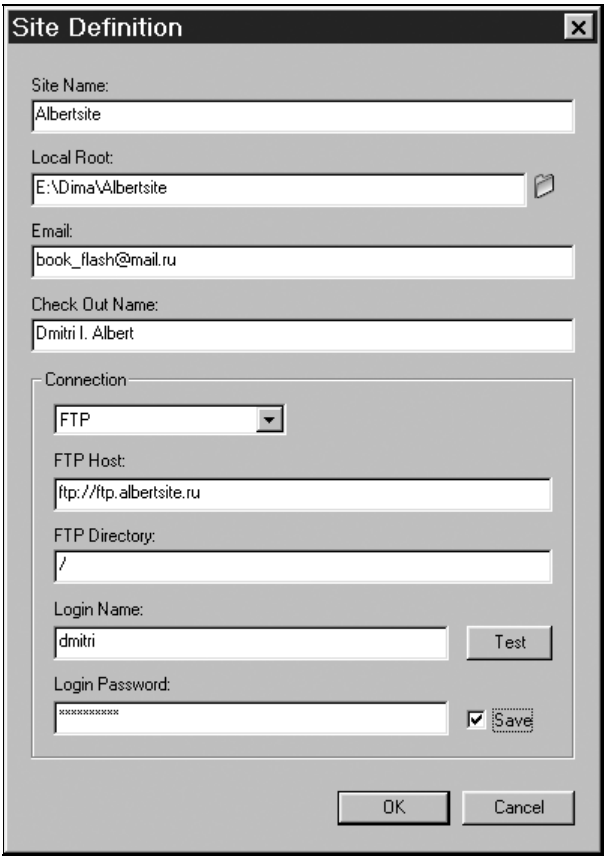


Рис. 3.8. Диалоговое окно **Site Definition**

- **Connection** — параметры соединения, локальное (local/network), FTP или специальный модуль системы управления версиями файлов. При выборе локального соединения в поле **Remote Directory** необходимо указать местоположение корневой папки удаленной копии сайта. При выборе FTP необходимо запол-

нить адрес FTP-сервера (**FTP host**), корневой каталог сайта на FTP-сервере (**FTP Directory**), имя, под которым осуществляется подключение с использование протокола FTP (**Login Name**), пароль доступа (**Login Password**). Для сохранения пароля можно установить флажок **Save**.

5. В панели **Project** щелкнуть на пиктограмме  и из всплывающего меню выбрать команду **Setings...**.
6. В списке **Site** диалогового окна **Project Settings** выбрать имя сайта, определенного на шагах 1—5.

Для того чтобы отредактировать уже определенный сайт, необходимо повторить шаги 1 и 2 и затем в диалоговом окне **Edit Sites** нажать кнопку **Edit...**. Аналогичным образом можно создать копию сайта (**Duplicate**) или удалить определенный сайт (**Remove**).

Реализация управления версиями файлов

Для того чтобы заблокировать локальную копию файла проекта, входящего в состав зарегистрированного сайта, нужно выполнить одно из следующих действий.

- Выделить файл в панели **Project** и, щелкнув на пиктограмме , выбрать команду **Check In** во всплывающем меню.
- Щелкнуть по файлу правой кнопкой мыши и из контекстного меню выбрать команду **Check In**.

В результате выполнения этого действия рядом с именем файла появляется пиктограмма с изображением замка. При этом локальная версия файла сохраняется с атрибутом только для чтения (Read-only), а удаленная копия перезаписывается с учетом внесенных изменений.

В дальнейшем при попытке открыть такой заблокированный файл из панели **Project** Flash выведет сообщение о том, что файл заблокирован, и предложит открыть файл только для чтения (**View**) или разблокировать его (**Check Out**).

Для того чтобы разблокировать файл (например, для внесения в него изменений или публикации), можно выполнить одно из следующих действий.

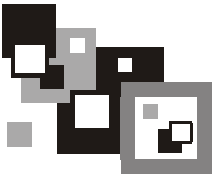
- Выделить файл в панели **Project** и, щелкнув на пиктограмме , выбрать команду **Check Out** во всплывающем меню;
- Щелкнуть по файлу правой кнопкой мыши и из контекстного меню выбрать команду **Check Out**.

В результате разблокирования рядом с именем файла появляется изображение зеленой галочки, а локальная версия файла перезаписывается, при этом файл становится доступным для редактирования. В каталоге локального файла автоматически создается текстовый документ с расширением lck, содержащий имя пользователя и адрес электронной почты. Эти данные указываются на основе информации, введенной на шаге 4 процедуры определения сайта.

Для того чтобы заменить локальную копию файла его удаленной копией, необходимо выполнить следующие действия.

1. Разблокировать требуемый файл.
2. Выполнить команду главного меню **File>Open from Site...**.
3. В списке **Site** диалогового окна **Open from Site** выбрать требуемый сайт.
4. Выбрать удаленный файл, который должен быть открыт, и нажать кнопку **Open**. В результате Flash выведет предупреждающее сообщение о перезаписи файла. Нажатие кнопки **OK** приведет к тому, что удаленный файл перезапишет локальную копию, в результате чего изменения, внесенные в локальную копию, могут быть утрачены.

Глава 4



Переменные

Закладывая что-то в память ЭВМ, помните,
куда вы это положили.

Первая компьютерная аксиома Лео Бейзера

Понятие переменной

При воспроизведении Flash-фильма необходимо непрерывно контролировать информацию о внутренних процессах, протекающих в фильме, и действиях пользователя. Эту информацию нужно сохранять, извлекать, обрабатывать и модифицировать. Для осуществления хранения и управления данными используются переменные — именованные контейнеры для хранения информации.

Переменная (variable) представляет собой ячейку памяти компьютера, имеющую имя и хранящую определенное значение. В данном определении подразумевается абстрактная, а не физическая ячейка памяти, поскольку зачастую для хранения программных переменных требуется более чем одна физическая ячейка памяти.

Переменная имеет два неотъемлемых атрибута: *имя* (name) и *значение* (value). Имя или *идентификатор* переменной — это набор символов, служащих для ее идентификации. Имя используется при обращении к переменной, извлечении хранимого в ней значения и выполнении с этим значением различных операций. Имя переменной должно соответствовать требованиям, предъявляемым к идентификаторам ActionScript *(см. разд. "Имена" главы 3)*. Далее представлены варианты корректных имен переменных:

```
myVar
user1
_totalScore
$color
```

А вот эти имена переменных не являются синтаксически корректными:

```
var // Зарезервированное слово
1user // Имя не может начинаться с цифры
total score //Пробелы в именах запрещены
max-speed // Недопустимый символ в имени
```

Значение переменной — это содержимое ячейки памяти, то есть данные, связанные с переменной. Переменные могут хранить данные различных категорий или *типов*. Тип данных, помещенных в переменную, определяет тип этой переменной. Так, например, в переменной `user1` может храниться имя пользователя (строковая переменная), в переменной `_totalScore` — счет игры (числовая переменная), в переменной `$color` — цвет клипа (переменная абстрактного типа `Color`) и т. д.

Наглядным примером переменной может служить ячейка камеры хранения. Номер, идентифицирующий ячейку, — это имя переменной, а разнообразное содержимое, которое хранится в ячейке, — это ее значение.

Создание и типизация переменной

Создание переменной называется *объявлением* (declaration) или *инициализацией* (initialization). Переменная может быть объявлена *явно* и *неявно*.

Явное объявление переменной

Явное объявление (explicit declaration) переменной содержит перечень создаваемых переменных и указания на тип этих переменных. Для явного объявления переменных используется предложение `var`, имеющее следующий синтаксис:

`var` *`переменная`* `1:`*`тип`* `1, `*`переменная`* `2:`*`тип`* `2, ......, `*`переменная`* `N:`*`тип`* `N`

Здесь *`переменная`* — это имя объявляемой переменной, а *тип* — ее тип. Как следует из этой записи, одно предложение `var` может содержать объявления любого количества переменных.

```
var x:Number; //Объявлена переменная x числового типа
var userName:String; // Объявлена переменная userName строкового типа
var isPlaying:Boolean; /* Объявлена переменная isPlaying логического
(булевского) типа */
```

Эти три предложения можно заменить одним эквивалентным:

```
var x:Number, userName:String, isPlaying:Boolean;
```

Строгая типизация

Указание типа или *строгая типизация* (strict data typing) переменной во время ее объявления — это новая для ActionScript концепция, которая поддерживается только его последней редакцией — ActionScript 2.0. Использование строгой типизации приводит к тому, что переменная в момент создания получает определенный тип и в дальнейшем может содержать данные только этого фиксированного типа. Попытка поместить в строго типизированную переменную данные, тип которых отличен от типа переменной, приводит к ошибке. Ошибка несоответствия типа (type mismatch) генерируется во время компиляции и обнаруживается при автоматической проверке синтаксиса (**Check Syntax**) или в среде тестирования. Так, например, попытка присвоить числовой переменной строковое значение в следующем примере приведет к

ошибке несоответствия типа. Информация об этой ошибке будет выведена в панель **Output** (рис. 4.1).

```
var numVar:Number;
numVar = "Dmitri"; // Ошибка несоответствия типа
```

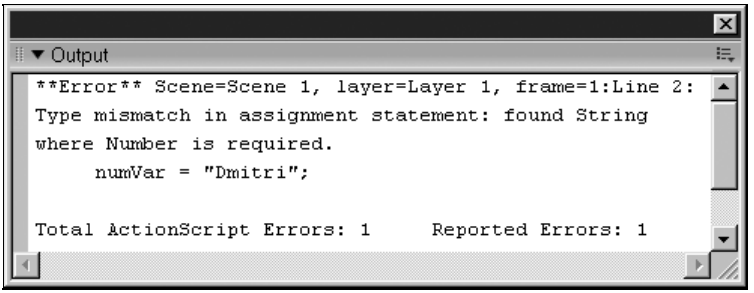


Рис. 4.1. Панель **Output**. Обнаружение ошибки несоответствия типа

Использование строгой типизации не является обязательным. Следующее предложение является совершенно корректным и законным:

```
var x, userName, isPlaying;
```

Тем не менее применение строгой типизации — это чрезвычайно полезная практика, поскольку дает возможность выявления всех неправильно используемых употреблений переменных, приводящих к ошибкам несоответствия типов. Так, например, если строго типизированной переменной типа `Number` попытаться присвоить значение строкового типа (`String`), это вызовет ошибку несоответствия типов. Если же эта операция будет выполнена с переменной, тип которой не был строго указан при объявлении, компилятор не обнаружит ошибки несоответствия типов, переменная получит значение нового типа, и ее тип будет автоматически переопределен. Это демонстрируется в следующем примере.

```
//Использование строгой типизации
var coord:Number;// Объявляем переменную типа Number
coord = 10; // Присваиваем ей числовое значение
coord = "bottom"; /* Присваивание строкового значения приводит к ошибке
несоответствия типа */
//Строгая типизация не используется
var coord; // Объявляем переменную без указания типа
coord = 10; /* Присваиваем ей числовое значение. Переменная coord теперь
имеет тип Number */
coord = "bottom"; /* Присваиваем ей строковое значение. Переменная coord
теперь имеет тип String. */
```

Строгая типизация переменных позволяет не только избежать ошибок несоответствия типа, но и дает преимущество, связанное с появлением всплывающих подсказок

по коду при работе в редакторе ActionScript. При использовании строго типизированной переменной, созданной на основе встроенных классов, точка, поставленная после имени переменной, приводит к появлению всплывающей подсказки, содержащей перечень свойств и методов, доступных для объекта данного класса. Например, если переменная `mc` объявлена с применением строгой типизации

```
var mc:MovieClip;
```

то после того как будут напечатаны символы `mc`, Flash автоматически выведет всплывающую подсказку с перечнем методов и свойств объекта `MovieClip`.

Примечание

Поскольку в ActionScript 1.0 строгий контроль типов не поддерживался, при его использовании необходимо убедиться в том, что в параметрах публикации документа значится ActionScript 2.0.

Определение типа значения

Для определения типа значения, хранимого в переменной, можно использовать унарный (принимающий один операнд) оператор `typeof`, имеющий следующий синтаксис:

```
typeof(выражение)
```

Здесь в качестве параметра `выражение` может использоваться идентификатор либо выражение, возвращающее единичное значение или ссылку на идентификатор, тип значения которого должен быть определен.

Вот несколько примеров использования оператора `typeof`:

```
var n:Number = 10;
trace (typeof(n)); // Выводит number
var s:String = "Dmitri";
trace (typeof(s)); // Выводит string
var b:Boolean = true; // булево значение
trace (typeof(b)); // Выводит boolean
var f:Function = gotoAndStop; // встроенная функция
trace (typeof(f)); // Выводит function
var ar:Array = []; // пустой массив
trace (typeof(ar)); // Выводит object
var mc:MovieClip = clip; // здесь clip - имя экземпляра клипа
trace (typeof(mc)); // Выводит movieclip
var obj:Object ={}; //пустой объект
trace (typeof(obj)); // Выводит object
```

Неявное объявление переменной

Неявное объявление (implicit declaration) переменных выполняется при присваивании значения несуществующей переменной. Когда интерпретатор встречает предложение, в котором выполняется присваивание значения переменной, не объявленной ранее, он автоматически создает новую переменную и помещает в нее это значение. Тип переменной в этом случае определяется типом помещаемого в нее значения. Контроль типов не может быть применен при неявном объявлении.

```
x = 25; //Интерпретатор создает переменную x и присваивает ей значение 25
var i:Number = 5;
var j:Number =3;
sum = i + j //Интерпретатор создает переменную sum и присваивает ей
значение 8
```

Динамическое создание переменной

Динамическое создание переменной позволяет без объявления создать переменную с динамически генерируемым именем. Для этих целей используется предложение `set`, которое имеет следующий синтаксис:

```
set (переменная, значение)
```

Здесь *переменная* — это строка или выражение, возвращающее имя создаваемой переменной, а *значение* — это значение, присваиваемое переменной. Предложение `set` присваивает значение *значение* переменной *переменная*. Если переменной не существует, она создается автоматически. Следующие строки иллюстрируют применение предложения `set`:

```
set ("myVar", 10); // Создает числовую переменную myVar со значением 10
set("login", "Dmitri"); /* Создает строковую переменную login со
значением "Dmitri" */
```

В этом случае переменные создаются без их предварительного объявления, то есть используется неявное объявление. Можно было вначале явно объявить их, однако строгая типизация не действует в предложении `set`. При использовании предложения `set` переменная получает тип, определяемый типом второго параметра *значение*, то есть типизация выполняется автоматически.

```
var myVar:String; // Объявлена строковая переменная myVar
set ("myVar", 10); // Переменной myVar присваивается числовое значение 10
```

Как видно из этого примера, строгая типизация переменной не принесла никакого результата. Несмотря на то, что переменная `myVar` была определена как строковая переменная, в результате выполнения предложения `set` ей было присвоено числовое значение. Переменная изменила тип без вызова ошибки несоответствия типа.

Предложение set позволяет программным образом генерировать имя создаваемой переменной, как это показано в следующем примере:

```
var varName:String = "ball";
var varNum:Number = 1;
var varValue:Number = 100;
set (varName + "_" + varNum, varValue); /* Создается переменная ball_1 со значением 100 */
```

В данном случае имя переменной задается выражением, которое разрешается в строку "ball_1", а в качестве значения создаваемой переменной используется значение переменной varValue. В результате создается переменная ball_1. При этом значения переменных varName, varNum, varValue остаются неизменными.

Присваивание, извлечение и изменение значения переменной

При объявлении переменной она автоматически получает особое значение undefined, указывающее на то, что значение переменной не определено. В большинстве случаев требуется, чтобы переменная содержала какое-то иное более полезное значение. Задание значения переменной называется *присваиванием* (assignment).

Присваивание значения

Присваивание значения выполняется с помощью оператора присваивания (=). При этом идентификатор (имя переменной) должен находиться слева от оператора присваивания, а присваиваемое значение — справа. Так, например запись

```
x = 25; // Верно
```

является корректной и указывает интерпретатору на то, что переменной x присваивается значение 25, а запись

```
25 = x; // Неправильно !!!
```

вызовет синтаксическую ошибку.

Присваивание можно объединить в одном предложении с объявлением переменной. Причем одновременно можно объявить несколько переменных и тут же присвоить им соответствующие значения:

```
var firstName:String = "Dmitri", lastName:String = "Albert";
```

При необходимости присвоить нескольким переменным одно и то же значение это также можно сделать в одном предложении. При этом необходимо либо явно объявить эти переменные заранее, либо использовать неявное объявление:

```
var x:Number;
var y:Number;
```

```
var z:Number;
x = y = z = 0; // Все переменные получают значение 0
```

В этом случае присваивание выполняется справа налево. Сначала переменная `z` получает значение `0`, затем переменной `y` присваивается значение переменной `z`, затем переменной `x` присваивается значение переменной `y`.

Для того чтобы программным образом вывести значение переменной в панель **Output**, нужно использовать функцию `trace()` *(см. главу 3)*. С ее помощью можно определить, какое значение принимает переменная в любом месте программы. Это позволяет проконтролировать, насколько верно выполняется программа, и часто помогает найти логические ошибки.

```
var a = 2;
trace (a); // Выводит 2
var b = 3;
trace (b); // Выводит 3
trace ((a+b)/2); // Выводит 2.5
trace (c); // Выводит undefined, так как переменная c еще не определена
var c = (a+b)/2;
trace (c); // Теперь выводит 2.5
```

Примечание

В среде тестирования в панель **Output** можно вывести список всех переменных, (объявленных и/или имеющих значение) и их значений с помощью команды **Debug>List Variables** (<Ctrl>+<Alt>+<V>). Это позволяет оценить функционирование сценария и часто помогает выявить причину ошибок в алгоритме.

В случае если переменная уже содержит какое-то значение, присваивание нового значения заменяет текущее значение. Таким образом, значение переменной может постоянно обновляться при каждом новом присваивании.

```
var x:Number = 5;
x = 10; // Новое значение 10
x = 15; // Новое значение 15
var y:Number = 20;
x = y; // Присваиваем переменной x значение переменной y
trace (x); // В панель Output выводится 20
```

Важно иметь в виду, что если тип переменной был строго указан при ее объявлении, то в дальнейшем такая переменная может принимать значения исключительно данного типа. В противном случае компилятор выдаст ошибку несоответствия типа. Если же переменная не была строго типизирована, то ее тип может изменяться в зависимости от типа вновь присваиваемого ей значения.

Автоматическая типизация переменной

Если тип переменной не указывается явно при ее объявлении (то есть строгая типизация не используется), тип переменной назначается *автоматически* в момент присваивания ей значения. Тип определяется интерпретатором, исходя из типа присваиваемых данных. Поскольку в ActionScript существуют жесткие синтаксические правила оформления литералов (то есть буквального описания) различных типов данных, тип всегда может быть однозначно определен на основании анализа вида литерала (О правилах оформления литералов см. в *главе 5*):

```
var x = 3.1415926 ; // Переменная x имеет тип Number (числовой)
var login = "Dmitri"; // Переменная имеет тип String (строковый)
var myArray = [1, 2, 3, 4, 5]; // Переменная имеет тип Array (массив)
```

Переменная, тип которой не был указан при объявлении, может динамически изменять тип в зависимости от типа помещаемых в нее данных. Присваивание не строго типизированной переменной, содержащей данные одного типа, нового значения другого типа не вызывает ошибки. Интерпретатор просто приводит тип переменной в соответствии с ее новым значением.

```
var x = 3.1415926 ; // Переменная x имеет тип Number
x = "Dmitri"; // Переменная x теперь имеет тип String
x = [1, 2, 3, 4, 5]; // Переменная x теперь имеет тип Array
```

Автоматическое преобразование типа значения, хранимого в переменной, может иметь место в выражениях, когда контекст ситуации требует определенного значения. Для пояснения сказанного рассмотрим следующий пример:

```
var a:Number = 2;
var b:String = "3";
var sum = a + b;
trace(sum); // Выводит 23, а не 5
```

Поскольку в выражении `a + b` производится попытка сложить числовое значение и строковое значение, число `2` здесь преобразуется в строку `"2"`, после чего два строковых значения *конкатенируются* или соединяются, образуя строку `23`. Такое преобразование объясняется тем, что, если один из операндов оператора сложения (+) является строкой, другой также преобразуется в строковый тип (О правилах преобразования данных см. *главы 4 и 5*). Если вместо оператора сложения использовать оператор умножения, приоритет будет отдан числовому типу:

```
var product = a * b;
trace(product); // Выводит 6
```

Важно отметить, что такое преобразование типа осуществляется не над самой переменной, используемой в выражении, а над *копией* хранимых в ней данных. Так, в предыдущих примерах типы переменных `a` и `b` остались без изменения.

```
trace (typeof(a)); //Выводит number
trace (typeof(b)); //Выводит string
```

Извлечение значения

Для того чтобы извлечь хранимое в переменной значение, необходимо использовать имя этой переменной. Ссылка на переменную в выражении всегда заменяется ее значением (за исключением ситуации, когда имя переменной указывается слева от оператора присваивания). По этой причине справа от оператора присваивания может находиться выражение, состоящее не только из литералов (то есть самих данных), но также включающее идентификаторы — ссылки на некоторые сущности, объявленные в программе. Во время вычисления такого выражения интерпретатором идентификаторы заменяются своими значениями, после чего выражение разрешается в конкретное результирующее значение. Если выражение содержит данные различных типов, интерпретатор попытается преобразовать их к одному типу. Правила таких преобразований см. в *главе 4*.

Ниже приводится пример, в котором в переменную `triangleArea` помещается значение площади треугольника, вычисленное на основе значений его высоты и основания, хранящихся соответственно в переменных `height` и `base`.

```
var height:Number = 10;
var base:Number = 5;
var triangleArea:Number = (height * base)/2;
trace (triangleArea); // Выводит 25
```

Если переменная не содержит значения или переменной не существует, но ее имя используется в выражении, то в качестве значения такой переменной используется `undefined`.

Область видимости и адресация переменных

Область видимости — это одна из важнейших характеристик переменной. *Область видимости* (scope) определяет, в каком фрагменте программного кода переменная является видимой или доступной элементам этого кода. В связи с тем, что программный код ActionScript может быть рассредоточен по разным структурным элементам документа, вопросы области видимости являются чрезвычайно важными. Необходимо четко представлять, в каких пределах переменная является непосредственно доступной операторам программы и каким образом можно обратиться к переменной из блока кода, лежащего за пределами ее области видимости.

Вопросы области видимости

При создании переменной она становится доступна или *видима* сценариям монтажной линейки, содержащей объявление этой переменной. При этом говорится, что переменная *прикрепляется* к данной монтажной линейке. В этом разделе будут перечислены основные аспекты, разъясняющие вопросы области видимости переменных.

- Переменная является доступной из любого фрагмента кода, содержащего ее объявление, расположенного после инициализации этой переменной. Так, напри-

мер, если в первом ключевом кадре основной монтажной линейки объявлена переменная с именем myVar, то к ней можно непосредственно обратиться из любого предложения сценария первого кадра фильма, расположенного после объявления этой переменной.

- После инициализации переменная является доступной для любого кадра монтажной линейки, содержащей ее объявление. Пусть, например, в первом кадре основной монтажной линейки фильма содержится объявление переменной:

```
var myVar:Number = 3;
```

Поместим следующее предложение в десятый кадр этой же монтажной линейки:

```
trace(myVar);
```

В панель **Output** будет выведено значение 3, то есть переменная myVar, объявленная в первом кадре, видна из десятого.

Теперь попробуем модифицировать этот пример и поменять кадры местами так, что первый кадр будет содержать предложение:

```
trace(myVar);
```

а в десятом будет выполняться инициирование переменной:

```
var myVar:Number = 3;
```

Теперь при первом воспроизведении в панели **Output** появится значение undefined. Это совершенно логично, так как в первом кадре переменной myVar еще не существует. Однако затем воспроизводящая головка входит в десятый кадр, где происходит инициализация переменной, поэтому при повторном воспроизведении в панели **Output** появляется значение 10. Теперь, после того как переменная объявлена и ей присвоено значение, эта переменная стала доступна сценариям всех кадров основной монтажной линейки.

- Переменная, объявленная на основной монтажной линейке, также является доступной для сценария любого кадра основной монтажной линейки, в том числе, расположенного на другой сцене (Scene).
- Переменная, объявленная на монтажной линейке, после своей инициализации доступна всем сценариям кнопок (то есть сценариям, прикрепленным к экземплярам символов типа Button), расположенных на этой же монтажной линейке. Так, например, пусть в первом кадре объявлена переменная:

```
var myVar:Number = 3;
```

Пусть в этом же или любом из следующих кадров находится кнопка, к которой прикреплен следующий сценарий:

```
on(press){
    trace(myVar);
}
```

При нажатии на кнопку в среде тестирования в панели **Output** будет выведено значение 3.

- Переменная, объявленная в сценарии кнопки, после инициализации доступна всем сценариям кадров монтажной линейки, содержащей эту кнопку.
- Переменная, объявленная в сценарии, прикрепленном к кадру монтажной линейки клипа или непосредственно к экземпляру клипа, доступна только в пределах монтажной линейки этого клипа. Область видимости такой переменной ограничена монтажной линейкой клипа, содержащего ее объявление. Так, например, если в первом кадре монтажной линейки находится клип, содержащий на своей собственной монтажной линейке объявление переменной, то эта переменная доступна только сценариям кадров монтажной линейки клипа или сценарию, прикрепленному к самому клипу. Однако такая переменная не видна сценариям основной монтажной линейки, сценариям кнопок или сценариям других клипов.
- Переменная, объявленная на основной монтажной линейке (в кадре или в сценарии кнопки), непосредственно недоступна сценариям кадра монтажной линейки клипа или сценариям самих клипов.

Локальные переменные

Локальная (local) переменная — это переменная, которая существует только в пределах определенного фрагмента кода. Область видимости такой переменной ограничена фрагментом кода, содержащего объявление локальной переменной.

Локальные переменные создаются при объявлении переменой с помощью предложения `var` в теле функции или метода. Речь о пользовательских функциях подробно пойдет в *главе 8*, в настоящем разделе мы вынуждены коснуться этой пока незнакомой темы для того, чтобы рассказ о переменных был полным.

В следующем примере создается функция `myFunc()`, в которой объявляются две локальные переменные `a` и `b`.

```
function myFunc (){
var a = 5;
var b = 10;
       }
```

Локальность переменных `a` и `b` заключается в том, что они создаются при вызове функции `myFunc()` и существуют только на протяжении времени ее выполнения. Локальные переменные недоступны из кода, расположенного вне тела функции. В этом можно убедиться, добавив после определения функции следующее предложение:

```
trace (a) // Выводит undefined
trace (b) // Выводит undefined
```

В панель **Output** выводится значение `undefined`, так как переменные существуют только в пределах функции во время ее выполнения. Если предложение `trace()` поместить в тело функции, то результат будет иным:

```
function myFunc (){
var a = 5;
```

```
var b = 10;
trace (a);
trace (b);
       }
myFunc(); // Вызываем функцию. Выводится 5 и 10
```

Локальные переменные используются в ситуациях, когда помещенные в них данные нужны только во время выполнения функции. После выполнения функции определенные в ней локальные переменные автоматически удаляются.

Рассмотренная в предыдущем примере функция на практике бесполезна, так как она не выполняет никаких действий. В следующем примере создается функция, позволяющая расположить десять экземпляров клипов с именами `mc0`, `mc1`, ..., `mc9` по горизонтали на расстоянии 10 пикселов друг от друга.

```
function spaceHor (xDist){
       for (var i=0; i<10; i++){
       this["mc"+i]._x=(this["mc"+i]._width+xDist)*i;
       }
}
```

Переменная `i`, исполняющая роль счетчика цикла, объявлена как локальная переменная (с использованием предложения `var`). В связи с этим после выполнения функции переменная `i` будет автоматически удалена, и попытка обратиться к ней извне окажется безрезультатной.

```
spaceHor(25); /* Вызываем функцию, передавая ей в качестве параметра
расстояние по горизонтали */
trace (i); /* Выводит undefined, так как переменной i не существует вне
определения функции */
```

Использование локальной переменной в данном примере оправдано, поскольку переменная-счетчик цикла требуется только на протяжении выполнения итераций, то есть она, как правило, не используется за пределами тела цикла. Поэтому после того как выполняется требуемое количество итераций цикла, переменная `i` становится равной 10 и удаляется из памяти, поскольку функция прекращает свое действие.

Если в определении функции переменная `i` будет объявлена как нелокальная, то есть без использования предложения `var` (неявное объявление), то к ней можно будет обратиться из любого сценария, прикрепленного к монтажной линейке, содержащей объявление функции.

```
function spaceHor (xDist){
       for (i=0; i<10; i++){
       this["mc"+i]._x=(this["mc"+i]._width+xDist)*i;
       }
}
spaceHor(25);
trace (i); // Выводит 10
```

В данном случае переменная i не является локальной. Она создается внутри объявления функции, однако не удаляется из памяти после ее выполнения и лежит в области видимости, ограниченной монтажной линейкой, содержащей определение функции. Таким образом, если переменная неявно объявляется внутри функции (без использования предложения var), то такая переменная не является локальной и прикрепляется к монтажной линейке, содержащей объявление функции.

Необходимо хорошо понимать, что если внутри функции используется неявное объявление переменной, то происходит создание переменной монтажной линейки. Такого же результата можно добиться, явно объявляя переменную вне определения функции. Ниже приводится функция, которая создает переменную монтажной линейки. Функции передаются два параметра: строка с именем переменной и само значение переменной.

```
function setVariable(varName:String, varValue) {
        this[varName] = varValue;
}
setVariable("myVar", 10); /* Создаем переменную монтажной линейки с
именем myVar и значением 10 */
trace(myVar) // Выводит 10
```

В этом примере предложение this[varName] = varValue создает переменную с именем varName на текущей монтажной линейке и присваивает ей значение varValue. Подробности синтаксиса, используемого в этой записи, будут разъяснены в *разд. "Адресация переменных" настоящей главы.*

Соответственно значение переменной, объявленной вне функции, можно легко изменить, присвоив ей новое значение внутри функции, как это показано в следующем примере:

```
var x:Number = 5; // Переменная монтажной линейки
function setValue(val){
        x = val; /* Модификация значения переменной x, объявленной вне
        функции */
}
setValue(25); /* вызываем функцию, передавая ей значение в качестве
параметра */
trace (x); // Выводит 25. Переменная x теперь имеет значение 25
```

Теоретически возможна ситуация, когда имя локальной переменной совпадает с именем переменной монтажной линейки. При этом внутри функции локальная переменная перекрывает переменную монтажной линейки.

```
var myVar:Number = 5;
function localCheck(){
        var myVar:String = "Flash";
        trace (myVar+": это локальная переменная");
}
```

```
localCheck(); // Выводит Flash: это локальная переменная
trace(myVar+": это не локальная переменная"); /* Выводит 5: это не
локальная переменная */
```

Рассмотренной ситуации по возможности желательно избегать, поскольку использование одинаковых имен для локальных и нелокальных переменных может привести к путанице и явиться причиной ошибки.

Применение локальных переменных бывает часто необходимо при объявлении методов в пользовательских классах. Если внутри метода возникает необходимость в сохранении каких-либо данных в переменной и в классе не определено соответствующего свойства для хранения этих данных, то нужно прибегнуть к использованию локальной переменной.

Глобальные переменные

В определенном смысле переменные монтажных линеек локальны в пределах своей области видимости, поскольку к ним нельзя непосредственно обратиться из произвольного сценария. В некоторых случаях возникает необходимость в создании переменных, непосредственно доступных любому сценарию фильма и видимых в пределах любой области видимости. Такие переменные называются *глобальными* (global).

Для создания глобальной переменной ее необходимо поместить в глобальный объект, содержащий конструкторы всех основных встроенных классов ActionScript, таких как `Number`, `String`, `Function`, `Array`, `MovieClip` и многие другие. Для того чтобы это сделать, нужно использовать идентификатор `_global`, являющейся ссылкой на глобальный объект. Для создания глобальной переменной используется следующий синтаксис:

```
_global.имя = значение;
```

В следующем предложении создается глобальная переменная `glVar` со значением 10:

```
_global.glVar = 10;
```

В отличие от обычных переменных создание глобальной переменной осуществляется без применения предложения `var`. Также следует отметить, что при создании глобальной переменной не может быть использована строгая типизация. В связи с этим следующее предложение вызовет ошибку:

```
var _global.glVar:Number = 10; // Синтаксическая ошибка !!!
```

К глобальной переменной можно обратиться непосредственно по имени с любой монтажной линейки. Так, пусть в первом кадре основной монтажной линейки создана глобальная переменная:

```
_global.glVar = 10;
```

В кадре монтажной линейки любого клипа поместим следующее предложение:

```
trace(glVar); // Выводит 10
```

Таким образом, глобальная переменная видна из сценария любой монтажной линейки. Исключение составляет ситуация, когда на монтажной линейке имеется соб-

ственная переменная с таким же именем, как и у глобальной переменной. В этом случае локальная переменная (переменная монтажной линейки) перекрывает свою глобальную тезку. Такая ситуация представлена ниже.

```
//Сценарий основной монтажной линейки
_global.glVar = 10;
// Сценарий кадра монтажной линейки клипа
glVar = "Local scope";
trace(_global.glVar); // Выводит Local scope
```

В данном случае местная переменная `glVar` перекрывает глобальную переменную `glVar`.

Для того чтобы в данном случае с монтажной линейки клипа обратиться к глобальной переменной, нужно использовать идентификатор `_global`.

```
//Сценарий основной монтажной линейки
_global.glVar = 10;
// Сценарий кадра монтажной линейки клипа
glVar = "Local scope";
trace(_global.glVar); // Выводит 10
```

При необходимости модификации (изменения) значения глобальной переменной также необходимо использовать идентификатор `_global`. В противном случае вместо присваивания нового значения глобальной переменной будет создаваться переменная монтажной линейки с тем же именем.

```
//Сценарий основной монтажной линейки
_global.glVar = 10;
// Сценарий кадра монтажной линейки клипа
_global.glVar = 25; // Глобальная переменная теперь имеет значение 25
trace(_global.glVar); // Выводит 25
_global.glVar = "Flash"; /* Глобальная переменная теперь имеет значение
"Flash" */
trace(_global.glVar); // Выводит "Flash"
glVar = true; // Создается переменная монтажной линейки клипа
trace(_global.glVar); /* Выводит "Flash". Значение глобальной переменной
не изменилось */
```

Примечание

Глобальные переменные сохраняются на протяжении всего времени работы проигрывателя Flash Player. Если текущий документ заменяется динамически загруженным в проигрыватель (например, с помощью метода `MovieClip.loadMovie()`) внешним документом, все переменные монтажных линеек удаляются, однако переменные, определенные в `_global`, продолжают существовать и являются доступными всем сценариям вновь загруженного документа.

Применение глобальных переменных не является универсальной практикой. Очень часто используемая в одном сценарии переменная либо просто не должна быть видна другим сценариям, либо в этом нет необходимости. На практике глобальные переменные используются для хранения информации, которая должна быть непосредственно доступна всем сценариям на протяжении всего фильма. Например, в `_global` можно поместить данные о настройках фильма, выбранных пользователем, или информацию, позволяющую управлять порядком загрузки (выгрузки) SWF-документов в проигрыватель Flash Player. При создании классов в стиле ActionScript 1.0 объект `_global` используется для управления доступом к членам класса. При решении рядовых задач если используемая переменная необходима только в пределах ограниченной области видимости (например, только на конкретной монтажной линейке), нет смысла помещать ее в `_global`.

В табл. 4.1, 4.2 обобщена информация о типах и областях видимости переменных.

Таблица 4.1. *Типы переменных*

Параметры	Типы переменных		
	Глобальная	**Переменная монтажной линейки**	**Локальная**
Определение	Переменная, непосредственно доступная любому сценарию фильма	Переменная, прикрепленная к монтажной линейке	Переменная, явно объявленная в теле функции
Объявление	Прикрепляется к объекту `_global`. Объявляется без применения предложения var. Тип строго не указывается	Объявление переменной с помощью предложения var или без него. Тип может быть строго указан	Объявление с помощью предложения `var` в теле функции. Тип может быть строго указан
Область видимости	Доступна любому сценарию фильма	Непосредственно доступна сценариям родительской линейки. Для обращения с других линеек используется адресация	Доступна только внутри тела функции
Время жизни	От создания до завершения работы проигрывателя	На протяжении времени существования родительской линейки	На протяжении времени выполнения функции
Назначение	Хранение данных, необходимых на протяжении всей работы проигрывателя	Хранение информации в пределах документа	Хранение информации, необходимой только в процессе выполнения функции

Таблица 4.2. *Области видимости переменных*

Доступность[1] переменной		Расположение сценария, содержащего обращение к переменной				
		Кадр основной монтажной линейки любой сцены	Сценарий кнопки, расположенной на основной монтажной линейке	Сценарий, прикрепленный к кадру монтажной линейки клипа	Сценарий, прикрепленный непосредственно к экземпляру клипа	Тело функции[2]
Место объявления переменной	Кадр основной монтажной линейки	+	+	–	–	+
	Сценарий кнопки, расположенной на основной монтажной линейке	+	+	–	–	+
	Сценарий, прикрепленный к кадру монтажной линейки клипа	–	–	+	+	+
	Сценарий, прикрепленный непосредственно к экземпляру клипа	–	–	+	+	+
	Тело функции	–	–	–	–	+
	Объект `_global`	+	+	+	+	+

[1] Условные обозначения: "+" — переменная доступна для непосредственного обращения к ней, "-" — переменная недоступна. — Прим. автора.

[2] Предполагается, что место объявления функции совпадает с местом объявления переменной. — Прим. автора.

Адресация переменных

В связи с тем, что область видимости переменной ограничена ее монтажной линейкой, к такой переменной нельзя непосредственно обратиться для извлечения или изменения ее значения. Тем не менее зачастую при разработке сценариев возникает необходимость обратиться из сценария одной монтажной линейки к переменной, прикрепленной к другой монтажной линейке. В этом случае необходимо либо работать с глобальными переменными, либо использовать *адресацию*. Адресация позволяет обратиться с одной монтажной линейки к переменной (или объекту), прикрепленной к другой монтажной линейке. Для того чтобы это сделать, необходимо перед указанием имени адресуемой переменной включить путь к этой переменной. Переменная (или объект), к которой осуществляется обращение, называется *целевой*

(target), а путь к ней называется *целевым путем* (target path). Принципиально это действует так же, как адресация файлов в файловой структуре. Для обращения к одному файлу из некоторого местоположения в файловой системе необходимо указать путь к этому файлу.

Все переменные в фильме, за исключением локальных и глобальных, прикрепляются либо к основной монтажной линейке, либо к монтажной линейке клипа. Еще раз резюмируем сведения об области видимости переменных.

- Переменные, объявленные в кадре основной монтажной линейки, прикрепляются к основной монтажной линейке.
- Переменные, объявленные в сценарии кнопки, прикрепляются к монтажной линейке, содержащей эту кнопку.
- Переменные, объявленные в кадре монтажной линейки клипа, прикрепляются к монтажной линейке этого клипа.
- Переменные, объявленные в сценарии самого клипа, также прикрепляются к монтажной линейке этого клипа.

Для использования адресации необходимо, чтобы клип, к переменной монтажной линейки которого требуется обратиться, имел идентификатор. В качестве идентификатора клипа используется *имя экземпляра* (instance name). Для того чтобы задать имя экземпляра, необходимо выделить экземпляр клипа на сцене и ввести требуемый набор символов в поле **Instance Name**, расположенное в левом верхнем углу панели **Properties**. Имя клипа должно соответствовать требованиям, предъявляемым к идентификаторам ActionScript *(см. разд. "Имена" главы 3)*.

При адресации объектов в ActionScript используется *точечный синтаксис* (dot syntax). Для того чтобы обратиться к объекту, расположенному внутри другого объекта, необходимо указать имя внешнего объекта, далее следует точка, после которой указывается имя вложенного объекта. Так, если на монтажной линейке клипа `mc` объявлена переменная `myVar`, то к этой переменной можно обратиться следующим образом:

```
mc.myVar
```

При использовании точечного синтаксиса для указания пути к целевому объекту используются идентификаторы, расположенные в соответствии со структурой вложенности представляемых ими объектов, разделенные точками. Так, если на основной монтажной линейке имеются три последовательно вложенных друг в друга клипа с именами `mc1`, `mc2`, `mc3` и на монтажной линейке клипа `mc3` объявлена переменная, то обратиться к переменной `myVar` из сценария основной монтажной линейки можно, используя следующий синтаксис:

```
mc1. mc2. mc3.myVar
```

Важно отметить, что в этой записи `mc1`, `mc2` и `mc3` — это идентификаторы, содержащие ссылки на конкретные объекты (клипы), а `myVar` — идентификатор переменной, содержащий ссылку на хранимое в ней значение.

Кроме использования точечного синтаксиса, обратиться к переменной можно с помощью оператора доступа к элементам массива `[]` (квадратные скобки). Дело в том, что любая переменная, объявленная на основной монтажной линейке или на монтажной линейке клипа, интерпретируется как свойство содержащего ее объекта.

Аналогично клип, помещенный на монтажную линейку другого клипа, интерпретируется как свойство родительского (то есть содержащего его) клипа. Доступ к свойству объекта может быть получен либо с помощью точечного синтаксиса, либо с помощью оператора `[]`. Во втором случае необходимо использовать следующий синтаксис:

```
объект["свойство "]
```

Здесь `свойство` — это строка (то есть набор символов, заключенный в кавычки) или выражение, выдающее строку. Так, наш предыдущий пример с вложенными клипами может быть переписан следующим образом:

```
this["mc1"]["mc2"]["mc3"]["myVar"]
```

Здесь `this` — это идентификатор, возвращающий ссылку на текущий объект, то есть на объект, к которому прикреплен сценарий, содержащий `this`. Об использовании идентификатора `this` см. в *разд. "Относительная адресация" настоящей главы.*

Примечание

Применение оператора `[]` при адресации переменных или объектов бывает необходимо, когда нужно динамически генерировать имя целевой переменной или объекта. Поскольку в квадратных скобках должно находиться выражение, возвращающее строку, это дает возможность программно генерировать имена свойств или целевых объектов. С помощью оператора `[]` можно динамически обращаться к методам и свойствам объектов или динамически создавать их. Эта возможность будет широко использоваться в дальнейшем при работе с клипами.

Существует два различных подхода к адресации переменных (и клипов): абсолютная и относительная адресация.

Абсолютная адресация

Абсолютная (absolute) адресация предполагает указание пути, начиная с самого верхнего (корневого) уровня. Если вновь вернуться к аналогии с файловой системой, то абсолютный путь всегда начинается с указания логического диска, например: **E:\Dima\Book_AS\Actions\Classes**.

Корневым элементом Flash-фильма является основная монтажная линейка. Для обращения к основной монтажной линейке используется глобальное свойство `_root`. Применение `_root` в любом сценарии всегда возвращает ссылку на основную монтажную линейку. Таким образом, при использовании абсолютной адресации путь к переменной или объекту всегда начинается с `_root`. Далее необходимо перечислить вложенные объекты в соответствии с иерархией их вложенности, разделяя соседние уровни вложенности точкой.

Рассмотрим следующий пример. Пусть на основной монтажной линейке находится клип `mc1`, который на своей монтажной линейке содержит клип `mc2`, на монтажной линейке которого, в свою очередь, расположен клип `mc3`. Эта иерархия представлена на рис. 4.2.

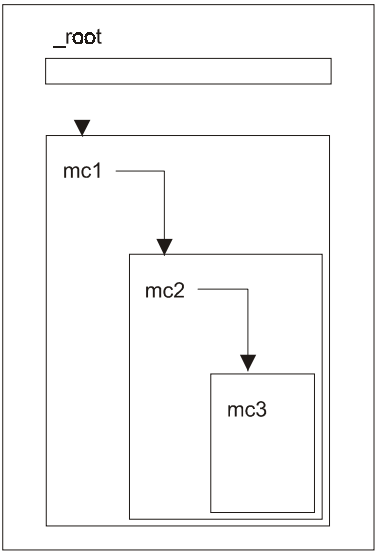


Рис. 4.2. Иерархия вложенности клипов

Если на монтажной линейке клипа `mc3` объявлена переменная `myVar`, то к ней можно обратиться следующим образом:

```
_root.mc1.mc2.mc3.myVar;
```

Используя адресацию, можно легко присвоить значение переменной, расположенной на другой монтажной линейке, изменить это значение или даже создать новую переменную. Так, если на монтажной линейке `mc3` уже объявлена переменная `myVar`, то следующая строка кода присвоит ей значение 10; если же такой переменной не существует, то она автоматически будет создана на монтажной линейке клипа `mc3`.

```
_root.mc1.mc2.mc3.myVar = 10;
```

При создании переменной на другой монтажной линейке нельзя использовать предложение `var` и, соответственно, нельзя строго указать тип этой переменной.

```
var _root.mc1.mc2.mc3.myVar:Number = 10; // Синтаксическая ошибка !!!
```

Характерной особенностью абсолютной адресации является независимость пути от местоположения сценария, содержащего ссылку на внешний объект. Все абсолютные ссылки всегда разрешаются относительно самого верхнего уровня, то есть `_root`. В связи с этим для правильного указания пути необходимо только верно указать местоположение целевого объекта (переменной или клипа) относительно `_root`, местоположение сценария, содержащего ссылку, не оказывает никакого влияния на целевой путь, поэтому такая система адресации и называется абсолютной. Это иллюстрируется следующим примером.

В сценарии монтажной линейки клипа mc3 создадим числовую переменную myVar.

```
var myVar:Number = 1; // Создаем переменную в клипе mc3
```

Изменим значение переменной myVar с помощью сценария основной монтажной линейки, для этого поместим в кадр основной монтажной линейки следующий код.

```
_root.mc1.mc2.mc3.myVar = 2; /* Изменяем значение переменной myVar из
сценария основной монтажной линейки */
```

Теперь изменим значение переменной myVar с помощью сценария монтажной линейки клипа mc1.

```
_root.mc1.mc2.mc3.myVar = 3; /* Изменяем значение переменной myVar из
сценария монтажной линейки клипа mc1*/
```

И, наконец, изменим значение переменной myVar с помощью сценария монтажной линейки клипа mc2.

```
_root.mc1.mc2.mc3.myVar = 4; /* Изменяем значение переменной myVar из
сценария монтажной линейки клипа mc2*/
```

Как видно из этого примера, путь к переменной myVar выглядит абсолютно одинаково вне зависимости от того, где в структуре документа находится сценарий, содержащий ссылку на целевую переменную.

Вместо использования точечного синтаксиса можно прибегнуть к применению оператора []. В этом случае абсолютный путь к переменной myVar из только что рассмотренного примера будет выглядеть следующим образом:

```
_root["mc1"]["mc2"]["mc3"]["myVar"]
```

При необходимости можно использовать комбинацию точечного синтаксиса и квадратных скобок:

```
_root["mc1"].mc2.mc3["myVar"] = 3.1415926;
```

Относительная адресация

Относительная (relative) адресация, в отличие от абсолютной, всегда предполагает указание пути относительно текущего уровня структуры. Относительные ссылки разрешаются относительно местоположения содержащего их сценария. Таким образом, относительный путь определяется исключительно взаимным расположением сценария, содержащего ссылку, и целевого объекта. Если взглянуть на иерархию, представленную на рис. 4.2, то становится очевидно, что при необходимости обратиться из сценария клипа mc2 к переменной, объявленной в клипе mc3, значительно проще указать путь просто от mc2 к mc3, чем начинать адресацию с _root. Такой путь будет называться относительным.

При использовании абсолютной адресации всегда происходит спуск из _root к требуемому уровню вложенности. Когда же речь идет об относительной адресации, то здесь может потребоваться как спуститься на более низкий уровень иерархии, так и подняться на более высокий (например, если нужно из клипа mc3 обратиться к переменной клипа mc2).

Для обращения к более низким уровням иерархии используется уже знакомый нам точечный синтаксис или оператор `[]`. Вот как, например, можно обратиться к переменной `myVar`, объявленной в клипе `mc3` из сценария клипа `mc1` (см. рис. 4.2):

```
mc2.mc3.myVar = 25;
```

или, что то же самое:

```
mc2["mc3"].myVar = 25;
```

Для того чтобы обратиться к более высокому уровню иерархии, используется идентификатор `_parent`. `_parent` возвращает ссылку на родительский по отношению к объекту, использующему этот идентификатор, объект. Так, в клипе `mc1` идентификатор `_parent` ссылается на `_root`; в клипе `mc2` идентификатор `_parent` ссылается на `mc1`, в клипе `mc3` идентификатор `_parent` ссылается на `mc2`. Если необходимо подняться сразу на несколько уровней, можно расположить требуемое количество идентификаторов `_parent`, разделяя их точкой. Рассмотрим это на конкретном примере.

Пример.

В сценарии монтажной линейки клипа `mc1` создадим числовую переменную `myVar`.

```
var myVar:Number = 10; // Создаем переменную в клипе mc1
```

Изменим значение переменной `myVar` с помощью сценария основной монтажной линейки, используя относительную адресацию.

```
mc1.myVar = 20; /* Изменяем значение переменной myVar из сценария
основной монтажной линейки */
```

Теперь изменим значение переменной `myVar` с помощью сценария монтажной линейки клипа `mc2`.

```
_parent.myVar = 30; /* Изменяем значение переменной myVar из сценария
монтажной линейки клипа mc2*/
```

И, наконец, изменим значение переменной `myVar` с помощью сценария монтажной линейки клипа `mc3`.

```
_parent. _parent.myVar = 40; /* Изменяем значение переменной myVar из
сценария монтажной линейки клипа mc3*/
```

Как видно из этого примера, путь к переменной `myVar` изменяется в зависимости от местоположения сценария, в котором осуществляется обращение. В этом выражается различие относительной и абсолютной адресации.

Следует отметить, что при использовании относительной адресации можно обратиться к переменной клипа, не имеющего имени. Так, например, если переменная `myVar` объявлена в анонимном клипе, расположенном на том же уровне, что и клип `mc2` на рис. 4.2, то можно обратиться к этой переменной из клипа `mc3`, не указывая его имени:

```
_parent.myVar
```

Для ссылки на текущий объект при использовании относительной адресации можно использовать идентификатор `this`. `this` содержит ссылку на объект, в сценарии которого используется этот идентификатор. Так, например, для того чтобы из сце-

нария клипа обратиться к переменной myVar, объявленной на монтажной линейке этого же клипа, можно использовать следующую запись:

```
this.myVar = 20;
```

или

```
this["myVar"] = 20;
```

Зачастую возникает необходимость явно обратиться к объекту из его же сценария. Как раз в таких ситуациях и применяется идентификатор this.

Примечание

Сценарий, использующий относительную адресацию, является более устойчивым и переносимым. Так, если внутри некоторой структуры клипов используется относительная адресация, то эта структура может быть безболезненно помещена в новый клип или перемещена в другой документ. При этом работа сценариев не будет нарушена (до тех пор, конечно, пока в этой структуре не будут изменены внутренние связи). Если же использовать абсолютную адресацию, то заключение всей структуры внутрь нового клипа приведет к утрате работоспособности сценариев из-за неверных ссылок.

Информация об абсолютной и относительной адресации переменных обобщена в табл. 4.3.

Таблица 4.3. *Виды адресации переменных*

Параметры	Виды адресации переменных	
	Абсолютная (absolute)	**Относительная (relative)**
Определение	Предполагает указание пути, начиная с самого верхнего (корневого) уровня	Предполагает указание пути относительно текущего уровня структуры
Особенности синтаксиса	Путь всегда начинается со свойства _root	Для обращения к текущему объекту используется идентификатор this. Для обращение к вышележащему уровню – идентификатор _parent
Особенности применения	Размещение структуры клипов внутри нового клипа требует изменения всех ссылок	Размещение структуры клипов внутри нового клипа не требует изменения ссылок
Примеры обращения к переменной	_root.mc1.mc2.mc3.myVar _root["mc1"]["mc2"].mc3.myVar	mc1.mc2.mc3.myVar this["mc1"]["mc2"].mc3.myVar _parent._parent.myVar1

Время жизни переменной

Время жизни переменной — это время, в течение которого переменная связана с определенной ячейкой памяти и хранимое в ней значение доступно для извлечения

и модификации. Время жизни переменной начинается в момент ее объявления и заканчивается, когда переменная удаляется из памяти. Рассмотрим понятие времени жизни для разных видов переменных.

- Локальные переменные имеют наиболее короткое время жизни. Время жизни локальной переменной совпадает с временем выполнения функции, в которой она объявлена. Локальная переменная создается при вызове функции и удаляется в момент завершения работы функции.
- Время жизни переменной монтажной линейки продолжается до тех пор, пока объект, содержащий эту монтажную линейку, присутствует в фильме. Так, переменная монтажной линейки клипа существует, начиная с момента объявления и до тех пор, пока содержащий ее клип не выгружается из проигрывателя или удаляется из фильма. Выгрузка клипа выполняется при загрузке в него внешнего SWF-документа, например, с помощью методов `MovieClip.loadMovie()` или `MovieClipLoader.loadClip()`. Клип также может быть выгружен из проигрывателя с помощью методов `MovieClip.unloadMovie()` или `MovieClipLoader.unloadClip()`. Удаление динамически созданного клипа может быть осуществлено с помощью метода `MovieClip.removeMovieClip()`. При выгрузке или удалении клипа все переменные, прикрепленные к его монтажной линейке, удаляются.

  Переменная основной монтажной линейки существует в фильме до тех пор, пока SWF-документ присутствует в проигрывателе. При выгрузке документа все переменные, объявленные на главной монтажной линейке и на линейках клипов этого документа, удаляются. Документ может быть выгружен из проигрывателя вследствие загрузки другого документа на его уровень либо в результате вызова метода `MovieClip.unloadMovie()` или `MovieClipLoader.unloadClip()`.
- Глобальная переменная имеет наиболее длительное время жизни, она существует, начиная с момента объявления и до завершения работы проигрывателя Flash Player. Даже при выгрузке фильма из проигрывателя глобальная переменная продолжает существовать (разумеется, если она была объявлена раньше, чем произошла выгрузка фильма).

Удаление переменной

Удаление переменной происходит при выгрузке или удалении фильма или клипа, содержащего объявление этой переменной. О загрузке и выгрузке клипов и документов см. *в главах 12* и *20*.

Также существует возможность удалить переменную без необходимости выгрузки или удаления содержащего ее объекта. Для этого используется оператор `delete`. Этот оператор имеет следующий синтаксис:

```
delete идентификатор
```

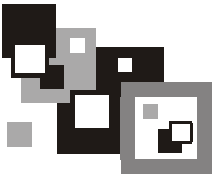
Здесь `идентификатор` — это имя переменной (или объекта). Оператор `delete` возвращает булево значение `true` в случае успешного удаления переменной и возвращает `false`, если удаление не может быть выполнено или невозможно. Удаление переменных с помощью оператора `delete` позволяет освободить память и вполне

целесообразно после того, как переменная исполнила свою роль и уже более не нужна.

В следующем примере осуществляется проверка пароля, введенного пользователем и сохраненного в переменной `userPass` (это не показано в примере). Для хранения правильного пароля используется переменная `pass`. Если пароль, введенный пользователем, совпадает с верным паролем, запускается анимация. Очевидно, в этом случае хранить верный пароль больше не нужно, и он может быть удален.

```
var pass:String = "correctPass"; // Сохраняем верный пароль
if (userPass == pass){ // Проверяем, совпадает ли пароль, введенный,
                       // пользователем, с верным паролем
       play();      // Если да, то запускаем анимацию и
       delete pass; // удаляем верный пароль, так как он больше не нужен
}
else stop(); /* Если не совпадает, останавливаем анимацию. Переменная
pass не удаляется */
```

Глава 5



Элементарные типы данных

Что для одного ошибки, для другого — исходные данные.

Следствие Бермана

Диапазон задач, с которыми приходится сталкиваться при разработке проектов, достаточно широк. Для того чтобы компьютерная программа могла обрабатывать информацию и выдавать конечный результат, необходимо, чтобы исходные данные были четко структурированы и представляли одну из категорий, определенных в применяемом языке программирования. Такая категория называется *типом данных*. Формально тип данных представляет множество значений и определяет набор допустимых операций, которые могут быть произведены над этими значениями. Если интерпретатор встречает определенное значение, он определяет его тип и "понимает", каким образом с этим значением нужно поступить. Если требуемая операция для указанного типа данных в языке не определена, то интерпретатор попытается либо преобразовать данные к нужному типу, либо не сможет выполнить требуемую операцию.

Тип данных может быть строго указан при объявлении переменной или он может быть автоматически определен из контекста. Интерпретатор "узнает", какой тип данных содержится в элементе по виду присваиваемого ему литерала или в результате строгого указания его типа. *Литерал* — это конкретное значение, заданное в программе буквально, в виде последовательности символов. Различные типы данных имеют разные правила оформления литералов. В данной главе речь пойдет об элементарных типах. Ссылочные типы будут рассмотрены далее в соответствующих главах.

Элементарные и ссылочные типы данных

В ActionScript данные разделены на две категории: *элементарные* (primitive) типы данных и *ссылочные* (reference) типы данных.

Элементарные типы данных — это типы данных, не определяемые в терминах других типов. Элементарные типы — это кирпичи, служащие для представления информации в программе, из которых строятся комплексные структуры данных, служащие для моделирования более сложных понятий.

Характерной особенностью элементарных типов данных является то, что такие данные непосредственно принадлежат содержащему их элементу (например, переменной). Так, если переменная содержит данные элементарного типа, то эти данные непосредственно сохраняются в ячейке памяти, связанной с переменной. Если в программе существует несколько разных переменных, содержащих одинаковые значения элементарного типа, то каждая из этих переменных получает свою личную копию данных. Таким образом, несмотря на то, что ряд переменных фактически представляет одинаковое значение, они никак не связаны друг с другом, поскольку каждая из них содержит свой собственный элемент данных.

ActionScript определяет следующие элементарные типы данных:

- `Number`, числовой;
- `String`, строковый;
- `Boolean`, булев или логический;
- `Undefined`, значение не определено;
- `Null`, отсутствие значения.

В отличие от элементарных данных, ссылочные данные не сохраняются непосредственно в каждой переменной. Ссылочные данные сохраняются в определенном участке памяти, а в переменную помещается только *ссылка* или указатель на эти данные. При этом говорят, что данные передаются по ссылке. Такое поведение позволяет избежать многократного копирования одних и тех же данных при их сохранении в разных переменных и за счет этого повышает эффективность работы программы. Поскольку переменные содержат только ссылку на данные, любое изменение данных автоматически приводит к изменению значений всех ссылающихся на них переменных. Это чрезвычайно важная особенность, о которой необходимо помнить!

К ссылочным типам данных относятся:

- `MovieClip`
- `Object`.

Здесь следует отметить, что любой объект, порожденный встроенным или пользовательским классом ActionScript, наследует от встроенного класса `Object`. В связи с этим любой объект, порожденный классом ActionScript, можно трактовать как объект `Object`. Таким образом, к ссылочным данным относятся такие элементы, как функции (класс `Function`), массивы (класс `Array`), кнопки (класс `Button`), экземпляры текстовых полей (класс `TextField`) и другие экземпляры встроенных (и пользовательских) классов. В связи с особой ролью и многочисленными характерными особенностями экземпляров символа типа Movie Clip они выделяются в особый тип `MovieClip`, хотя клипы, как и объекты, порожденные другими встроенными классами, наследуют от класса `Object`.

Примечание

ActionScript 2.0 допускает возможность объявления типов данных на основе встроенных или пользовательских классов. Это позволяет реализовать строгий контроль типа. Так,

например, при объявлении переменной, содержащей ссылку на массив, можно строго указать ее тип как `Array`: `var myArray:Array = new Array();` *(об определении пользовательских типов данных будет сказано в главе 25).*

Сведения о типах данных, поддерживаемых языком ActionScript, обобщены в табл. 5.1.

Таблица 5.1. *Типы данных ActionScript*

Тип данных	Назначение	Пример литерала
Элементарные (primitive) типы данных		
`Number` (числовой)	Числа, используются для счета и математических выражений	`534, 0001, 0xFF00CC`
`String` (строковый)	Последовательность произвольных символов (букв, цифр, знаков пунктуации и управляющих символов)	`"Lena", "67", "\r"`
`Boolean` (логический)	Представление или результат какого-либо условия или сравнения	Допустимы только два значения `true` (истина) и `false` (ложь)
`Undefined` (значение не определено)	Отсутствие данных	`undefined`
`Null` (нулевое)	Отсутствие данных	`null`
Ссылочные (reference) типы данных		
`MovieClip` (клип)	Для обработки клипов	Отсутствует

`Object` (объект) этот тип лежит в основе всех остальные встроенных и пользовательских объектов	Для обработки сложных структур данных	`{`
		`name:"Lena", gender: "female"`
		`}`

Числовой тип. Работа с числами

Числовой тип данных используется для хранения числовой информации и выполнения арифметических операций. С помощью числового типа данных осуществляется представление *целых чисел* (integer) и *чисел с плавающей точкой* (floating point numbers).

Оболочкой (wrapper object) числового типа данных является встроенный класс `Number`. С его помощью можно выполнять различные манипуляции с числами, а также получать доступ к некоторым числовым значениям, хранящимся в этом классе в качестве свойств.

Литералы чисел

Целые числа

Литералы целых чисел могут состоять из следующих символов:

- цифры от 0 до 9;
- знаки – или + перед числом, в отсутствие знака число считается положительным.

Числа 10, −8 являются целыми.

Кроме простых десятичных чисел, которые в основном используются на практике, можно также использовать восьмеричные и шестнадцатеричные числа.

Литерал восьмеричного числа начинается с цифры 0. В основании восьмеричной системы счисления лежит число 8. Максимальное значение для цифры каждого разряда составляет 7. Самый правый разряд представляет нулевую степень числа 8 (то есть 1), следующий за ним — первую степень (то есть 8), далее вторую (64) и т. д. Для того чтобы преобразовать число из восьмеричной системы в десятичную, нужно умножить каждую его цифру на соответствующую ее разряду степень числа 8 и сложить результаты. Вот несколько примеров:

```
01 // В десятичной системе 0*8^1 + 1*8^0 = 1
010 // В десятичной системе 0*8^2 + 1*8^1 + 0*8^0 = 8
0100 // В десятичной системе 0*8^3 + 1*8^2 + 0*8^1 + 0*8^0 = 64
0144 // В десятичной системе 0*8^3 + 1*8^2 + 4*8^1 + 4*8^0 = 100
```

Литерал шестнадцатеричного числа начинается с префикса `0x`. В основании шестнадцатеричной системы лежит число 16. Поскольку 10 цифр для отображения цифры разряда шестнадцатеричного числа оказывается недостаточно, используются шесть начальных букв английского алфавита от A до F. Максимальное значение для цифры каждого разряда составляет F. Для преобразования числа из шестнадцатеричной системы в десятичную используется рассмотренный выше принцип, только в данном случае вычисляются степени числа 16. Рассмотрим несколько примеров:

```
0x9 // В десятичной системе 9*16^0 = 9
0xA // В десятичной системе 10*16^0 = 10
0xF // В десятичной системе 15*16^0 = 15
0xFF // В десятичной системе 15*16^1 + 15*16^0 = 240 + 15 = 255
```

Числа с плавающей точкой

Эти числа используются для моделирования действительных чисел. Большинство действительных чисел могут быть представлены лишь путем аппроксимации, то есть с приближением. В Flash применяются числа с плавающей точкой двойной точности (double-precision), позволяющие представить число с точностью до 15 значащих цифр.

Литералы чисел с плавающей точкой могут состоять из следующих символов:

- десятичные цифры от 0 до 9;
- знаки – или + перед числом, в отсутствие знака число считается положительным;

- точка (.) в качестве разделителя целой и дробной части числа;
- порядок числа (десятичный логарифм).

Если целая часть дробного числа равна нулю, то этот ноль может быть опущен и литерал начинается с десятичной точки:

```
.5 // То же что и 0.5
-.123456 // То же что и -0.123456
```

Порядок обозначается с помощью буквы `E` или `e` и используется для представления больших или, наоборот, малых чисел. Для определения величины числа, использующего запись с порядком, нужно умножить это число на 10 в степени, заданной его порядком. Если порядок положительный, нужно передвинуть десятичную точку вправо на число разрядов, равное порядку. К целому числу добавляется соответствующее число нулей. Если порядок отрицательный, точка передвигается влево. Это демонстрируется ниже:

```
0.02E3 // 0.02 * 10^3 = 20
25.345E4 // 25 * 10^4 = 253450
16.123e-4 // 16.123 * 10^-4 = 0.0016123
```

Специальные числовые значения

Специальные числовые значения — это встроенные константы, которые могут являться результатом математических вычислений или присвоены переменным в качестве значения. Специальные числовые значения могут быть доступны как свойства встроенного класса `Number`. Некоторые из них также являются глобальными константами.

Минимальное и максимальное значения

ActionScript может работать с ограниченным диапазоном действительных чисел. Минимальное и максимальное значения хранятся в свойствах класса `Number` `Number.MIN_VALUE` и `Number.MAX_VALUE`. Следующий код выводит в окно **Output** наименьшее и наибольшее положительные числа, которые могут быть представлены в среде ActionScript.

```
trace (Number.MIN_VALUE); // Минимальное представляемое число 5e-324
trace (Number.MAX_VALUE); // Максимальное представляемое число 1.79E+308
```

Минимальное или максимальное значение при необходимости можно сохранить в переменной:

```
var max:Number = Number.MAX_VALUE;
```

Отрицательная и положительная бесконечность

Если результатом выполнения математических вычислений является значение, превышающее предельно допустимые значения, возвращается специальное значение

отрицательная бесконечность `-Infinity` (negative infinity) или положительная бесконечность `Infinity` (positive infinity).

Для того чтобы указать данные значения, можно воспользоваться либо свойствами класса `Number Number.NEGATIVE_INFINITY` и `Number.POSITIVE_INFINITY`, либо соответствующими глобальными константами `-Infinity` и `Infinity`.

Рассмотрим следующий пример:

```
var x = 5;
var y = 0;
trace (x/y); // Выводит Infinity
trace (-x/y); // Выводит -Infinity
```

При необходимости сохранения данного значения можно использовать свойство класса `Number` или константу.

```
var posInf = Number.POSITIVE_INFINITY;
var negInf = -Infinity
```

В ActionScript существует специальная глобальная функция, позволяющая проверить, является ли результат вычисления выражения положительной или отрицательной бесконечностью. Эта функция называется `isFinite()` и имеет следующий синтаксис:

```
isFinite(выражение)
```

Параметр *выражение* — это выражение, для результата вычисления которого выполняется проверка на конечность. Если результат вычисления выражения конечен, то функция возвращает булево значение `true`, если результат выходит за пределы, ограниченные минимальным и максимальным значениями, то функция возвращает значение `false`.

Значение *NaN*

Значение `NaN` (Not a Number) используется для представления данных числового типа, не являющихся числом. Значение `NaN` может быть получено в результате выполнения арифметической операции или в результате преобразования типа данных.

```
var x = 0;
var y = 0;
trace (x/y); // Выводит NaN
var num: Number = 5;
var str:String = "Flash";
trace (num - str); // Выводит NaN
```

Для использования этого значения в программе можно прибегнуть к статическому свойству класса `Number Number.NaN` или применить предопределенную переменную

NaN. Следует отметить, что хотя NaN и указывает на то, что значение не является числом, однако относится к числовому типу:

```
trace (typeof NaN); // Выводит number
var num:Number = NaN; /* Ошибки несоответствия типа нет, так как NaN
имеет числовой тип */
```

При сравнении двух переменных, имеющих значение NaN, результатом проверки является значение false. Для того чтобы проверить, не принимает ли переменная значение NaN, необходимо воспользоваться глобальной функцией isNaN(). Функция имеет следующий синтаксис:

```
isNaN (выражение)
```

Функция isNaN() возвращает значение true, если выражение *выражение* дает значение NaN, и возвращает значение false в противном случае. В приведенном далее примере выполняется проверка того, что числовая переменная является числом.

```
if (isNaN(myVar)){
trace ("Не число");
       }
else {
       trace ("Число");
}
```

Примечание

Свойства встроенного класса Math позволяют получить доступ к целому ряду математических констант, таких как число π, число E (число Эйлера), натуральный логарифм числа 10 и т. д. Об этих свойствах, а также о методах класса Math, позволяющих выполнять различные математические вычисления, см. в *главе 15*.

Операции над числами

С данными числового типа можно выполнять четыре основных арифметических операции (сложение, вычитание, умножение и деление), а также находить остаток от деления одного числа на другое. Для проведения вычислений формируются выражения, состоящие из операндов, то есть элементов, над которыми производится операция, и операторов, которые определяют действие, производимое с операндами. Числовые данные также можно сравнивать друг с другом с помощью операторов отношений. Выражение отношений, как правило, состоит из двух сравниваемых операндов и оператора отношений. Выражение отношений всегда возвращает булево значение true (истина) — если отношение верно или false (ложь) — если отношение неверно.

В данном разделе будут рассмотрены арифметические операторы и операторы отношений, применение которых характерно для числовых данных. Перечень таких операторов представлен в табл. 5.2.

Таблица 5.2. *Операторы для работы с числами*

Оператор	Операция
Арифметические операторы	
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Вычисление остатка от деления
++	Инкрементирование
--	Декрементирование
Операторы отношений	
==	Равно
===	Строгое равенство
!=	Не равно
!==	Строгое неравенство
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Применение оператора `%` позволяет вычислить остаток от деления, как это показано в следующем примере:

```
var x = 15;
var y = 7;
trace (x%y); // 15-(2*7)=1. Выводит 1
```

Следующий фрагмент кода позволяет выяснить, является ли значение переменной `myVar` четным числом.

```
if (myVar % 2 == 0) {
        trace("Четное число");
}
else {
        trace("Нечетное число");
}
```

Следующее выражение позволяет исключить из значения угла, хранящегося в переменной thi, целое число оборотов:

```
thi = thi % 360;
```

Операторы инкрементирования и декрементирования будут подробно рассмотрены в *главе 6*.

Операторы отношений позволяют выяснить, какой из двух принимаемых ими операндов имеет большее/меньшее значение, или установить их равенство.

Оператор равенства (==) используется для проверки того, являются ли два значения равными. Если отношение выполняется, выражение разрешается в true, если значения не равны, то результатом выражение является значение false. Важно отметить, что оператор равенства проверяет только равенство значений, но не идентичность типов данных. Если сравниваются данные, не являющиеся строками или имеющие разный тип, то интерпретатор преобразовывает их тип в числовой. Для проверки равенства значений и типов данных используется оператор строгого равенства (===). Более подробно эти вопросы будут рассмотрены в *разд. "Автоматическое преобразование типов данных" данной главы*.

```
trace(2==3); // Выводит false
var x:Number = 10;
var y:Number = 50/5;
trace (x==y); // Выводит true
```

Следует обратить внимание на то, что оператор равенства состоит из двух знаков =. Это чрезвычайно важный факт, который нужно иметь в виду, поскольку, если вместо оператора == использовать оператор =, левый операнд будет интерпретирован как идентификатор, а правый — как присваиваемое ему значение. Поэтому вместо проверки равенства выполнится присваивание значения, расположенного справа от оператора присваивания, идентификатору, расположенному слева! Конечно же, это приведет к неожиданным результатам или даже к ошибке. Такая ситуация представлена в следующем примере:

```
var x:Number = 10;
var y:Number = 25;
if(x=y){ // Неверный оператор, нужно писать x==y
trace ("x равно y");
       }
trace (x); //Выводит 25
```

Здесь вместо проверки равенства переменных x и y переменной x присваивается значение переменной y. В результате блок условного предложения выполняется вне зависимости от того, выполняется отношение или нет, так как условное выражение разрешается в true в результате автоматического преобразования типов.

Оператор неравенства (!=) возвращает булево значение true, если значения его операндов не равны, и значение false в противном случае.

Это демонстрируется в следующем примере:

```
var x:Number = 10;
var y:Number = 25;
trace (x != y); //Выводит true
trace ( x != (25-15)); //Выводит false
```

Если операнды не являются строками, они преобразуются в числовой тип.

Оператор строгого неравенства (!==) не выполняет преобразование типа, а в остальном действует аналогично оператору (!=).

Операторы отношений в основном применяются для составления проверочных выражений в условных предложениях и в предложениях цикла для определения требуемого числа итераций.

Встроенные методы для работы с числами

Класс Number включает только два метода toString() и valueOf(). Для вызова этих методов необходимо использовать объект Number или переменную, содержащую числовое значение.

Для создания объекта используется оператор new и функция конструктора Number(). Создаваемый объект, как правило, сохраняется в переменной. При этом используется следующий синтаксис:

идентификатор:Number = new Number(*число*)

Здесь *идентификатор* — имя переменной, содержащей объект Number, *число* — числовое значение объекта Number.

Как правило, используется более простой способ, когда переменной просто присваивается числовое значение. После этого, используя идентификатор этой переменной, можно получить доступ к любому свойству и методу класса Number.

Метод *Number.toString()*

Метод toString() возвращает строковое значение, содержащее представление числа в заданной системе счисления. Данный метод имеет следующий синтаксис:

число.toString(*основание*)

Здесь *число* — идентификатор объекта Number или переменной, содержащей числовое значение. Если объект или переменная *число* имеет значение undefined, то метод toString() возвращает значение NaN[1]. Следует иметь в виду, что данный метод не возвращает двоичное представление отрицательного числа. Вместо этого выводится строка, содержащая двоичное представление положительного числа со знаком минус.

Параметр *основание* задает основание системы счисления, в которой будет возвращено строковое представление исходного числа. Этот параметр может принимать

[1] При публикации для Flash Player 6 возвращается значение 0. — *Прим. автора*.

значения в диапазоне от 2 до 36 и является необязательным. Если он опущен, используется основание 10. Ниже представлено несколько примеров использования метода `toString()`.

```
var n:Number= new Number(1103);// Создаем объект Number со значением 1103
n.toString(16);/* Возвращает строку "44f" - представление числа 1103 в
шестнадцатеричной системе счисления */
var num:Number = 1000;
num.toString(2); /* Возвращает строку "1111101000" - представление числа
1000 в двоичной системе счисления */
num = -6
trace(x.toString(2)); /* Выводит -110, а не
11111111111111111111111111111010 */
```

Этот метод может быть очень полезен для изменения системы счисления при получении символа строки на основе значения его кода или при задании значения цвета объекта.

Метод *Number.valueOf()*

Метод `valueOf()` не принимает параметров и возвращает числовое значение объекта `Number`.

Данный метод имеет следующий синтаксис:

```
число. valueOf()
```

Здесь *число* — идентификатор объекта `Number` или переменной, содержащей числовое значение.

```
var n:Number= new Number(1103);
n.valueOf();// Возвращает число 1103
var num:Number = 1000;
num.valueOf(); // Возвращает число 1000
```

Строковый тип. Работа со строками

Строка (string) представляет собой произвольную последовательность символов, включающую буквы, цифры, знаки пунктуации, а также непечатные (управляющие) символы, как, например, символ переноса строки. Данные строкового типа часто называют строками. Строковый тип применяется для представления и обработки текстовых данных для ввода и вывода.

Литералы строк

Строковый литерал оформляется с помощью кавычек. Кавычки могут быть двойными или одинарными (апострофы). Любая последовательность символов, заключенная в кавычки, в тексте программы интерпретируется как строка и при использова-

нии режима подсветки синтаксиса (Syntax coloring) выделяется установленным цветом (по умолчанию голубым).

Правила оформления

Важно, чтобы при оформлении строкового литерала кавычки находились в соответствии. Если открывающие кавычки двойные, то и закрывающие также должны быть двойными. Если в качестве открывающих кавычек используется апостроф, то он должен и закрывать литерал.

```
"Это строка"
'Это тоже строка'
```

Внутри строки нельзя использовать тип кавычек, применяемых для записи литерала. Так, если для оформления строкового литерала применяются двойные кавычки, то внутри строки можно использовать только одинарные и наоборот.

```
'To replay the movie press the "S" key' //Допустимая строка
"Rock'n'Roll" //Допустимая строка
'Welcome to the Albert's home' // Некорректная строка
```

Кодировка

Символьные данные хранятся в памяти компьютера с помощью цифрового *кодирования* (encoding). Каждому символу соответствует цифровой код, состоящий из определенного количества бит. Чем больше бит в кодировке можно использовать для кодирования символа, тем больший диапазон символов может быть представлен с помощью данной кодировки. Так, наиболее распространенная система кодирования ASCII (American Standart Code for Information Interchange) — Американский стандартный код обмена информацией — использует для кодирования символов 7 бит и позволяет закодировать 128 символов, включающих латинский алфавит, числа, знаки пунктуации и основные управляющие символы.

С выходом седьмой версии Flash в ActionScript включена поддержка кодировки Unicode, позволяющей охватить символы большинства мировых языков, включая знаки пунктуации, математические, технические символы и т. д. Это означает, что файл ActionScript может содержать буквенные символы разных языков (в том числе кириллицу). В основном в Flash применяется особый формат кодировки Unicode UTF-8, использующей для кодирования символов последовательности байтов различной длины, которые преобразуются в коды Unicode. Первые 128 кодовых позиций в кодировке UTF-8 совпадают с символами кодировки ASCII, причем для кодирования этого набора символов используется такое же, как и в кодировке ASCII, количество бит. Для кодирования кириллических символов требуется уже два байта (то есть 16 бит).

Escape-последовательности

Кодировка Unicode позволяет использовать множество символов, которые нельзя ввести непосредственно с клавиатуры. К таким символам, например, относится знак copyright (©). Для того чтобы использовать такие символы, можно прибегнуть к

применению Escape-последовательностей. *Escape-последовательность* (Escape sequence) представляет собой комбинацию символов, используемую для задания неотображаемых символов и символов, имеющих специальное значение.

Escape-последовательность может быть задана с помощью символа обратной косой черты, после которой следует некоторый символ либо его код. В табл. 5.3 представлен перечень Escape-последовательностей, используемых в ActionScript, и соответствующие им ASCII-коды и коды Unicode.

Таблица 5.3. *Escape-последовательности в Unicode*

Escape-последовательность	Функция	Код ASCII	Код Unicode
\b	Забой (backspace)	8	\u0008
\t	Горизонтальная табуляция (tab character)	9	\u0009
\n	Перевод строки (line feed)	10	\u000a
\f	Перевод страницы (form feed)	12	\u000c
\r	Возврат каретки (carriage return character)	13	\u000d
\"	Кавычки (double quotation mark)	34	\u0022
\'	Апостроф (single quotation mark)	39	\u0027
\\	Обратная косая черта (backslash)	92	\u005c
\000 - \377	Восьмеричный код символа		
\x00 - \xFF	Шестнадцатеричный код символа (сокращенная форма)		
\u0000 - \uFFFF	Шестнадцатеричный код Unicode		

При записи Escape-последовательности после обратной косой черты можно указать восьмеричный или шестнадцатеричный код требуемого символа. При использовании восьмеричного кода можно получить любой из первых 256 символов кодировки Unicode, этот набор известен как Latin1. Вот несколько примеров:

```
trace ("\100"); // Выводит @
trace ("\251"); // Выводит ©
```

Для получения любого символа из набора Latin1 также можно использовать сокращенную форму шестнадцатеричной записи. Для этого после обратной наклонной черты указывается префикс x, после которого следует двузначное шестнадцатеричное число. Предыдущие символы могут быть получены с помощью следующих Escape-последовательностей:

```
trace ("\x40"); // Выводит @
trace ("\xa9"); // Выводит ©
```

Для записи Escape-последовательности, позволяющей получить любой символ кодировки Unicode, после обратной косой черты используется префикс `u`, после которого указывается четырехзначный шестнадцатеричный код символа.

```
trace ("\u0040"); // Выводит @
trace ("\u00a9"); // Выводит ©
trace ("\u0410"); // Выводит А (Кириллица)
trace ("\u044F"); // Выводит я
```

Escape-последовательность часто приходится использовать не только для представления символов, отсутствующих на клавиатуре, но также для использования обоих типов кавычек внутри строки. Если строковый литерал использует двойные кавычки и внутри строки также требуется использовать двойные кавычки, необходимо прибегнуть к использованию Escape-последовательности, как это показано ниже.

```
"Нажмите клавишу \"S\" для остановки фильма"
'Нажмите клавишу \'S\' для остановки фильма'
```

При необходимости переноса строки можно воспользоваться Escape-последовательностью `"\n"`.

```
trace ("Одна строка\nДругая строка"); // Выводит текст в две строки
```

Операции над строками

Со строковыми данными можно выполнять два типа операций: конкатенация и сравнение. Кроме того, встроенный класс `Stirng` предоставляет целый ряд методов, позволяющих выполнять разнообразные действия со строками. Обобщенная информация о возможностях работы со строками представлена в табл. 5.4, расположенной в конце данного раздела.

Конкатенация

Конкатенация (concatenation) — это процедура, формирующая строку из нескольких строк путем их соединения ("склеивания"). Для выполнения конкатенации используется оператор сложения (+). При сложении строк они объединяются в новую строку.

```
trace("Привет" + "Flash"); //Выводит ПриветFlash
```

Для того чтобы использовать пробелы между словами в результирующей строке, можно включить их в состав строк, которые должны быть соединены, либо сложить эти строки со строками, состоящими только из пробелов, — пустыми строками. Литерал пустой строки состоит только из открывающих и закрывающих кавычек (`""`) или апострофов (`''`).

```
trace("Привет " + "Flash"); //Выводит Привет Flash
trace("Привет" + "" + "Flash"); //Выводит Привет Flash
```

С помощью операции конкатенации можно формировать выражения, включающие литералы и идентификаторы, которые при вычислении выдают единичное строковое значение.

```
var userName:String = "Dmitiri I. Albert";
var email:String = "book_flash@mail.ru";
var url:String = "mailto:" + userName + "<" + email + ">";
trace(url); // Выводит mailto:Dmitiri I. Albert<book_flash@mail.ru>
trace(typeof(url)); // Выводит string
```

В приведенном примере переменной `url` в качестве значения присваивается результат вычисления выражения, содержащего строковые литералы и идентификаторы переменных строкового типа. В результате это выражение выдает единую строку, которая в дальнейшем может быть передана какой-либо функции в качестве параметра (например, функции `getURL()`).

Следует помнить, что при сложении строкового и числового значения в результате автоматического преобразования типов данных число будет преобразовано в строку, после чего будет выполнена конкатенация. При этом тип исходных переменных не изменится, преобразование выполняется только с копией данных, хранимых в переменной:

```
var sVar:String = "PI=";
var nVar:Number = 3.14;
var mes = sVar+nVar;
trace(mes); //Выводит PI=3.14
trace(typeof(mes)); // Выводит string
trace(typeof(nVar));// Выводит number, переменная не претерпела изменений
```

Сравнение строк

Для сравнения строк применяются те же операторы, что и для сравнения чисел: больше (>), меньше (<), больше или равно (>=), меньше или равно (<=), равно (==) и не равно (!=). Критерием сравнения строк является значение кода символов, составляющих строки. Большим считается символ, кодовое значение которого больше. При сравнении строк, состоящих из нескольких символов, происходит попарное сравнение их кодовых значений слева направо. Если строки начинаются с одинаковых символов, сравнение происходит до тех пор, пока не будут обнаружены разные символы. При обнаружении разных символов сравниваются их кодовые значения, и на основе этого принимается решение об отношении строк. Если сравниваемые строки содержат разное количество символов, то после исчерпания символов в одной из сравниваемых строк вместо очередного символа используется *пустая строка*. Пустая строка — это самый первый символ в кодировке Unicode (ее код `\u0000`), поэтому любой символ за исключением пустой строки при сравнении с ней оказывается больше. Следует иметь в виду, что в кодировке Unicode цифры предшествуют заглавным буквам латинского алфавита, за которыми, в свою очередь, следуют

строчные, а коды кириллических символов имеют более высокие значения, чем латинских. Это может быть показано с помощью функции `trace()`. Все приведенные ниже выражения выдают значение `true`, то есть их отношения верны.

```
trace("0"<"9");
trace("9"<"A");
trace("A"<"Z");
trace("Z"<"a");
trace("a"<"z");
trace("z"<"A");
trace("A"<"Я");
trace("Я"<"a");
trace("a"<"я");
```

Следующие примеры иллюстрируют основные закономерности сравнения строк:

```
trace ("FLASH"<"flash"); /* Выводит true, так как коды прописных букв
меньше, чем коды строчных */
trace ("Flash"<"flash"); // Выводит true, т.к код "F" меньше кода "f"
trace ("Flash 6"<"Flash MX"); //Выводит true, т.к код "6" меньше кода "M"
trace ("FlashMX">"Flash"); // Выводит true, т.к код "M" больше кода ""
```

Встроенные свойства и методы для работы со строками

Для выполнения различных манипуляций над строками в ActionScript используется встроенный класс `String`, являющийся оболочкой для строкового типа данных. Характерной особенностью данного класса является то, что для обращения к его свойствам и методам не требуется создавать экземпляр класса с помощью функции конструктора. Методы и свойства класса `String` доступны непосредственно для строкового литерала. При вызове свойства или методов класса `String` с использованием литерала интерпретатор автоматически преобразует его в объект типа `String`, вызывает требуемый метод (или свойство) и затем удаляет созданный объект. Синтаксис вызова метода класса `String` с помощью строкового литерала имеет следующий вид:

```
строка.метод();
```

Здесь *строка* — это либо строковый литерал, либо переменная, содержащая строковое (или для некоторых методов иное) значение. Важно помнить, что строковый литерал всегда содержит кавычки. В следующем предложении вызывается метод `String.toUpperCase()`, делающий все буквы исходной строки прописными.

```
"hello".toUpperCase(); // Возвращает HELLO
```

В приведенном примере метод `toUpperCase()` вызывается с помощью строкового литерала; аналогично данный метод может быть вызван с помощью идентификатора переменной, хранящей строковое значение:

```
var greeting:String = "hello";
greeting.toUpperCase(); // Возвращает HELLO
```

При необходимости (в более редких случаях) можно явно создать объект `String` с помощью оператора `new` и функции конструктора. В этом случае используется следующий синтаксис:

```
new String (строка);
```

Здесь *строка* — это строковый литерал или строковая переменная.

При создании объекта с помощью оператора `new` и функции конструктора его обычно сохраняют в переменной. В дальнейшем доступ к свойствам и методам объекта `String` можно получить с помощью этой переменной. Ниже приводится пример, иллюстрирующий весь этот процесс.

```
var greeting:String = new String("hello"); /* Создаем новый объект String
со значением "hello" и помещаем его в переменную hello */
trace(greeting.toUpperCase()); // Выводит HELLO
```

Примечание

Все методы класса `String`, за исключением методов `String.concat()`, `String.fromCharCode()`, `String.slice()` и `String.substr()`, — это методы общего назначения, то есть их можно вызывать не только с помощью объектов String, но и посредством других объектов. Это связано с тем, что перед выполнением своих действий такие методы сами преобразуют вызвавший их объект в строку посредством метода `Object.toString()`.

Все символы, входящие в состав строки, проиндексированы, то есть каждый символ, расположенный в строке в определенной позиции, имеет свой собственный порядковый номер — индекс. Нумерация символов в строке начинается с нуля. Это обстоятельство необходимо иметь в виду, как и то, что индекс последнего символа в строке всегда на единицу меньше длины строки, то есть количества входящих в нее символов.

Свойство *String.length*

Это единственное свойство, определенное в классе `String`. Свойство `length` указывает количество символов в строке. Для обращения к нему можно использовать следующий синтаксис:

```
строка.length
```

где *строка* — это строковый литерал или переменная, содержащая строковое значение. Ниже приводятся примеры использования этого свойства.

```
//Использование литерала
var l:Number = "ActionScript".length;
```

```
trace(l); // Выводит 12
//Использование строковой переменной
var login:String = "Dmitri";
trace(login.length); // Выводит 6
//Использование объекта String
var myStr:String = new String("Dmitri");
trace (myStr.length); // Выводит 6
```

Метод *String.concat()*

Этот метод осуществляет операцию конкатенации со строками, переданными ему в качестве параметров, и возвращает результирующую строку. Значение исходного объекта `String` при этом не меняется. Зачастую вместо применения метода `concat()` можно использовать оператор сложения или сложение с присваиванием, о котором будет сказано в *главе 6*.

Метод `concat()` имеет следующий синтаксис:

строка.concat(*строка1, строка2, ..., строкаN*)

Здесь *строка* — строка, в которой осуществляется поиск, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Параметры *строка1, строка2, ..., строкаN* — это строковые литералы или выражения, дающие строковые значения, которые будут соединены.

Ниже демонстрируется применение данного метода.

```
trace("Cogitum".concat(" ergo", " sum")); // Выводит Cogitum ergo sum
var str:String = new String("Flash")
trace(str.concat(" MX", " 2004", " Professional")); /* Выводит Flash MX
2004 Professional */
trace(str); // Выводит Flash. Значение str не изменилось
```

Метод *String.charAt()*

Метод `charAt()` возвращает символ строки на основе его индекса, переданного в качестве параметра. Этот метод имеет следующий синтаксис:

строка.charAt(*индекс*)

Здесь *строка* — это строковый литерал или переменная, содержащая строковое значение, а *индекс* — число в диапазоне от `0` до `length-1` или выражение, дающее число, лежащее в указанном диапазоне. Возвращаемое значение имеет строковый тип.

Если параметр выходит за пределы допустимого диапазона (то есть задается индекс несуществующего в строке символа), метод возвращает пустую строку (""). Если *индекс* задается с помощью выражения и оно возвращает значение `undefined`, это значение преобразуется в 0, в результате чего метод возвращает первый символ строки.

```
var language:String = "ActionScript";
```

```
trace(language.charAt(6)); /* Выводит S - седьмой символ строки
с индексом 6 */
trace(language.charAt(myVar)); /* Выводит A, так как переменная myVar не
определена и значение undefined преобразуется в 0 */
```

Метод *String.indexOf()*

Этот метод применяется для поиска символа или последовательности символов в строке. Метод `indexOf()` осуществляет в строке поиск заданного символа или последовательности символов, начиная с указанной позиции, и возвращает индекс первого вхождения в строку искомого фрагмента. Метод `indexOf()` имеет следующий синтаксис:

строка`.indexOf(`*подстрока*`, `*начальный индекс*`)`

Здесь *строка* — строка, в которой осуществляется поиск, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Параметр *подстрока* — набор искомых символов (или один символ), указанный как строковый литерал или выражение, дающее строку.

Параметр *начальный индекс* — задает индекс символа, с которого будет начат поиск. Он может быть задан как число или выражение, дающее числовое значение. Все символы, расположенные перед символом с этим индексом, при поиске не учитываются. Параметр *начальный индекс* является необязательным и может быть опущен. В этом случае его значение устанавливается равным 0, то есть поиск осуществляется с самого начала строки.

Метод `indexOf()` возвращает числовое значение, указывающее индекс символа, соответствующего месту первого вхождения искомого фрагмента. Если фрагмент не найден, возвращается значение −1.

```
"ActionScript".indexOf("Script"); // Возвращает 6 - индекс символа S
var searchString:String = "ActionScript"; // Строка для поиска
var searchChar:String = "t"; // Искомый символ
trace(searchString.indexOf(searchChar, 3)); /* Выводит 11 - индекс
первого вхождения символа "t", начиная с символа "i" с индексом 3 */
trace(searchString.indexOf("Script", 7)); /*Выводит -1, так как после
символа с индексом 7 искомый фрагмент не встречается в строке */
```

В *разд. "Рекурсивные функции" главы 8* приводится функция, позволяющая обнаружить индексы всех вхождений заданного набора символов в строку.

Метод *String.lastIndexOf()*

Этот метод, как и метод `indexOf()`, используется для поиска произвольного набора символов в строке, однако в отличие от предыдущего метода осуществляет поиск в строке справа налево. Синтаксис метода `lastIndexOf()` идентичен синтаксису метода `indexOf()`.

строка`.indexOf(`*подстрока*`, `*начальный индекс*`)`

Здесь *строка* — строка, в которой осуществляется поиск, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Параметр *подстрока* — набор искомых символов (или один символ), указанный как строковый литерал или выражение, дающее строку.

Параметр *начальный индекс* — задает индекс символа, с которого будет начат поиск. Здесь следует иметь в виду, что символы в строке индексируются слева направо, а метод `lastIndexOf()` осуществляет поиск в строке справа налево. В связи с этим параметр *начальный индекс* задает индекс последнего символа последнего вхождения подстроки (при чтении слева направо). Он может быть задан как неотрицательное число или выражение, дающее неотрицательное числовое значение. Параметр *начальный индекс* является необязательным и может быть опущен. В этом случае его значение устанавливается равным `length-1`, то есть поиск осуществляется с конца строки.

Как и предыдущий метод, метод `lastIndexOf()` возвращает числовое значение. Если требуемая подстрока не найдена — возвращается значение −1.

```
"ActionScript".lastIndexOf("i"); /* Возвращает 9 - индекс последнего
вхождения символа "i" */

var searchString:String = "ActionScript";

var searchChar:String = "t";

trace(searchString.lastIndexOf(searchChar, 10)); /* Выводит 2 - индекс
первого (справа налево) вхождения символа "t", начиная с символа "p" с
индексом 10 */
```

Метод *String.substr()*

Этот метод вырезает из строки заданное количество символов, начиная с указанной позиции, и возвращает вырезанную таким образом подстроку. При выполнении данной процедуры исходная строка не изменяется, вместо этого генерируется новая строка, состоящая из вырезанных символов. Синтаксис метода `substr()` представлен ниже.

строка.substr(*начальный индекс*, *длина подстроки*)

Здесь *строка* — строка, из которой извлекаются символы, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Параметр *начальный индекс* — задает индекс первого символа, входящего в извлекаемую подстроку. Этот параметр может быть задан литералом числа или выражением, дающим числовое значение. Если значение параметра *начальный индекс* отрицательно, то позиция, с которой выполняется извлечение подстроки, отсчитывается с конца строки. При этом индекс последнего (слева направо) символа в строке записывается как −1. Следует отметить, что даже если *начальный индекс* имеет отрицательное значение, извлечение подстроки осуществляется слева направо.

Параметр *длина подстроки* — задает количество извлекаемых из строки символов. Этот параметр может быть задан литералом неотрицательного числа или выражением, дающим неотрицательное числовое значение. Этот параметр является необязательным. Если он опущен, то извлекаемая подстрока включает в себя все символы,

начиная с символа, заданного параметром *начальный индекс*, и заканчивая последним символом исходной строки.

Возвращаемым значением данного метода является извлеченная подстрока. Рассмотрим несколько примеров:

```
"ActionScript".substr(6); /* Возвращает строку Script. Подстрока
извлекается, начиная с 7-го символа и до конца */
"ActionScript".substr(-12, 6); /* Возвращает строку Action. Подстрока
извлекается, начиная с 12-го с конца символа ("A") и включает 5 следующих
за ним символов */
var language:String = "ActionScript";
// Конкатенация двух извлекаемых из значения переменной language подстрок
var shortLang:String = language.substr(0, 1)+language.substr(-6, 1);
trace (shortLang); // Выводит AS
```

Метод *String.substring()*

Как и предыдущий, этот метод позволяет вырезать из строки последовательность символов. Однако в случае использования метода `substring()` для определения извлекаемой подстроки задаются начальный и конечный индексы символов. Этот метод использует следующий синтаксис:

строка.substring(*начальный индекс*, *конечный индекс*)

Здесь *строка* — строка, из которой извлекаются символы, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Параметр *начальный индекс* — задает индекс первого символа, входящего в извлекаемую подстроку. Этот параметр может быть задан литералом неотрицательного числа или выражением, дающим неотрицательное числовое значение. Допустимые значения параметра *начальный индекс* лежат в диапазоне от `0` до `lehgth-1`. Если значение отрицательно, то *начальный индекс* устанавливается равным нулю.

Параметр *конечный индекс* — задает индекс символа, следующего за последним символом извлекаемой подстроки. Символ с индексом *конечный индекс* не входит в извлекаемую подстроку. Этот параметр может быть задан литералом неотрицательного числа или выражением, дающим неотрицательное числовое значение. Допустимые значения параметра *конечный индекс* лежат в диапазоне от `0` до `lehgth`. Если значение отрицательно, то *конечный индекс* устанавливается равным нулю. Этот параметр является необязательным. Если он опущен, то *конечный индекс* устанавливается равным `length`, то есть извлекаемая строка включает все символы, начиная с символа с индексом *начальный индекс* и до конца строки.

Если параметры *начальный индекс* и *конечный индекс* равны, то метод возвращает пустую строку (""). Если значение параметра *начальный индекс* больше, чем значение параметра *конечный индекс*, то они автоматически меняются местами, после чего происходит извлечение подстроки. Применение данного метода иллюстрируется следующими примерами:

```
"ActionScript".substring(6); /* Возвращает строку Script. Подстрока
извлекается, начиная с 7-го символа и до конца */
```

```
"ActionScript".substring(0, 6); /* Возвращает строку Action. Подстрока
извлекается, начиная с 0-го символа ("A") и помимо него включает пять
следующих символов. Символ с индексом 7 ("S") не входит в подстроку */

"ActionScript".substring(6, 0); /* Возвращает строку Action.Параметры
меняются местами */
```

Метод *String.slice()*

По своему назначению метод `slice()` полностью аналогичен методу `substring()`. Различие заключается в том, что параметры метода `slice()` могут задавать индексы символов не только относительно начала, но и относительно конца строки. Метод `slice()` имеет следующий синтаксис:

```
строка.slice(начальный индекс, конечный индекс)
```

Здесь *строка* — это, как обычно, строка, из которой извлекаются символы, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Параметр *начальный индекс* — задает индекс первого символа, входящего в извлекаемую подстроку. Он может быть задан числовым литералом или выражением, возвращающим число. В отличие от соответствующего параметра метода `substring()`, параметр *начальный индекс* может принимать отрицательные значения. В этом случае отсчет позиции символа в строке ведется относительно позиции ее последнего символа, имеющего индекс −1.

Параметр *конечный индекс* — задает индекс символа, следующего за последним символом извлекаемой подстроки. Символ с индексом *конечный индекс* не входит в извлекаемую подстроку. Этот параметр может быть задан литералом числа или выражением, дающим числовое значение. Если значение параметра *конечный индекс* отрицательно, отсчет позиции символа в строке ведется относительно позиции ее последнего символа, имеющего индекс −1. Данный параметр является необязательным. Если *конечный индекс* не указан, то он устанавливается равным `length`.

Если символ, заданный параметром *начальный индекс*, расположен в строке после символа, заданного параметром *конечный индекс*, или параметры равны, то метод возвращает пустую строку (""). Рассмотрим несколько примеров:

```
"ActionScript".slice(6); /* Возвращает строку Script. Подстрока
извлекается, начиная с 7-го символа и до конца */

"ActionScript".slice(0, 6); // Возвращает строку Action

"ActionScript".slice(-6); /* Возвращает строку Script. Подстрока
извлекается, начиная с 6-го с конца символа и до конца */

"ActionScript".slice(-12, 3); /*Возвращает строку Act. Подстрока
извлекается, начиная с 12-го символа с конца и до 2-го символа с начала
включительно */

"ActionScript".slice(-3, -12); /* Возвращает пустую строку, так как
символ с индексом -3 находится после символа с индексом -12 */
```

Метод *String.split()*

Метод `split()` позволяет разбить строку на ряд подстрок и помещает каждую из них в массив. Строка разбивается на подстроки в местах обнаружения разделителя, представляющего собой символ или набор символов, который передается методу `split()` в качестве параметра. Сам разделитель не входит в состав подстрок. Получившиеся в результате разбиения подстроки помещаются в массив, который возвращается после выполнения метода. Метод `split()` позволяет ограничить количество элементов в результирующем массиве. Метод `split()` имеет следующий синтаксис:

строка`.split(`*разделитель*`, `*количество элементов*`)`

Здесь *строка* — это строка, из которой извлекаются символы, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Параметр *разделитель* — задает последовательность символов, разделяющую извлекаемые из исходной строки подстроки. В массив в качестве элементов помещаются подстроки, расположенные перед, между и после набора символов, заданного параметром *разделитель*. Этот параметр может быть задан строковым литералом или выражением, дающим строковое значение. Если параметр *разделитель* имеет значение пустой строки (""), то исходная строка разбивается на отдельные символы, и каждый из них помещается в массив. Если параметр *разделитель* имеет значение `undefined`, то исходная строка целиком помещается в массив в качестве единственного элемента.

Параметр *количество элементов* — задает число элементов возвращаемого массива. Этот параметр задается числовым литералом или выражением, возвращающим число. Если значение параметра *количество элементов* меньше, чем количество подстрок, полученных в результате разбиения исходной строки, то в массив помещается только соответствующее количество начальных подстрок. Этот параметр является необязательным, если он опускается или принимает значение `undefined`, то количество элементов в возвращаемом массиве совпадает с количеством извлекаемых подстрок. Если значение параметра *количество элементов* является дробным, оно округляется в сторону меньшего целого.

Ниже представлено несколько примеров, демонстрирующих возможности данного метода. Следует отметить, что при передаче массива в качестве аргумента функции `trace()`, его элементы выводятся в панель **Output** через запятую. В данных примерах метод `split()` возвращает массив и функция `trace()` выводит в панель **Output** его элементы.

```
trace("ActionScript".split("c")); /* Выводит A,tionS,ript. Разделитель
символ "c" */

trace("ActionScript".split("")); /* Выводит A,c,t,i,o,n,S,c,r,i,p,t.
Разделитель пустая строка */

trace("Per aspera ad astrum".split(" "));// Выводит Per,aspera,ad,astrum

trace("Ars longa vita brevis".split(" ", 2)); /* Выводит Ars,longa. Длина
массива ограничена двумя элементами */
```

Метод *String.toLowerCase()*

Данный метод применяется для перевода символов строки в нижний регистр и возвращает копию исходной строки, в которой все прописные буквы преобразованы в строчные. Исходное значение объекта `String` при этом не меняется. Метод имеет следующий синтаксис:

```
строка. toLowerCase()
```

Здесь *строка* — это строка, из которой извлекаются символы, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Данный метод не принимает параметров.

```
trace("DMITRI".toLowerCase()); //Выводит dmitri
var login:String = new String("AlBeRt");
trace(login.toLowerCase());
trace(login);//Выводит AlBeRt. Исходное значение не изменилось
```

Метод *String.toUpperCase()*

Данный метод применяется для перевода символов строки в верхний регистр и возвращает копию исходной строки, в которой все строчные буквы преобразованы в прописные. Исходное значение объекта `String` при этом не меняется. Метод `toUpperCase()` имеет следующий синтаксис:

```
строка.toUpperCase()
```

Здесь *строка* — это строка, из которой извлекаются символы, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Данный метод не принимает параметров.

```
trace("Dmitri".toUpperCase()); //Выводит DMITRI
```

Примечание

К преобразованию регистра часто прибегают для того, чтобы элиминировать несущественные различия строк, связанные с использованием одинаковых символов в разных регистрах. Так, строки `"Dmitri"` и `"dmitri"`, по сути, содержат одно и то же имя, но при сравнении `"Dmitri"` оказывается меньше, так как коды заглавных букв меньше кодов строчных. Если регистр символов не имеет принципиального значения при сравнении строк, во избежание возможных ошибок можно привести их к общему регистру с помощью методов `toLowerCase()` или `toUpperCase()`.

Метод *String.fromCharCode()*

Метод `fromCharCode()` позволяет получить строку на основе десятичного представления кодов составляющих ее символов. В качестве параметра методу передается набор десятичных кодов кодировки Unicode, и после выполнения метод возвращает строку, составленную из соответствующих символов. Применение этого метода эк-

вивалентно использованию Escape-последовательностей с той лишь разницей, что в Escape-последовательности код задается шестнадцатеричным (или восьмеричным) числом, а методу `fromCharCode()` передается его десятичное представление.

Метод `fromCharCode()` использует следующий синтаксис:

```
String.fromCharCode(код1,код2,...кодN)
```

В отличие от остальных методов класса `String` метод `fromCharCode()` является *статическим*. Это означает, что он вызывается непосредственно из класса без создания экземпляра класса. Поэтому для его вызова необходимо использовать имя класса.

Параметры *код1,код2,...кодN* — это десятичные коды символов, которые должны быть получены. Код может быть задан с помощью числового литерала или выражением, дающим число, диапазон значений которого составляет от 0 до 65 535.

```
trace (String.fromCharCode(8364)); // Выводит €
trace(String.fromCharCode(1044, 1080, 1084, 1072)); // Выводит Дима
var copyright:String = "Дмитрий & Елена Альберт";
trace(copyright+" "+String.fromCharCode(169)); /* Выводит Дмитрий & Елена
Альберт © */.Метод charCodeAt()
```

Данный метод позволяет получить десятичное значение кода символа, расположенного в строке по указанному индексу. Метод `charCodeAt()` возвращает целое число, лежащее в диапазоне от 0 до 65 535. Данный метод имеет следующий синтаксис:

```
строка.charCodeAt(индекс)
```

Здесь *строка* — это строка, из которой извлекаются символы, заданная как строковый литерал или идентификатор, содержащий строковое значение.

Параметр *индекс* — задает индекс символа в строке, код которого должен быть получен. Этот параметр может быть числовым литералом или выражением, дающим число. Если значение параметра *индекс* не лежит в допустимом диапазоне (от `0` до *строка*`.length-1`), метод возвращает значение `NaN` (Not a Number).

```
trace("Дмитрий".charCodeAt(0)); // Выводит 1044 - код символа "Д"
```

Таблица 5.4. *Операции со строками в ActionScript*

Операции со строками	Способ выполнения	Пример	Результат
Конкатенация (соединение) строк			
Соединение строк	1. Оператор +	`"a"+"1"`	`"a1"`
	2. Метод `concat()`	`"a".concat("b", "c")`	`"abc"`
Соединение строк с присваиванием	Оператор +=	`"a"+="b"`	`"ab"`

Таблица 5.4 *(продолжение)*

Операции со строками	Способ выполнения	Пример	Результат
Сравнение строк по порядку символов и алфавиту			
Равенство	Оператор `==`	`"A" == "a"`	`false`
Неравенство	Оператор `!=`	`"A" == "a"`	`true`
Больше	Оператор `>`	`"a" > "9"`	`true`
Больше или равно	Оператор `>=`	`"a" >= "9"`	`true`
Меньше	Оператор `<`	`"a" < "aa"`	`true`
Меньше или равно	Оператор `<=`	`"A" <= "z"`	`true`
Поиск в строке			
Возврат количества символов в строке	Свойство `length`	`"Lena".length`	`4`
Возврат символа по указанному индексу	Метод `charAt()`	`"Lena".CharAt(2)`	`"n"`
Возврат индекса первого вхождения искомого символа в строку, начиная с заданного индекса	Метод `indexOf()`	`"Dmitri".indexOf("i",3)` Поиск индекса первого символа `"i"`, начиная с символа с индексом 3 (`"t"`) Поиск слева направо	`5`
Возврат индекса первого вхождения искомого символа в строку с конца строки, начиная с заданного индекса	Метод `lastIndexOf()`	`"Dmitri".lastIndexOf("i", 4)` Поиск индекса первого с конца строки символа `"i"`, начиная с символа с индексом 4 (`"r"`). Поиск справа налево	`2`
Извлечение частей строк			
Извлечение части строки на основании заданных значений индекса начального символа подстроки и индекса символа, следующего за подстрокой	Метод `substring()`	`"ActionScript".substring(0, 6)` Подстрока извлекается, начиная с 0-го символа (`"A"`) и помимо него включает пять следующих символов. Символ с индексом 7 (`"S"`) не входит в подстроку	`"Action"`
Извлечение части строки на основании начального индекса символа подстроки и длины подстроки	Метод `substr()`	`"ActionScript".substr(-12, 6)` Подстрока извлекается, начиная с 12-го с конца символа (`"A"`) и включает 6 символов	`"Action"`

Таблица 5.4 *(окончание)*

Операции со строками	Способ выполнения	Пример	Результат
Извлечение частей строк			
Извлечение части строки на основании индекса начального и конечного символов подстроки, заданных относительно начала или конца строки	Метод `slice()`	`"ActionScript".slice(-12, 3)` Подстрока извлекается, начиная с 12-го символа с конца и до 2-го символа с начала включительно	`"Act"`
Извлечение подстрок, разделенных указанным набором символов, и помещение их в массив	Метод `split()`	`"A Bc De F".split(" ", 2)` Извлечение подстрок, разделенных пробелами. Длина массива ограничена двумя элементами	`["A", "Bc"]`
Преобразование регистра символов			
Возврат копии строки, преобразованной в верхний регистр	Метод `toUpperCase()`	`"Dmitri".toUpperCase()`	`"DMITRI"`
Возврат копии строки, преобразованной в нижний регистр	Метод `toLowerCase()`	`"DMITRI".toLowerCase()`	`"dmitri"`
Работа с кодами символов строки			
Создание символьной строки на основании десятичных кодов символов	Метод (статический) `fromCharCode()`	`String.fromCharCode(169)` Возвращает символ с кодом 169	©
Определение десятичного кода символа с заданным индексом в строке	Метод `charCodeAt()`	`"Дмитрий".charCodeAt(0)` Возвращает код символа `"Д"`	`1044`

Булев тип

Данные булева типа используются для проверки истинности или ложности отношений, условий, а также для проверки состояния объектов. Булев тип определяет всего два значения: истина и ложь. В качестве литералов этих значений используются специальные константы `true` и `false`.

Оболочкой данных этого типа является класс `Boolean`, включающий методы, позволяющие получить значение объекта `Boolean` и его строковое представление.

Объект `Boolean` может быть создан с помощью оператора `new` и конструктора, однако этот способ на практике используется очень редко.

```
var bool:Boolean = new Boolean(true);
```

В основном при необходимости использования булева значения применяют его литерал, который можно сохранить в переменной.

```
var bool:Boolean = true;
```

При всей своей простоте булев тип имеет чрезвычайно важное значение, поскольку определяемые им данные являются результатом выполнения операндами множества широко используемых операций. К их числу можно отнести следующие:

- операции отношений (больше, меньше, равно и т. д.) возвращают значение булева типа;
- в качестве условного выражения в условном предложении используется выражение, дающее булево значение;
- булевы значения используются при выполнении логических операций;
- методы или функции после выполнения могут возвращать булево значение, служащее индикатором успешности произведенной операции.

Так, например, когда интерпретатор встречает выражение

```
2+6*3>=100/5,
```

он вычисляет значения операндов слева и справа от оператора >=, после чего оценивает истинность их отношения и выдает значение `true`. Проверка различных условий — это очень распространенная задача. Так, например, если клип должен двигаться внутри прямоугольной области, отталкиваясь от стенок, возникает необходимость проверки того, что координаты клипа не выходят за пределы координат границ. Поскольку очень часто необходимо контролировать сразу несколько условий, возникает необходимость в обработке нескольких условных выражений. Так, в рассмотренном примере нужно следить, чтобы координаты клипа не вышли за допустимые границы и по вертикали, и по горизонтали. В таких ситуациях используются логические операции, которые проводятся как раз над данными булева типа.

Логические операции

Результатом выполнения логической операции является булево значение. Для их выполнения применяются логические операторы. Два из них являются бинарными (то есть имеют два операнда), а один — унарный (имеет только один операнд). Логические операторы принимают в качестве операндов булевы значения. Если операнды имеют значения другого типа, интерпретатор преобразовывает их в булев тип согласно правилам, описанным в следующем разделе настоящей главы. Определено три логические операции:

- логическое ИЛИ — в результате этой операции возвращается булево значение `true`, если **хотя бы один** из операндов имеет булево значение `true`. В остальных случаях возвращается значение `false`. Таким образом, для того чтобы получить значение `true`, достаточно, чтобы хотя бы один операнд имел значение `true`.

Эта операция носит название *логического сложения*. Для выполнения логического сложения используется оператор `||`;

Примечание

Символ вертикальной черты | на клавиатуре (как правило) расположен справа от клавиши <Backspace> в правом верхнем углу.

- логическое И — в результате этой операции возвращается булево значение `true`, если оба операнда имеют булево значение `true`. В остальных случаях возвращается значение `false`. Таким образом, для того чтобы получить значение `true`, необходимо, чтобы **оба** операнда имели значение `true`. Эта операция носит название *логического умножения*. Для выполнения логического умножения используется оператор `&&`;
- логическое отрицание (НЕ) — в результате этой операции возвращается булево значение, противоположное булеву значению операнда. Для выполнения логического отрицания используется оператор `!`.

Вспомнив, что числовыми эквивалентами значений `true` и `false` являются 1 и 0, можно очень просто запомнить правила выполнения логических операций.

При логическом сложении можно сложить числовые эквиваленты операндов. Например, если один операнд имеет значение `true`, а другой `false`, то в результате сложения 1 и 0 получаем 1, то есть `true`. Естественно, если оба значения `true` — результат тоже `true`.

При логическом умножении можно умножить числовые эквиваленты операндов. Например, если один операнд имеет значение `true`, а другой `false`, то в результате умножения 1 и 0 получаем 0, то есть `false`.

Правила выполнения логических операций обобщены в табл. 5.5.

Таблица 5.5. *Логические операции*

Операция	Оператор	Операнд 1	Операнд 2	Результат
Логическое ИЛИ (логическое сложение)	`||`	`true (1)`	`true (1)`	`true (1)`
		`true (1)`	`false (0)`	`true (1)`
		`false (0)`	`true (1)`	`true (1)`
		`false (0)`	`false (0)`	`false (0)`
Логическое И (логическое умножение)	`&&`	`true (1)`	`true (1)`	`true (1)`
		`true (1)`	`false (0)`	`false (0)`
		`false (0)`	`true (1)`	`false (0)`
		`false (0)`	`false (0)`	`false (0)`
Логическое отрицание НЕ (унарная операция)	`!`	`true (1)`	`отсутствует`	`false (0)`
		`false (0)`	`отсутствует`	`true (1)`

Ниже приводится несколько примеров, иллюстрирующих выполнение логических операций, и применение логических операторов.

```
var x = 3;
var y = 5;
var z = 10;
trace(x<5 || y>5);// Выводит true, так как выполняется условие x<5
trace(x*y==z+y || x==0);/* Выводит true, так как выполняется условие
x*y==z+y */
trace(x>0 && x<=5);//Выводит true, так как x входит в полуинтервал (0;5]
trace(x+y<z && x>y);/* Выводит false, так как условие x > y не
выполняется */
var i:Boolean = true;
trace(!i); //Выводит false
var i:Boolean = true;
trace(!i); //Выводит false
trace(x<y && y<z && z<20); // Выводит true, так как все три условия
                           // выполняются
trace(x<5 || y>10 && z==10); /* Выводит true, так как выполняются условия
x<5 и z==10 */
```

Встроенные методы для работы с булевыми значениями

Как и класс `Number`, класс `Boolean` включает два метода: `toString()` и `valueOf()`. Для вызова этих методов необходимо использовать объект `Boolean` или переменную, содержащую булево значение.

Метод *Boolean.toString()*

Метод `toString()` возвращает строковое представление своего значения в виде строки "`true`" или "`false`". Данный метод имеет следующий синтаксис:

булеан`.toString()`

Здесь *булеан* — идентификатор объекта `Boolean` или переменной, содержащей булево значение.

```
var bool:Boolean = true;
bool.valueOf(); // Возвращает строку "true"
```

Метод *Boolean.valueOf()*

Данный метод не принимает параметров и возвращает элементарное значение объекта `Boolean` (`true` или `false`).

Данный метод имеет следующий синтаксис:

булеан`.valueOf()`

Здесь *булеан* — идентификатор объекта `Boolean` или переменной, содержащей булево значение.

```
var bool:Boolean = true;
bool.valueOf(); // Возвращает булево значение true
```

Типы *null* и *undefined*

Типы `null` и `undefined` — это особые типы данных, каждый из которых имеет только одно значение. Оба этих значения применяются для того, чтобы показать отсутствие данных.

Значение *null*

Значение `null` является единственным значением типа данных `null`. Оно означает отсутствие данных и, в соответствии с документацией по ActionScript, может быть использовано в следующих случаях:

- для указания того, что переменная еще не получила значения;
- для указания того, что переменная уже не содержит значения;
- в качестве возвращаемого функцией значения, если нет данных, которые должны быть возвращены;
- в качестве параметра функции для указания того, что параметр опускается. Выполнение арифметических операций со значением `null` дает значение `NaN`.

Значение *undefined*

Значение `undefined` используется в случае, когда переменная не содержит значения. Между объявлением переменной и присваиванием ей значения переменная содержит значение `undefined`.

Чрезвычайно важным является тот факт, что при использовании несуществующей переменной в выражении она принимает значение `undefined`.

```
trace(x++); // Выводит undefined, переменной x не существует
```

Выполнение арифметических операций со значением `undefined` дает значение `NaN`.

Примечание

Следует отметить, что при сравнении значения `null` и `undefined` считаются равными:

```
trace(undefined == null); // Выводит true
```

Значения `undefined` при сравнении также считаются равными:

```
trace(undefined == undefined); // Выводит true
trace(undefined >= undefined); // Выводит true
trace(undefined <= undefined); // Выводит true
```

Явное и автоматическое преобразование элементарных типов данных

Преобразование типов данных имеет место в ситуации, когда имеются данные одного типа, а контекст требует данных другого типа. Так, например, оба операнда оператора умножения (*) должны иметь числовой тип. Если их тип иной, необходимо выполнить преобразование в числовой тип. Преобразование типа может выполняться явно — с помощью специальных функций — или косвенным образом, то есть автоматически.

Информация о возможностях преобразования элементарных типов данных обобщена в табл. 5.6 и подробно рассмотрена далее в этом разделе.

Таблица 5.6. *Способы преобразования элементарных типов данных в ActionScript*

Способ преобразования		Тип данных, к которому происходит преобразование		
		Number	**String**	**Boolean**
Исходный тип данных	**Number**		1. Функция `String()` 2. Метод `toString()` 3. Сложение со строкой	Функция `Boolean()`
	String	1. Функция `Number()` 2. Функция `parseInt()` 3. Функция `parseFloat()` 4. Вычитание, умножение или деление на число		Функция `Boolean()`
	Boolean	Функция `Number()`	Функция `String()`	

Явное преобразование типов данных

Явное преобразование типов данных осуществляется с помощью встроенных глобальных функций. Необходимость в явном преобразовании может быть вызвана тем, что имеющиеся данные относятся к типу, не поддерживающему требуемую процедуру, или предусмотрительным желанием разработчика избежать случайных ошибок несоответствия типов данных.

Ситуация, требующая явного преобразования типов, может возникнуть, например, при создании формы, предусматривающей ввод пользователем числовых значений для последующих вычислений. Для этих целей используются пользовательские (input) текстовые поля. Значение, которое пользователь вводит в такие поля, может включать только числовые символы, однако, несмотря на это, всегда имеет строко-

вый тип. Если такие значения должны арифметически складываться, то их перед сложением необходимо преобразовать в числовой тип. В противном случае вместо сложения будет выполнена конкатенация, то есть соединение строк:

```
"2"+"2"="22"
```

Следует отметить, что при выполнении явного преобразования типа данных это преобразование выполняется только с копией данных. Тип исходных данных остается неизменным. Это иллюстрируется следующим примером:

```
var n:Number = 5; // Создаем числовую переменную
String(n); // Преобразуем копию значения переменной n в строку
trace(typeof n); // Выводит number. Значение переменной n не изменилось
```

Преобразование в число

Для преобразования литерала или выражения в число используется глобальная функция `Number()`, имеющая следующий синтаксис:

`Number(`*выражение*`)`

Здесь *выражение* — это литерал или выражение, которое необходимо преобразовать в число.

Функция `Number()` возвращает конкретное значение числового типа или значение `NaN`, которое хотя и имеет числовой тип, однако не является настоящим числом.

Результаты преобразования различных типов данных и особых значений в числовой тип представлены в табл. 5.7.

Для преобразования строковых данных в число можно воспользоваться встроенными глобальными функциями `parseInt()` и `parseFloat()`.

Функция *parseInt()*

Эта функция анализирует исходную строку и выделяет из нее целое число, которое возвращается в качестве результата. Если строка не может быть преобразована в число, функция возвращает значение `NaN`. Данная функция имеет следующий синтаксис:

`parseInt(`*выражение*`, `*основание*`)`

Параметр *выражение* — это строковый литерал или идентификатор переменной, содержащей строковое значение, которое должно быть преобразовано в число. Если значение данного параметра начинается с символов `0x`, функция возвращает десятичное представление выделенного из строки шестнадцатеричного числа. Если значение параметра *выражение* начинается с нуля, то функция возвращает десятичное представление выделенного из строки восьмеричного числа. Если значение параметра *выражение* не начинается с числового символа, символов `+` , `-` или пробелов, возвращается значение `NaN`. Все нечисловые символы строки *выражение*, следующие за числовыми символами, игнорируются.

Параметр *основание* — это число или идентификатор переменной, содержащей числовое значение, задающее основание системы счисления, в которой представле-

но выделяемое из строки *выражение* число. Этот параметр является необязательным и может быть опущен. По умолчанию используется основание 10 (за исключением случаев, когда строка начинается с символов `0x` или `0`).

Рассмотрим несколько примеров:

```
parseInt("3user"); // Возвращает число 3
parseInt("3.1415926"); // Возвращает число 3
parseInt("0xff"); // Возвращает число 255
parseInt(" -3"); // Возвращает число -3
parseInt("user3"); // Возвращает NaN. Первый символ нечисловой
parseInt("1001", 2); /* Возвращает число 9. Десятичное представление
двоичного числа 1001 */
```

Функция *parseFloat()*

Эта функция похожа на предыдущую, она анализирует исходную строку и выделяет из нее число с плавающей точкой, которое возвращается в качестве результата. Если строка не может быть преобразована в число, функция возвращает значение `NaN`. Данная функция имеет следующий синтаксис:

```
parseFloat(выражение)
```

Параметр *выражение* — это строковый литерал или идентификатор переменной, содержащей строковое значение, которое должно быть преобразовано в число. Если строка не начинается с числовых символов, символов `+`, `-`, пробелов или символа точки (`.`), функция возвращает значение `NaN`.

```
parseFloat(" 3.5percent"); // Возвращает число 3.5
parseFloat("3.1415926"); // Возвращает число 3.1415926
trace(parseFloat("3,1415926")); /* Возвращает число 3, так как в
качестве разделителя целой и дробной частей числа используется точка,
а не запятая */
parseFloat("-6.2e-3"); // Возвращает число -0.0062
parseFloat("0xff"); // Возвращает 0, так как далее следует нечисловой
                    // символ
```

Преобразование в строку

Для преобразования литерала или выражения в строку используется глобальная функция `String()`, имеющая следующий синтаксис:

```
String(выражение)
```

Здесь *выражение* — это литерал или выражение, которое необходимо преобразовать в строку.

Функция `String()` возвращает конкретное значение строкового типа или значение `undefined`.

Результаты преобразования различных типов данных и особых значений в строковый тип представлены в табл. 5.7.

Преобразование в булев тип

Для преобразования литерала или выражения в строку используется глобальная функция `Boolean()`, имеющая следующий синтаксис:

```
Boolean(выражение)
```

Здесь *выражение* — это литерал или выражение, которое необходимо преобразовать в булев тип.

Функция `Boolean()` возвращает булево значение `true` или `false`.

Результаты преобразования различных типов данных и особых значений в булев тип представлены в табл. 5.7.

Автоматическое преобразование типов данных

Автоматическое преобразование типов данных происходит в ситуациях, когда контекст выполняемой операции требует наличия данных иного типа, чем существующий тип. Так, например, логический оператор ИЛИ (`||`) используется только с операндами булева типа, поэтому если его операнды имеют другой тип, они преобразуются в булев тип на основании правил, представленных в табл. 5.7.

Преобразование логическими операторами

Рассмотрим следующее выражение:

```
5+"10" && 2<8
```

Ни один из операндов логического И не имеет булев тип и, тем не менее, результатом оценки этого выражения является булево значение `true`. Этот результат получен вследствие вычисления выражений операндов и последующего автоматического преобразования типа результата. В выражении левого операнда выполняется сложение числа и строки, что дает в итоге строковое значение, так как оператор `+`, встретив число, автоматически преобразовал его в строку и затем выполнил конкатенацию двух строковых значений:

```
"5"+"10"="510"
```

Полученная строка затем преобразуется в булев тип, согласно правилам, описанным в табл. 5.7, и дает значение `true`.

Правый операнд представляет собой отношение, являющееся истинным и, соответственно, дающее булево значение `true`. Таким образом, исходное выражение сводится к следующему:

```
true && true
```

что, естественно, выдает значение `true`.

Преобразование арифметическими операторами

Зная правила, в соответствии с которыми выполняется автоматическое преобразование типов данных, можно намеренно выполнять преобразования, не прибегая к явному преобразованию типа. Так, например, для преобразования строки, состоящей

из символов, которые могут быть преобразованы в число, можно вычесть из нее число или умножить или разделить эту строку на число, как это показано ниже:

```
var str:String = "3.25e3";
num = str - 0; // Преобразование в строку путем вычитания нуля
trace(num); // Выводит 3250
trace(typeof num);// Выводит number
var num = str*1; // Преобразование в строку путем умножения на единицу
trace(num); // Выводит 3250
trace(typeof num);// Выводит number
num = str/1; // Преобразование в строку путем деления на единицу
trace(num); // Выводит 3250
trace(typeof num);// Выводит number
trace(typeof str);/* Выводит string. Исходное значение и его тип не
изменились */
```

Напомним, что преобразование типов данных выполняется только с копией данных. Исходный тип при этом не изменяется, что и демонстрирует последняя строка приведенного кода.

Преобразование операторами отношений

Автоматическое преобразование типа данных также может иметь место при сравнении данных разного типа с помощью операторов отношений (за исключением операторов строго равенства и строгого неравенства, не выполняющих преобразования типа). При сравнении значений, не являющихся строками, или значений разных типов с помощью операторов >, <, >=, <=, ==, != они всегда преобразуются в числовой тип.

```
x = 1;
y = "1.2";
trace(x<y); //Выводит true
a = true;
b = 0;
trace(a!=b); //Выводит true
```

При необходимости проверки равенства двух значений и их типов можно прибегнуть к использованию оператора строгого равенства ===. Этот оператор действует так же, как и оператор равенства ==, за исключением того, что автоматического преобразования типа данных операндов не происходит. В качестве иллюстрации рассмотрим следующий пример:

```
var str:String = "3.5"; // строковый тип
var num:Number = 3.5; // числовой тип
trace(str==num); // true. Выполняется преобразование типа данных
trace(str===num); // false. Преобразования типа данных не выполняется
```

Оператор строгого неравенства !== позволяет установить, что два операнда не равны, не преобразуя при этом их типы. Он возвращает значение false только в том случае, если значения операндов равны, и они имеют одинаковый тип.

```
var str:String = "5.9"; // строковый тип
var num:Number = 5.9; // числовой тип
trace(str!=num); // false. Выполняется преобразование типа данных
trace(str!==num); // true. Разные типы данных
```

Таблица 5.7. *Результат преобразования элементарных типов данных в ActionScript*

Результат преобразования		Тип, к которому происходит преобразование		
		Number	**String**	**Boolean**
Исходный тип данных/ значение	**undefined**	NaN	"undefined"[1]	false
	null	NaN[2]	"null"	false
	NaN	NaN	"NaN"	false
	Number	Number	строковое представление числа	false, если исходное число 0, true в остальных случаях
	Boolean true	1	"true"	true
	Boolean false	0	"false"	false
	String Непустая строка	Числовая строка в число, нечисловая строка в NaN	Исходная строка	true[3]
	String Пустая строка ("")	NaN	Исходная строка	false
	Infinity	Infinity (бесконечность)	"Infinity"	true
	-Infinity	-Infinity (отрицательная бесконечность)	"-Infinity"	true

[1] При публикации для Flash Player 6 результат преобразования пустая строка "".

[2] При публикации для Flash Player 6 результат преобразования 0.

[3] При публикации для Flash Player 6 строка вначале преобразуется в числовой тип. Далее применяются правила преобразования из числового типа в строковый. — Прим. автора.

Большинство операций отношений преобразуют тип значения операндов в числовой тип. В случае, когда преобразовать значение операнда в число нельзя, используется значение NaN. Если один из двух или оба операнда имеют значение NaN, результат сравнения этих операндов зависит от оператора отношений. В табл. 5.8 показаны результаты сравнения операндов, один или оба из которых имеют значение NaN.

Таблица 5.8. *Результат сравнения операндов, имеющих значение* NaN

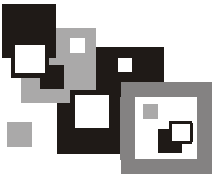
Оператор	Результат	Оператор	Результат
>	undefined	==	false
<	undefined	===	false
>=	true	!=	true
<=	true	!==	true

Приведем несколько примеров в качестве иллюстрации:

```
var wrongString:String = "2PI"
var num:Number = 3;
trace (wrongString<3); // Выводит undefined
trace (wrongString<=3); // Выводит true
trace (wrongString==3); // Выводит false
```

Описание синтаксиса каждого оператора, предложения, функции и т. д. включает указание типа данных, с которым оперирует данный элемент. Попытка использования данных другого типа в большинстве случаев приводит либо к его автоматическому преобразованию, либо к ошибке. Знание правил использования элементов ActionScript позволит избежать случайных ошибок, связанных с автоматическим преобразованием типа данных.

Глава 6



Выражения и операторы

Доступность операции — еще не показатель того, что ее надо делать.

Комментарий лорда Коэна

Выражения являются основными средствами вычисления в программе. Выражение состоит из *операторов*, определяющих выполняемое действие, и *операндов*, задающих элементы данных, над которыми производятся требуемые операции. Многие операторы и выполняемые ими действия уже были рассмотрены в предыдущих главах. Данная глава описывает еще не изученные операторы и содержит обобщенные сведения об операторах, рассмотренных ранее.

Операторы могут быть классифицированы по числу принимаемых ими операндов. Различают три типа операторов:

- унарные (unary) — операторы, имеющие только один операнд, например, оператор логического отрицания (`!`);
- бинарные (binary) — операторы, имеющие два операнда. К этому типу относится большая часть операторов ActionScript;
- тернарные (ternary) — операторы, имеющие три операнда. Примером такого оператора является условный оператор (`?:`), речь о котором пойдет в *главе 7*.

Если один и тот же оператор применяется для выполнения различных действий, он называется *перегруженным*. Единственным перегруженным оператором ActionScript является оператор `+`, поскольку он применяется для выполнения арифметического сложения и для конкатенации (соединения) строк.

Приоритет операторов

Если выражение включает сразу несколько операторов, то возникает вопрос, в каком порядке будут выполняться определяемые ими действия. Рассмотрим следующий пример:

```
a + b * c
```

В данном выражении вначале будет выполнена операция умножения, и затем результат умножения (`b * c`) будет суммирован со значением `a`. Порядок выполнения действий в выражении определяется приоритетом используемых операторов. *При-*

оритет оператора (operator precedence) определяет очередность выполнения операций при вычислении выражения.

В табл. 6.1 приведены данные о назначении и приоритете операторов ActionScript. Приоритет каждого оператора здесь условно обозначен числом. Наиболее высокое значение (16) соответствует высшему приоритету, наиболее низкое значение — низшему приоритету. В таблице операторы представлены в порядке понижения приоритета.

Зная приоритет оператора, можно всегда легко определить порядок выполнения действий операторов с разным приоритетом. Так, из таблицы видно, что оператор умножения имеет более высокий приоритет, чем оператор сложения, поэтому операция умножения выполняется перед сложением.

Если несколько операторов имеют равный приоритет, то при их одновременном использовании в выражении порядок действий определяется правилами ассоциативности.

Таблица 6.1. *Приоритет операторов в ActionScript*

Оператор	Описание	Приоритет	Ассоциативность[1]	Число операндов	Тип данных операндов
`x--`	постфиксное инкрементирование	16 (высший)	→	1	`Number`
`x++`	постфиксное декрементирование	16	→	1	`Number`
`.`	доступ к свойству объекта	15	→	2	`любой`
`[ ]`	доступ к элементу массива	15	→	1	`Number/ String`
`()`	круглые скобки	15	→	>= 0	`любой`
`++x`	префиксное инкрементирование	14	←	1	`Number`
`--x`	префиксное декрементирование	14	←	1	`Number`
`-`	унарное отрицание	14	←	1	`Number`
`~`	поразрядное отрицание	14	←	1	`Number`
`!`	логическое отрицание	14	←	1	`Boolean`

[1] → — ассоциативность слева направо.
← — ассоциативность справа налево.

Таблица 6.1 *(продолжение)*

Оператор	Описание	Приоритет	Ассоциативность	Число операндов	Тип данных операндов
new	создание объекта	14	←	1	Object (Function)
delete	удаление объекта/свойства/метода/элемента массива	14	←	1	любой
typeof	определение типа данных	14	←	1	любой
void	возврат неопределенного значения	14	←	1	любой
*	умножение	13	→	2	Number
/	деление	13	→	2	Number
%	остаток от деления	13	→	2	Number
+	сложение	12	→	2	Number/ String
-	вычитание	12	→	2	Number
<<	поразрядный сдвиг влево	11	→	2	Number
>>	поразрядный сдвиг вправо со знаком	11	→	2	Number
>>>	поразрядный сдвиг вправо без знака	11	→	2	Number
<	меньше	10	→	2	Number/ String
<=	меньше или равно	10	→	2	Number/ String
>	больше	10	→	2	Number/ String
>=	больше или равно	10	→	2	Number/ String
instanceof			→	1	Object
==	равенство	9	→	2	любой
===	строгое равенство	9	→	2	любой
!=	не равно	9	→	2	любой
&	поразрядное И	8	→	2	Number

Таблица 6.1 *(окончание)*

Оператор	Описание	Приоритет	Ассоциативность	Число операндов	Тип данных операндов
^	поразрядное исключающее ИЛИ	7	→	2	Number
|	поразрядное ИЛИ	6	→	2	Number
&&	логическое И	5	→	2	Boolean
||	логическое ИЛИ	4	←	2	Boolean
?:	условный оператор	3	←	3	—
=	присваивание	2	←	2	любой
+=	сложение с присваиванием	2	←	2	Number/ String
-=	вычитание с присваиванием	2	←	2	Number
*=	умножение с присваиванием	2	←	2	Number
/=	деление с присваиванием	2	←	2	Number
%=	деление по модулю с присваиванием	2	←	2	Number
<<=	поразрядный сдвиг влево с присваиванием	2	←	2	Number
>>=	поразрядный сдвиг вправо с присваиванием	2	←	2	Number
>>>=	поразрядный сдвиг вправо (без знака) с присваиванием	2	←	2	Number
&=	поразрядное & с присваиванием	2	←	2	Number
^=	поразрядное XOR с присваиванием	2	←	2	Number
|=	поразрядное ИЛИ с присваиванием	2	←	2	Number
,	запятая	1 (низший)	←	>= 2	любой

Ассоциативность

Ассоциативность операторов определяет, в каком порядке выполняются действия, заданные операторами с равным приоритетом. Ассоциативность оператора может быть левосторонней или правосторонней. Это означает, что операторы с равным приоритетом могут выполняться слева направо или справа налево, соответственно. Правила ассоциативности приведены в табл. 6.1. Здесь символ → означает левостороннюю ассоциативность, а символ ← правостороннюю ассоциативность.

В выражении

```
a + b - c
```

вначале выполняется сложение, а затем вычитание, поскольку для этих операторов используется левосторонняя ассоциативность.

А в выражении

```
a = b = c
```

присваивание выполняется справа налево в соответствии с правосторонней ассоциативностью.

Применение скобок позволяет изменить не только правила ассоциативности, но и приоритеты операторов. С помощью скобок можно четко организовать порядок выполнения действий в выражении и получить ясную, хорошо читаемую запись.

Если рассмотренное ранее выражение

```
a + b * c
```

переписать с использованием скобок, то можно добиться того, чтобы вначале выполнялось сложение и лишь затем умножение:

```
(a + b) * c
```

Применение скобок позволяет не беспокоиться о различных приоритетах операторов и не запоминать правила ассоциативности. При записи сложных выражений скобки могут быть вложенными. В этом случае вначале выполняются действия во внутренних скобках, а затем во внешних.

```
((1+2)*3+1)*3 = 30
```

При использовании вложенных скобок важно следить за их соответствием. Количество открывающих скобок должно совпадать с количеством закрывающих. В противном случае имеет место синтаксическая ошибка.

Модификация значения

Значение, хранящееся в переменной или в свойстве объекта, часто приходится модифицировать, то есть изменять. Например, для того чтобы заставить клип плавно двигаться по горизонтали, необходимо постепенно наращивать (или уменьшать) значение его горизонтальной координаты. Для того чтобы многократно выполнить определенную операцию в цикле, необходимо на каждой итерации увеличивать значение счетчика цикла. Таких примеров можно привести очень много.

Вопросы, связанные с выполнением арифметических операций, а также операция конкатенации были рассмотрены в *главе 5, в разд. "Числовой тип. Работа с числами"* и *"Строковый тип. Работа со строками"*, в связи с этим в данном разделе речь пойдет о конкретных способах изменения текущего значения переменной.

Простое присваивание

Для изменения значения переменной можно использовать операцию присваивания, при котором данная переменная также используется и в правой части выражения. Рассмотрим это на следующем примере.

```
var x:Number = 5; // Создаем числовую переменную со значение 5
x=x+10; // Увеличиваем значение x на 10
```

Для того чтобы понять смысл этого выражения, необходимо обратиться к табл. 6.1 и оценить приоритеты используемых операторов. Из таблицы видно, что оператор сложения имеет более высокий приоритет, чем оператор присваивания. В связи с этим вначале выполняется вычисление выражения правого операнда оператора присваивания. В этом выражении текущее значение переменной `x` складывается с числом 10, что в итоге дает 15. После этого получившийся результат сложения присваивается переменной `x` в качестве нового значения. Выполним проверку:

```
trace (x); // Выводит 15
```

Фактически изменение значения переменной можно выполнять, просто присваивая ей новое значение, как константу:

```
var x:Number = 5; // Создаем числовую переменную со значение 5
x=15; // Значение x увеличилось на 10
```

Несмотря на то, что внешне в первом и во втором случае был получен одинаковый результат, эти подходы принципиально различны. Различие становится очевидным при многократном выполнении выражений, осуществляющих изменение значения. В первом случае при многократном выполнении выражения.

```
x=x+10;
```

значение переменной `x` будет всякий раз увеличиваться на 10 по сравнению с ее текущим значением. Таким образом, будет происходить постоянное наращение значения или модификация.

Во втором случае переменной `x` будет всякий раз присваиваться одно и то же фиксированное значение и никакой модификации не произойдет.

Модификация значения с использованием присваивания может выполняться с применением любых арифметических операторов. Приведем пример:

```
var x:Number = 25;
x=x-10; // Уменьшаем текущее значение x на 10
trace (x); // Выводит 15
x=x*10; // Увеличиваем текущее значение x в десять раз
trace (x); // Выводит 150
```

```
x=x/10; // Уменьшаем текущее значение x в десять раз
trace (x); // Выводит 15
```

Модифицировать текущее значение с использованием присваивания можно и при работе со строковыми переменными. В этом случае в правой части выражения присваивания применяется оператор сложения:

```
var str:String = "A";
str = str + "B";
trace(str); // Выводит AB
```

Составное присваивание

Составное присваивание — это сокращенная форма присваивания, эквивалентная присваиванию с использованием целевой переменной в правой части выражения присваивания, рассмотренному выше. Для такой формы записи применяются *составные операторы* присваивания, объединяющие требуемый бинарный оператор и оператор присваивания. Применение составных операторов позволяет сократить запись и делает программный код более простым для восприятия.

Следующие выражения попарно эквивалентны:

```
x = x + 5;
x += 5;// Сложение с присваиванием
x = x - 10;
x -= 10;// Вычитание с присваиванием
x = x * 10;
x *= 10;// Умножение с присваиванием
x = x / 10;
x /= 10;// Деление с присваиванием
x = x % 5;
x %= 5;// Деление по модулю с присваиванием
```

Таким образом, применение составных операторов сводится к тому, что в качестве левого операнда используется идентификатор, значение которого модифицируется, а в качестве правого операнда используется значение модификации. Используемый составной оператор определяет, **на** сколько или **во** сколько раз увеличивается или уменьшается текущее значение переменной.

Инкрементирование/декрементирование

Операторы инкрементирования и декрементирования — это унарные операторы, то есть операторы, имеющие только один операнд. Инкрементирование увеличивает значение своего операнда на единицу, а декрементирование, соответственно, уменьшает значение своего операнда на единицу. Эти операторы позволяют модифицировать текущее значение своего операнда с шагом в единицу.

Следующие предложения иллюстрируют их применение.

```
myVar = 5;
myVar ++; //Текущее значение myVar увеличивается на единицу
trace(myVar); // Выводит 6
myVar --; // Текущее значение myVar уменьшается на единицу
trace(myVar); // Выводит 5
```

Часто применение инкрементирования и декрементирования эквивалентно использованию составного сложения или составного вычитания единицы, однако позволяет использовать более компактную запись. Так, следующие два выражения увеличивают значение `x` на единицу:

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

Инкрементирование и декрементирование бывает префиксным (если оператор расположен перед своим операндом) или постфиксным (если оператор расположен после своего операнда):

```
++ nVar; // префиксное инкрементирование
nVar ++; // постфиксное инкрементирование
```

Чрезвычайно важно запомнить следующее правило.

Префиксный оператор изменяет значение своего операнда на единицу и возвращает полученный *результат*, в то время как постфиксный оператор изменяет значение своего операнда на единицу и возвращает его *исходное значение*. Следующий пример иллюстрирует эту особенность.

```
x=5;
y=x--; /* x уменьшается на единицу и становится равным 4, а y принимает
исходное значение переменной x, равное 5*/
trace (y); // Выводит 5
trace (x); // Выводит 4
y=--x; /* x уменьшается на единицу и становится равным 3, после чего y
принимает текущее значение x, равное 3 */
trace (y); // Выводит 3
trace (x); // Выводит 3
```

Другой пример, иллюстрирующий различие префиксного и постфиксного инкрементирования, можно найти в *разд. "Цикл while" главы 7*.

Инкрементирование/декрементирование часто применяется при необходимости постепенного изменения значения определенного параметра с шагом в единицу. Зачастую эти операторы применяются в циклах для модификации значения счетчика. Инкрементирование или декрементирование может быть использовано для плавного изменения прозрачности или скорости перемещения объекта, для динамического генерирования имен объектов путем прибавления к постоянной части имени объекта числа, отличающегося от предыдущего на единицу, и др.

Побитовые операции

До сих пор при работе с числовыми данными мы оперировали с десятичными числами. Большинство задач, связанных с обработкой чисел, как правило, может быть решено с помощью их десятичного представления. Система счисления с основанием 10 применяется и в среде ActionScript, в связи с чем числа, заданные в иных системах счисления, при выводе результата все равно преобразуются в десятичные. В этом можно легко убедиться, попробовав вывести шестнадцатеричное число в панель **Output** с помощью функции `trace()`:

```
trace(0x0000FF); // выводит 255
```

Тем не менее ActionScript обладает возможностями, позволяющими оперировать с двоичными числами, то есть с числами в их *битовом* представлении. Операции с числами в их битовом представлении называются *побитовыми* или поразрядными. Выполнение побитовых операций осуществляется посредством побитовых операторов. Побитовые операции выполняются только над целыми числами, поскольку в числах с плавающей точкой последовательность битов кодирует не само число, а его особое математическое представление.

Использование побитовых операций позволяет более эффективно осуществлять обработку данных за счет более рационального использования памяти. Так, например, при необходимости контроля набора параметров, принимающих значения `true` или `false`, вместо определения отдельной переменной для каждого такого параметра можно использовать всего одно число, битовое представление которого будет определять значения параметров. В результате существенно сокращается количество операций и увеличивается скорость обработки данных. Применение побитовых операций также оказывается чрезвычайно удобным при работе с цветовыми значениями RGB.

Перечень побитовых операторов и примеров выполняемых ими операций представлен в табл. 6.2 (числовые значения представлены в двоичной и десятичной системах счисления).

Таблица 6.2. *Побитовые операции*

Побитовая операция	Оператор	Операнд 1	Операнд 2	Результат
Логические операции				
Побитовое И	`&`	10101 21	01010 10	00000 0
Побитовое ИЛИ	`|`	10101 21	01010 10	11111 31
Побитовое исключающее ИЛИ	`^`	11101 29	01011 11	10110 22
Побитовое отрицание (НЕ)	`~`	01000 8	отсутствует	11111111111111111111111111110111 –9

Таблица 6.2 *(окончание)*

Побитовая операция	Оператор	Операнд 1	Операнд 2	Результат
Операции сдвига				
Сдвиг влево с учетом знака	<<	00001 1	4	10000 16
Сдвиг вправо с учетом знака	>>	01000 8	3	00001 1
Сдвиг влево без учета знака	>>>	00001 1	2	00000 0

Перед тем как приступить к изучению побитовых операторов и выполняемых ими действий, выясним, что собой представляет двоичная система счисления и как выглядит битовое представление числа.

Битовое представление числа

В основании двоичной системы счисления, как следует из названия, находится число 2. Это означает, что любое двоичное число может быть представлено в виде суммы степеней числа 2, умноженных на 1 или на 0. Каждый разряд в составе двоичного числа называется *битом* и может содержать только одну из двух цифр: ноль или единицу. Таким образом, двоичное число записывается исключительно с помощью нулей и единиц. Если бит имеет значение 1, говорят, что он *установлен*, если бит принимает значение 0, говорят, что бит *сброшен* или очищен.

Нумерация битов числа производится справа налево, начиная с нуля. Таким образом, самый правый разряд указывает состояние нулевого бита, следующий — состояние первого бита и т. д. Это иллюстрируется далее на примере двоичного числа 1001011, соответствующего десятичному числу 75.

Номер бита	31	...	7	6	5	4	3	2	1	0
Значение бита	0	...	0	1	0	0	1	0	1	1

Каждый разряд двоичного числа представляет собой произведение значения бита на степень числа 2, соответствующую номеру этого бита. Так, нулевой бит соответствует нулевой степени числа 2 (2^0), первый бит соответствует первой степени числа 2 (2^1) и т. д. Для того чтобы перевести двоичное число в десятичное представление, нужно для каждого разряда умножить значение бита на соответствующую степень числа 2 и сложить полученные результаты.

Рассмотрим несколько примеров:

$1 = 0 \times 2^0$

$10 = 1 \times 2^1 + 0 \times 2^0 = 2$

$100 = 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4$

$1000 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8$

$10000 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 16$

$1001011 = 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 =$
$= 64 + 8 + 2 + 1 = 75$

При выполнении побитовых операций используются 32-разрядные двоичные числа. Это означает, что любое двоичное число записывается с помощью 32 цифр, каждое из которых может быть нулем или единицей. Биты, непосредственно используемые для записи числа, называются значащими. Так, в числе 16 (двоичное 10000) значимыми являются первые пять битов. Остальные биты этого числа сброшены. Нулевые (незначащие) биты в левой части числа обычно при записи опускаются. Старший бит (31-й по счету), расположенный слева, отвечает за знак числа. Если число положительное, — 31-й бит нулевой, если число отрицательное, — 31-й бит равен единице. Двоичное представление отрицательного числа можно получить следующим образом:

1. Отбросить знак десятичного числа.
2. Вычесть единицу из десятичного числа.
3. Представить результат в двоичном виде.
4. Инвертировать биты, то есть заменить нули единицами и единицы нулями.

Приведем несколько примеров:

1	Десятичное число:	6
	Двоичное число:	0 0000 0000 0000 0000 0000 0000 0000 110
	Двоичное представление отрицательного числа –6	1 1111 1111 1111 1111 1111 1111 1111 010
2	Десятичное число:	15
	Двоичное число:	0 0000 0000 0000 0000 0000 0000 0001 111
	Двоичное представление отрицательного числа –15	1 1111 1111 1111 1111 1111 1111 1110 001
3	Десятичное число:	21
	Двоичное число:	0 0000 0000 0000 0000 0000 0000 0010 101
	Двоичное представление отрицательного числа –21	1 1111 1111 1111 1111 1111 1111 1101 011

Примечание

Следует иметь в виду, что результаты всех побитовых операций возвращаются в виде десятичного целого числа.

Для получения двоичного представления десятичного числа можно воспользоваться методом `Number.toString`.

```
var num = 237;
trace(num.toString(2)); // Выводит строку 11101101
```

Для получения десятичного представления двоичного числа можно воспользоваться функцией `parseInt()`.

```
trace(parseInt("1101101001",2)); // Выводит число 873
```

Функция `parseInt()` и метод `Number.toString` подробно обсуждались в *главе 5*.

Применение двоичных чисел

Двоичные числа и побитовые операции часто используются в ситуации, когда необходимо обрабатывать большое количество булевых переменных, называемых флагами (flag). Переменная-флаг служит для контроля состояния объекта, которое может быть определено в терминах булева типа данных. Так, например, для контроля анимации клипа можно использовать переменную-флаг, принимающую значение `true`, когда анимация воспроизводится; и значение `false` — в случае остановки. Вместо булевых значений часто используются эквивалентные числовые значения 1 и 0. Если подобным образом приходится контролировать несколько клипов, может потребоваться большое количество переменных. Используя двоичные числа, можно представить состояние нескольких клипов одним-единственным числом и произвольно управлять флагами для каждого клипа с помощью побитовых операций.

Рассмотрим следующий пример. Целью разработчика является обеспечить пользователю возможность указывать, какие мультимедийные элементы могут быть включены, а также определить, разрешается или нет сохранять информацию локально на стороне клиента. Для этих целей можно было бы создать пять переменных и сохранить в них информацию о выбранных пользователем настройках.

```
var bitmap = 1; // Включение растровой графики (фото)
var animation = 1; // Включение анимации
var audio = 1; // Включение звука
var video = 1; // Включение видео
var localStorage = 1; /* Включение возможности локального сохранения данных */
```

Для изменения настроек нужно присвоить переменной другое значение:

```
var audio = 0; // Отключение звука
var video = 0; // Отключение видео
```

Такой подход имеет право на существование, и, если перед разработчиком не стоит задача максимально оптимизировать программу, можно пойти таким менее изящным, но более простым и понятным путем.

Тем не менее приведенные пять параметров настроек могут быть представлены одним-единственным пятизначным двоичным числом и, соответственно, помещены в одну переменную. В этом случае каждый параметр будет представлен одним битом двоичного числа, представляющего не что иное, как набор флагов, каждый из которых может быть установлен или снят. Это иллюстрируется табл. 6.3.

Таблица 6.3. *Кодирование нескольких параметров одним двоичным числом*

Номер бита	Параметр настройки	Двоичное значение	Десятичное значение
		Включение элемента мультимедиа	
0	Фото	00001	1
1	Анимация	00010	2
2	Аудио	00100	4
3	Видео	01000	8
4	Локальное хранение информации	10000	16

Поскольку для представления чисел используется десятичная система счисления, двоичное число, отражающее выбор пользователя, нужно преобразовать в десятичное, которое в свою очередь, можно сохранить в переменной.

```
userSettings:Number = 31; // Двоичное 11111. Все элементы
```

Переменная `userSettings` содержит значение, указывающее на то, что все мультимедийные элементы должны присутствовать в фильме (все биты включены).

Для того чтобы отключить звук, достаточно сбросить второй бит, что даст двоичное число 11011.

```
userSettings:Number = 27; // Двоичное 11011. Отключить звук
```

Для того чтобы отключить звук и видео одновременно, нужно сбросить второй и третий биты.

```
userSettings:Number = 11; // Двоичное 10011. Отключить видео и звук
```

Далее в этом разделе будут рассмотрены побитовые операции, позволяющие выполнять различные действия с битами чисел.

Логические побитовые операции

Данные операции выполняются с числовыми операторами. Логические побитовые операторы преобразуют свои операнды в двоичные 32-разрядные целые числа без учета знака. Это означает, что преобразование положительного числа и отрицатель-

ного числа дает один и тот же двоичный результат, а знак просто переписывается. Например, преобразование десятичного числа 16 в двоичное дает 10000, а преобразование числа −16 дает −10000.

Побитовое ИЛИ

Побитовый оператор ИЛИ имеет два числовых операнда. Данный оператор анализирует соответствующие биты (то есть биты, имеющие одинаковый порядковый номер) своих операндов и возвращает числовое значение, полученное на основании следующего правила. Если хотя бы один из соответствующих битов исходных операндов установлен, то данный бит в возвращаемом значении также будет установлен. Если у обоих операторов соответствующие биты сброшены, то и в возвращаемом значении такой бит будет сброшен.

Оператор побитового ИЛИ имеет следующий синтаксис:

операнд 1 | *операнд 2*

Здесь *операнд 1* и *операнд 2* — это числовые литералы или идентификаторы, содержащие числовые значения. Если операнды являются дробными числами, дробная часть отбрасывается.

Результат применения данного оператора иллюстрируется табл. 6.4.

Таблица 6.4. *Применение оператора "побитовое ИЛИ"*

Побитовое ИЛИ (|)	**Представление числа**					
	двоичное	**десятичное**	**двоичное**	**десятичное**	**двоичное**	**десятичное**
Операнд 1	00001	1	10101	21	000101	5
Операнд 2	00000	0	01010	10	101101	45
Результат	00001	1	11111	31	101101	45

Операцию побитового ИЛИ часто применяют для объединения нескольких отдельных значений в одно, включающее установленные биты обоих операндов.

Вернемся к рассмотренному ранее примеру с параметрами фильма. Значение 00011 (десятичное 3) указывает на то, что пользователь хочет задействовать только два мультимедийных элемента: фото и анимацию. Значение 11100 (десятичное 24) разрешает применить только звук, видео и допускает локальное хранение информации. Для того чтобы включить сразу все элементы настроек, достаточно применить операцию логического ИЛИ к этим двум значениям:

```
11100|00011 // Возвращает 11111, включены все элементы
```

Побитовое И

Побитовый оператор И имеет два числовых операнда. Данный оператор анализирует соответствующие биты своих операндов и возвращает числовое значение, полученное на основании следующего правила. Если оба соответствующих бита исходных

операндов установлены, то данный бит в возвращаемом значении также будет установлен. Если у обоих операторов соответствующие биты сброшены или у одного операнда бит установлен, а у другого сброшен, то и в возвращаемом значении такой бит будет сброшен.

Оператор побитового И имеет следующий синтаксис:

```
операнд 1 & операнд 2
```

Здесь *операнд 1* и *операнд 2* — это числовые литералы или идентификаторы, содержащие числовые значения. Если операнды являются дробными числами, дробная часть отбрасывается.

Результат применения данного оператора иллюстрируется в табл. 6.5.

Таблица 6.5. *Применение оператора "побитовое И"*

Побитовое И (&)	Представление числа					
	двоичное	десятичное	двоичное	десятичное	двоичное	десятичное
Операнд 1	00001	1	10101	21	000101	5
Операнд 2	00000	0	01010	10	101101	45
Результат	00000	0	00000	0	000101	5

Операция побитового И используется для проверки факта установки одного или нескольких битов операндов.

Так, если переменная `userSettings` содержит некоторое значение, определяющее выбор пользователя, то для того чтобы проверить, установлен ли определенный бит, можно выполнить операцию побитового И, в которой в качестве первого операнда будет выступать переменная `userSettings`, а в качестве второго будет использоваться значение, у которого установлен только проверяемый бит. Если соответствующий бит установлен у значения переменной `userSettings`, это значение будет возвращено, если же этот бит снят, то будет возвращен ноль, так как все остальные биты второго операнда сняты. Это иллюстрируется на следующем примере.

Проверим, выбран ли звук:

```
if (userSettings & 4){
trace("Звук включен");
}
```

Выражение условного предложения (в круглых скобках) преобразуется в булево значение. Если третий бит в `userSettings` установлен, побитовая операция возвращает числовое значение, отличное от нуля, которое преобразуется в булево значение `true` согласно правилам преобразования типов данных. Если третий бит в `userSettings` сброшен, побитовая операция возвращает значение 0, которое преобразуется в `false`.

Поясним это на конкретных числах. Допустим, пользователь включил все элементы, кроме видео:

```
userSettings = 23 // двоичное значение: 10111
```

Для проверки того, включен ли звук (2-ой бит), используется такое выражение:

```
23&4; // Возвращает 23, значит 2-ой бит установлен
```

Или то же самое в двоичном представлении:

```
10111 & 100
```

Теперь проверим, включено ли видео:

```
23&8; // Возвращает 0
```

Или то же самое в двоичном представлении:

```
10111& 1000
```

Здесь нет совпадающих установленных битов, поэтому выражение выдает ноль.

Побитовое исключающее ИЛИ

Этот оператор действует похоже на оператор "побитовое ИЛИ" с одной лишь существенной разницей. Если соответствующие биты у обоих операндов установлены, данный бит в результирующем значении сбрасывается. Таким образом, бит в результирующем значении принимает значение 1 в том случае, если один из них сброшен, а другой установлен. Если оба бита операндов установлены или оба сброшены, в результирующем значении данный бит будет сброшен.

Оператор побитового исключающего ИЛИ имеет следующий синтаксис:

операнд 1 ^ операнд 2

Здесь *операнд 1* и *операнд 2* — это числовые литералы или идентификаторы, содержащие числовые значения. Если операнды являются дробными числами, дробная часть отбрасывается.

Результат применения данного оператора иллюстрируется табл. 6.6.

Таблица 6.6. *Применение оператора "исключающее ИЛИ"*

Побитовое исключающее ИЛИ (^)	Представление числа					
	двоичное	десятичное	двоичное	десятичное	двоичное	десятичное
Операнд 1	00001	1	10101	21	000101	5
Операнд 2	00000	0	01010	10	101101	45
Результат	00001	1	11111	31	101000	40

Оператор исключающего ИЛИ может использоваться для переключения флага, то есть изменения состояния требуемого бита.

Допустим, пользователь выбрал три элемента: фото, анимацию и звук:

```
userSettings = 7; // двоичное значение: 00111
```

Для того чтобы теперь отключить звук, нужно изменить состояние второго бита (сбросить его) и сохранить новое значение в переменной `userSettings`. Для этого можно выполнить следующее действие:

```
userSettings = userSettings ^ 4; /* значение userSettings теперь равно 3
или 00011 в двоичном представленнии */
```

В двоичном представлении это можно записать так:

```
00111 ^ 00100 = 00011
```

Побитовое отрицание (НЕ)

В отличие от предыдущих операторов, оператор побитового отрицания имеет только один операнд, то есть это унарный оператор. Этот оператор инвертирует биты своего операнда. В результате все сброшенные биты устанавливаются, а все установленные биты сбрасываются.

Оператор побитового отрицания имеет следующий синтаксис:

```
операнд 1 ~ операнд 2
```

Здесь *операнд 1* и *операнд 2* — это числовые литералы или идентификаторы, содержащие числовые значения. Если операнды являются дробными числами, дробная часть отбрасывается.

Примечание

Символ ~ называется "тильда". На клавиатуре он находится в левом верхнем углу, слева от цифры 1.

Для того чтобы вычислить результат данной операции, необходимо вначале изменить знак операнда, а затем вычесть из него единицу.

Результат применения данного оператора иллюстрируется ниже.

```
trace (~1); // Выводит -2
trace (~0); // Выводит -1
```

Побитовые составные операторы

Побитовые составные операторы позволяют объединить выполнение побитовой операции и присваивание.

Составные побитовые операции представлены в табл. 6.7.

Таблица 6.7. *Составные побитовые операторы*

Побитовая операция	Оператор
Побитовое ИЛИ с присваиванием	|=
Побитовое И с присваиванием	&=

Таблица 6.7 *(окончание)*

Побитовая операция	Оператор
Побитовое исключающее ИЛИ с присваиванием	^=
Побитовое отрицание с присваиванием	~=
Сдвиг вправо с присваиванием	>>=
Сдвиг вправо без знака с присваиванием	>>>=
Сдвиг влево с присваиванием	<<=

Как и обычные составные операторы, побитовые составные операторы позволяют использовать более компактную форму записи при необходимости модификации состояния битов.

Так, пусть значение переменной `userSetings`, содержащей информацию о выборе пользователя, по умолчанию составляет 31 (двоичное значение 11111), то есть выбраны все элементы.

Если пользователь хочет запретить возможность локального сохранения информации, выполняем следующее действие:

```
userSettings ^= 16 // Значение userSettings теперь 15 или 01111
```

Если после этого пользователь хочет отключить видео и звук, выполняем следующее действие:

```
userSettings &= ~12 // Значение userSettings теперь 3 или 00011
```

Рассмотрим это выражение более подробно. В соответствии с приоритетом операторов (см. табл. 6.1) первой выполняется операция побитового отрицания. Операция ~12 дает значение −13. В связи с особенностями представления отрицательных чисел в двоичном виде это число записывается так: 1 1111 1111 1111 1111 1111 1111 1110 011. Применение операции побитового И (`&`) возвращает результат, в котором установлены только биты, которые были установлены у обоих операндов, то есть второй и третий.

Если после этого пользователь по каким-то причинам захотел вновь включить звук, видео, и к тому же решил все-таки разрешить сохранять информацию у себя на компьютере, мы можем помочь ему в этом следующим образом:

```
userSettings |= 4|8|16; // Значение userSettings теперь 31 или 11111
```

Здесь вначале вычисляется выражение в правой части присваивания. Результатом объединения является число 28 или 11100 в двоичном представлении. После этого значение `userSettings` объединяется со значением 28, что в результате дает 31 или 11111. Все биты снова установлены.

Побитовые операции сдвига

Побитовые операции сдвига осуществляют смещение значащих битов в числе вправо или влево. Это равносильно делению или умножению числа на степень числа 2,

соответствующую количеству бит, на которое осуществляется сдвиг. Остаток, полученный в результате деления, отбрасывается.

Сдвиг вправо

Побитовый сдвиг вправо осуществляет смещение битов числа на заданное количество позиций вправо. При этом биты, которые выталкиваются за пределы числа, отбрасываются, а новые биты, появляющиеся слева, заполняются нулями или единицами в зависимости от значения 31-го бита (то есть от знака числа). Если число положительное, то при сдвиге свободные биты оказываются сброшены, то есть заполняются нулями. Если число отрицательное, свободные биты заполняются единицами.

Оператор побитового сдвига вправо имеет следующий синтаксис:

операнд >> число битов

Здесь *операнд* — это число, в котором выполняется сдвиг битов, заданное как числовой литерал или идентификатор, содержащий числовое значение. Если значение левого операнда дробное, дробная часть отбрасывается.

Правый операнд *число битов* — это число в диапазоне от 0 до 31, указывающее, на сколько битов выполняется сдвиг. Если значение этого операнда дробное, дробная часть отбрасывается.

Применение оператора побитового сдвига вправо эквивалентно вычислению целой части значения, полученного делением числа *операнд* на степень числа 2, заданную значением операнда *число битов*. Рассмотрим несколько примеров:

```
8>>3; // Исходное число 1000. Результат 1 или 0001.
8.33>>3; // Результат 1. Дробная часть отброшена
10>>3; // Результат 1. Целая часть от деления 10 на 2^3 равна 1
-8>>3; // Результат -1 или 11111111111111111111111111111001
```

Рассмотрим последний пример более подробно. Число −8 в двоичном представлении выглядит так:

1 1111 1111 1111 1111 1111 1111 1111 000

Сдвигая биты на 3 позиции вправо, получаем:

1 1111 1111 1111 1111 1111 1111 1111 111

Свободные биты в левой части заполняются единицами, так как 31-й бит установлен, то есть число отрицательное. Полученное значение в десятичном представлении равно −1.

Выделение цветовых составляющих из шестнадцатеричного RGB-значения цвета

Побитовый сдвиг вправо в сочетании с операцией логического И очень удобно использовать для выделения цветовых составляющих из шестнадцатеричного RGB значения цвета. В шестнадцатеричном представлении любой цвет из модели RGB задается шестизначным числом, в котором на каждую цветовую составляющую от-

водится по два разряда. Два правых разряда указывают количество красного, два следующих разряда — количество зеленого и, наконец, два левых разряда — количество синего цвета в результирующем цвете. Двоичное представление цвета — это число, состоящее из 24 битов: по восемь битов на каждую составляющую. Приведем несколько примеров (табл. 6.8).

Таблица 6.8. *Двоичное представление цвета*

Шестнадцатеричное представление цвета	Двоичное представление цвета	Описание цвета
0xFFFFFF	1111 1111 1111 1111 1111 1111	белый
0x000000	0000 0000 0000 0000 0000 0000	черный
0xFF0000	1111 1111 0000 0000 0000 0000	красный
0x00FF00	0000 0000 1111 1111 0000 0000	зеленый
0x0000FF	0000 0000 0000 0000 1111 1111	синий
0xFFFF00	1111 1111 1111 1111 0000 0000	желтый

Для того чтобы выделить нужную цветовую составляющую из двоичного представления цвета, необходимо отбросить биты, соответствующие остальным составляющим.

Для того чтобы выделить красную составляющую, нужно отбросить 16 битов справа. Для выделения зеленой составляющей необходимо отбросить 8 битов слева и справа. Для выделения синей составляющей отбрасываются 16 левых битов.

Выделение цветовой составляющей в общем случае выполняется в два приема:

1. Используя побитовый оператор И, сбрасываем все биты, за исключением битов, отвечающих за выделяемую составляющую.
2. Для выделения красной составляющей выполняем побитовый сдвиг вправо на 16 битов. Для выделения зеленой составляющей выполняем побитовый сдвиг вправо на 8 битов. Для выделения синей составляющей сдвиг не требуется.

Рассмотрим пример:

```
var color:Number = 0xFFCC66; // Исходный цвет
var red = (color&0xFF0000)>>16; // Красная составляющая 255 (FF)
var green = (color&0xFF00)>>8; // Зеленая составляющая 204 (CC)
var blue = color&0xFF // Синяя составляющая 102 (66)
```

Рассмотрим численный пример. Возьмем случайный цвет `color` = F30C2E или то же самое в двоичном представлении:

1111 0011 0000 1100 0010 1110

Выделим красную составляющую:

1111 0011 0000 1100 0010 1110 `&` 1111 1111 0000 0000 0000 0000

Результат этой операции:

1111 0011 0000 0000 0000 00002.

Сдвинем биты:

1111 0011 0000 0000 0000 0000 >> 16

Результат этой операции:

1111 0011

Полученное значение является двоичным представлением шестнадцатеричного числа F3 или десятичного 243. Аналогично можно выделить и остальные составляющие.

Сдвиг вправо без знака

Побитовый сдвиг вправо без знака осуществляет такое же смещение битов числа, как и сдвиг вправо со знаком. Различие заключается в том, что при выполнении сдвига без знака знак числа не сохраняется. Число всегда преобразуется в положительное двоичное число, поскольку освободившиеся в результате сдвига биты слева, включая 31-й бит, заполняются нулями.

Оператор побитового сдвига вправо без знака имеет следующий синтаксис:

операнд >>> число битов

Здесь *операнд* — это число, в котором выполняется сдвиг битов, заданное как числовой литерал или идентификатор, содержащий числовое значение. Если значение левого операнда дробное, дробная часть отбрасывается.

Правый операнд *число битов* — это число в диапазоне от 0 до 31, указывающее, на сколько битов выполняется сдвиг. Если значение этого операнда дробное, дробная часть отбрасывается.

Для положительных чисел результат действия оператора беззнакового сдвига вправо совпадает с действием оператора сдвига вправо. Поэтому рассмотрим пример беззнакового сдвига для отрицательного числа −17. Его двоичное представление имеет следующий вид:

1 1111 1111 1111 1111 1111 1111 1101 111

Выполняем беззнаковый сдвиг вправо на три разряда:

```
-17 >>> 3; //Результат 536870909
```

В результате сдвига без знака три крайние справа бита выталкиваются за пределы числа, а три освободившихся старших бита слева заполняются нулями. Получаем:

0 0011 1111 1111 1111 1111 1111 1111 101 = 536870909

Сдвиг влево

Побитовый сдвиг влево осуществляет смещение битов числа на заданное количество позиций влево. При этом левые биты, которые выталкиваются за пределы числа, отбрасываются, а новые биты, появляющиеся справа, заполняются нулями.

Оператор побитового сдвига влево имеет следующий синтаксис:

операнд << *число битов*

Здесь *операнд* — это число, в котором выполняется сдвиг битов, заданное как числовой литерал или идентификатор, содержащий числовое значение. Если значение левого операнда дробное, дробная часть отбрасывается.

Правый операнд *число битов* — это число в диапазоне от 0 до 31, указывающее, на сколько битов выполняется сдвиг. Если значение этого операнда дробное, дробная часть отбрасывается.

Применение оператора побитового сдвига вправо эквивалентно умножению на степень числа 2, заданную операндом *число битов*. Рассмотрим несколько примеров:

```
1<<3; // Исходное значение 0001. Результат 8 или 1000. 1*2^3 = 8
4<<2; // Исходное значение 00100. Результат 16 или 10000. 4*2^2 = 16
-13<<4; // Результат -208
```

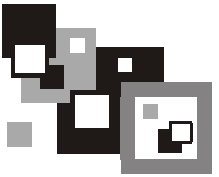
В последнем примере исходное число −13 имеет следующее двоичное представление:

1 1111 1111 1111 1111 1111 1111 1110 011

В результате сдвига битов влево на четыре разряда четыре старших бита слева отбрасываются, а новые биты, добавленные слева, заполняются нулями, что дает следующий результат:

1 1111 1111 1111 1111 1111 1110 0110 000 = −208

Глава 7



Управляющие структуры: Операторы ветвления. Циклы

При любой последовательности вычислений ошибки
начнут выявляться на том конце, который противостоит началу проверки.

Закон ошибок Грельба

Для решения большинства практических задач требуется более сложная и гибкая обработка программного кода, чем просто однократное последовательное выполнение операторов. Зачастую возникает необходимость в выполнении только одного из нескольких фрагментов программного кода или в многократном выполнении одинакового набора команд. Для этих целей необходимо использовать возможности языка, позволяющие организовать управление процессом выполнения кода. Такие возможности предоставляют управляющие структуры. *Управляющей структурой* называется предложение, позволяющее контролировать выполнение блока кода, состоящего из совокупности предложений.

К управляющим структурам ActionScript можно отнести предложения, реализующие ветвление кода, предложения цикла и обработчики событий. В данной главе речь пойдет об операторах ветвления и программных циклах.

Ветвление кода

Операторы ветвления (предложения) позволяют реализовать несколько путей выполнения программного кода. С помощью ветвления можно разбить весь код или фрагмент кода на блоки, каждый из которых выполняется только при выполнении заданного условия. Если условие не выполнено, соответствующий блок кода не обрабатывается. Это дает очень мощные и гибкие возможности управления программой.

Предложения, позволяющие реализовать ветвление кода, могут быть одновариантными, двухвариантными и многовариантными. Одновариантное ветвление имеет место в случае, когда имеется только один блок кода, выполняющийся при определенном условии. Двухвариантное ветвление предполагает наличие двух блоков кода. При выполнении определенного условия выполняется один из блоков, в случае ложности условия выполняется другой блок кода. И, наконец, многовариантное

ветвление предполагает наличие множества блоков кода, каждый из которых выполняется при выполнении заданного для данного блока кода условия и невыполнении условий остальных блоков управляющей структуры.

Предложения ActionScript, реализующие ветвление кода, обобщены в табл. 7.1, расположенной в конце этого раздела.

Предложение *if*

Это предложение называется *условным* и позволяет организовать одновариантное ветвление кода. Условное предложение используется для задания условия, выполнение которого инициирует выполнение блока кода. Если условие не выполняется, данный блок кода игнорируется интерпретатором, который продолжает обработку инструкций, следующих за условным предложением. Алгоритм выполнения предложения `if` представлен на рис. 7.1.

Предложение `if` использует следующий синтаксис:

```
if (условие){
блок кода;
...
}
```

Здесь `if` — ключевое слово.

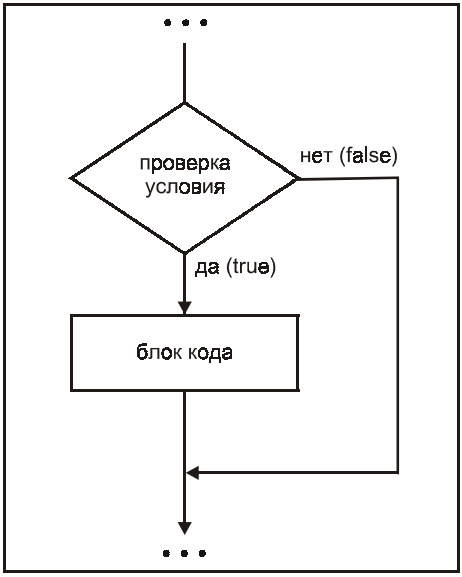


Рис. 7.1. Алгоритм выполнения предложения `if`

В круглых скобках находится проверочное выражение *условие*, задающее условие, выполнение которого приводит к выполнению всех предложений, помещенных в фигурные скобки. Проверочное выражение может включать данные любого типа, однако результатом его вычисления должно быть булево значение `true` или `false`. Если результатом вычисления выражения *условие* является значение иного типа, оно автоматически приводится к булеву типу в соответствии с правилами, описанными в *главе 5*. Зачастую в качестве проверочных выражений используются операции отношений, часто выступающие операндами логических операторов, позволяющих объединить проверку истинности сразу нескольких отношений. Проверочное выражение обязательно заключается в круглые скобки.

В фигурные скобки заключен *блок кода*, который выполняется *только* в случае истинности проверочного выражения *условие*. Блок кода в фигурных скобках может включать в себя неограниченное количество других предложений. Каждое из этих предложений записывается с новой строки и заканчивается точкой с запятой согласно стандартным правилам синтаксиса предложений.

Важно отметить, что после обработки условного предложения, интерпретатор переходит к инструкциям, расположенным после него вне зависимости от того, выполняется ли проверочное выражение или нет.

Обратите внимание на фигурные скобки, ограничивающие блок условного предложения. Их наличие является обязательным, если *блок кода* включает более одного предложения, поскольку скобки ограничивают блок условного предложения от внешних предложений программного кода. Что касается места их размещения, то в принципе компоновка может быть различной. Можно переносить открывающую фигурную скобку на следующую строку и здесь либо размещать ее в одной строке с первым предложением внутреннего блока, либо начинать это предложение на следующей строке. Закрывающая скобка может размещаться либо в конце строки последнего внутреннего предложения, либо переносится на следующую строку (как это показано в образце синтаксиса). В принципе можно поместить всю конструкцию даже в одной строке, внимательно расставляя фигурные скобки и точку с запятой после каждого предложения блока. Тем не менее представленная запись является наиболее ясной и читаемой, поэтому в дальнейшем мы будем придерживаться именно такой формы.

Рассмотрим простой пример использования условного предложения:

```
var flag:Boolean = true;
if(flag){
trace ("Выполняется блок условного предложения");
}
```

В результате выполнения этого сценария в панель **Output** выводится сообщение `"Выполняется блок условного предложения"`, то есть инструкция в блоке условного предложения выполняется. Это происходит в связи с тем, что условное выражение содержит булево значение `true`, заданное с помощью переменной `flag`.

Теперь рассмотрим следующую ситуацию. Допустим, нам необходимо увеличивать текущее значение некой переменной `counter` на единицу при каждом выполнении сценария. Для этого удобно использовать операцию инкрементирования (`++`). Про-

блема состоит в том, что на момент выполнения модификации значение этой переменной еще может быть не определено. В этом случае перед выполнением модификации переменную нужно инициализировать, то есть создать ее и присвоить ей исходное значение. Если перед модификацией явно присвоить переменной фиксированное значение, оно не будет обновляться при каждом выполнении сценария. Если же ничего не предпринимать, то инкрементирование переменной со значением `undefined` выдаст значение `NaN`, что приведет к проблемам с дальнейшим использованием этой переменной. Поэтому в данной ситуации удобно прибегнуть к условному выражению и проверить, если у переменной нет значения, присвоить его, а если значение есть, то можно выполнить его инкрементирование.

```
if(counter==undefined){
        //counter = 0; // Если значение не определено присваиваем 0
}
counter ++; // Инкрементируем значение counter
trace (counter); // Выводит 1
```

Теперь при первом выполнении приведенного кода переменная `counter` будет инициализирована, а при каждом следующем выполнении ее текущее значение будет увеличиваться на единицу. Блок условного предложения при этом выполняться не будет, поскольку проверочное выражение будет разрешаться в `false`. Для того чтобы в результате многократного выполнения данного кода значение переменной `counter` не достигло бы слишком большой величины, его можно ограничить (например, значением 10), добавив к уже имеющемуся коду следующие строки:

```
if (counter >= 10) {
        counter = 0;
}
```

Многократное выполнение приведенного кода будет заставлять переменную `counter` изменять свое значение от 0 до 10 и выводить значения 1, 2, ..., 10 в панель **Output**.

При необходимости проверки сразу нескольких условий в проверочном предложении можно использовать логические операторы. Так, например, если блок кода должен быть выполнен при условии попадания некоего значения `x` в диапазон от 0 до 10, можно сформировать такое условное предложение:

```
if ((x >= 0) && (x <= 10)) {
        trace("x лежит на отрезке [0, 10]");
}
```

Если инструкции условного предложения должны быть выполнены при условии, что `x` не попадает в данный диапазон, вместо логического И используется логическое ИЛИ:

```
if ((x < 0) || (x > 10)) {
        trace("x не лежит на отрезке [0, 10]");
}
```

Предложение *else*

Предложение `else` всегда используется в сочетании с предложением `if` и позволяет реализовать двухвариантное ветвление. С помощью этого предложения задается блок кода, который выполняется только в случае, если проверочное выражение условного предложения оказывается ложно. Таким образом, предложение `else` содержит альтернативный блок кода, определяющий, какие действия необходимо произвести, если блок условного предложения не обрабатывается. Блоки предложений `if` и `else` никогда не выполняются одновременно. Такая конструкция используется для выбора одного из двух возможных путей выполнения программы.

Предложение `else` использует следующий синтаксис:

```
if (условие){
блок кода;
...
}
else {
альтернативный блок кода;
...
}
```

Семантика (то есть смысл) этой конструкции заключается в том, что, если проверочное выражение *условие* оказывается истинно, выполняется только *блок кода*, а *альтернативный блок кода* игнорируется. Если проверочное выражение *условие* оказывается ложно, *блок кода* игнорируется и выполняется только *альтернативный блок кода*. Алгоритм выполнения предложения `else` представлен на рис. 7.2.

Вопрос взаимного размещения фигурных скобок, ключевого слова `else` и предложений альтернативного блока кода решается точно так же, как и с предложением `if`. Приведенная форма записи является наиболее читаемой и будет использоваться в дальнейшем. Тем не менее при желании можно задать параметры автоматического форматирования (**Auto Format Options...**), позволяющие автоматически реализовать иную компоновку *(см. разд. "Использование редактора ActionScript для создания сценариев" главы 3)*.

Следующий пример иллюстрирует принцип работы предложения `else`.

```
var flag:Boolean = false;
if(flag){
       trace("Выполняется блок условного предложения");
}
else{
       trace("Выполняется блок предложения else");
}
```

В результате выполнения этого сценария в панель **Output** выводится сообщение `"Выполняется блок предложения else"`, поскольку проверочное выражение условного предложения содержит значение `false`.

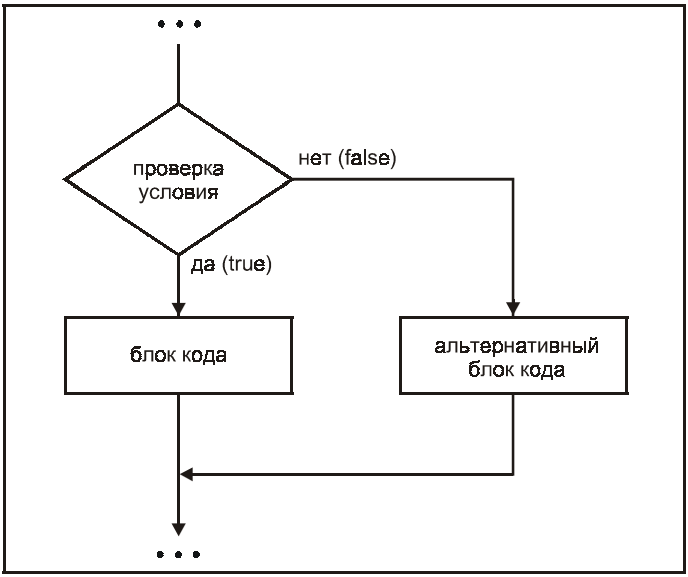


Рис. 7.2. Алгоритм выполнения предложения `else`

Рассмотрим пример кода, позволяющего определить, является число целым или дробным. Для этого вначале конвертируем число в строку и затем с помощью метода `String.indexOf()` проверим, содержит ли эта строка символ (.). Напомним, что если искомый символ в строке отсутствует, метод `String.indexOf()` возвращает значение −1.

```
var num:Number = 3.1415926; /* Задаем число, дробность которого нужно
установить */
if(String(num).indexOf(".")!=-1){
        trace("Число дробное");
}
else {
        trace("Число целое");
}
```

В результате выполнения этого сценария в панель выводится сообщение "`Число дробное`", так как выполняется блок кода условного предложения.

С помощью предложения `else` можно реализовать переключатель, контролирующий два состояния объекта или процесса и выполняющий определенные действия в зависимости от его текущего состояния. Это очень часто встречающаяся на практике задача. Наиболее характерной ее иллюстрацией служит кнопка-флаг, последовательное нажатие которой приводит к последовательному выполнению двух разных действий, например, запуска и остановки анимации. Листинг 7.1 содержит сценарий

кнопки, нажатие которой останавливает воспроизведение монтажной линейки, а повторное нажатие вновь запускает воспроизведение.

Допустим, что анимация запущена по умолчанию, таким образом, первое нажатие кнопки должно ее останавливать. Для контроля состояния воспроизведения введем переменную-флаг, значение которой будет изменяться в зависимости от состояния воспроизведения, и назовем ее `flag`. Условимся, что, если анимация запущена, `flag` принимает значение `false`, а если остановлена, то `true`. Сценарий, реализующий функциональность кнопки, прикрепим непосредственно к ее экземпляру. Мы намеренно не стали заранее явно объявлять переменную `flag`, поэтому, когда интерпретатор впервые встретит ссылку на эту переменную, он автоматически создаст ее, присвоив в качестве значения `undefined`. При самом первом нажатии переменная-флаг еще не создана, поэтому использование ее идентификатора в качестве проверочного выражения даст значение `false` (так как при преобразовании значения `undefined` в булев тип результатом является `false`). Значит, будет выполнен блок предложения `else`, в котором осуществляются остановка монтажной линейки и переключение флага. В связи с этим при следующем нажатии уже будет выполнен блок предложения `if`, где анимация будет запущена, и снова произойдет переключение флага.

Листинг 7.1. Кнопка-флаг

```
on(press){
if(flag){
       play(); // Запускаем воспроизведение
       flag = false; // Переключаем флаг
       }
else{
       stop(); // Останавливаем воспроизведение
       flag = true; // Переключаем флаг
       }
}
```

Сокращенная запись

Если блоки кода предложений `if` и `else` состоят только из одного предложения, то фигурные скобки могут быть опущены. В этом случае предложение блока записывается на той же строке, что и ключевое слово. Следующая запись эквивалентна только что приведенной:

```
if (flag) trace("Выполняется блок условного предложения");
else trace("Выполняется блок предложения else");
```

Такая форма записи называется *сокращенной*. Преимущество сокращенной записи заключается в ее компактности, недостаток — в снижении читаемости кода. Сокращенную запись имеет смысл использовать, только если блоки кода предложений `if` и `else` состоят из несложных коротких выражений.

Условный оператор

Конструкция, состоящая из предложений `if` и `else`, может быть записана с помощью оператора, называемого *условным* (conditional). Условный оператор может быть использован, если блоки кода предложений `if` и `else` состоят только из одного предложения. Он дает возможность присваивать переменной одно из двух значений в зависимости от выполнения определенного условия, а также применяется для более компактной формы записи.

Условный оператор имеет следующий синтаксис:

```
условие ? предложение : альтернативное предложение
```

Здесь *условие* — это операнд, определяющий проверочное выражение. Результат вычисления проверочного выражения преобразуется в булев тип. Обычно проверочное выражение включает операторы отношений и логические операторы, например: `age >=14 || age <21`.

Операнд *предложение* — это выражение, содержащее данные любого типа или предложение, инициирующее выполнение определенных действий. Операнд *предложение* обрабатывается только в том случае, если проверочное выражение оказывается истинно, то есть выражение *условие* дает значение `true`.

Операнд *альтернативное предложение* — это выражение, содержащее данные любого типа, или предложение, инициирующее выполнение определенных действий. Операнд *альтернативное предложение* обрабатывается только в том случае, если проверочное выражение оказывается ложно, то есть выражение *условие* дает значение `false`.

Рассмотрим простой пример:

```
x == undefined ? trace("Значение не определено") : trace(x);
```

В данном случае условный оператор используется для проверки того, определено ли значение переменной `x`. Если значение не определено, выводится сообщение `"Значение не определено"`, если `x` имеет конкретное значение, оно выводится в панель **Output**. Вот как выглядит эта проверка, реализованная с помощью предложений `if` и `else`.

```
if (x == undefined) {
        trace("Значение не определено");
}
else {
        trace(x);
}
```

С помощью условного оператора удобно осуществлять присваивание переменной значения, зависящего от выполнения определенного условия. Так, в следующем примере переменная `access` принимает значение `true`, если значение переменной `userPass` совпадает со значением переменной `correctPass`, и значение `false` в противном случае:

```
var correctPass:String = "actionscript";
var userPass:String = "javascript";
```

```
var access:Boolean = (userPass == correctPass) ? true : false;
trace(access); // Выводит false
```

Предложение *else if*

Предложение `else if` всегда используется в сочетании с предложением `if` и позволяет организовать многовариантное ветвление кода. Как видно из названия, это предложение объединяет в себе свойства предложений `if` и `else`. Предложение `else if` включает проверочное выражение и блок кода, который выполняется только в случае истинности проверочного выражения. Данное предложение позволяет реализовать несколько вариантов путей выполнения программы, которые могут быть исполнены только в случае ложности проверочного выражения предложения `if` и при условии истинности проверочного выражения предложения `else if`. Из всей цепочки предложений `else if` может быть выполнено не более одного блока кода.

Предложение `else if` использует следующий синтаксис:

```
if (условие 1){
блок кода 1;
...
}
else if (условие 2){
блок кода 2;
...
}
else if (условие 3){
блок кода 3;
...
}
...
else if(условие N){
блок кода N;
...
}
else {
альтернативный блок кода;
...
}
```

Семантика приведенной конструкции заключается в следующем.

Если *условие 1* оказывается истинным, выполняется *блок кода 1*, а все остальные предложения этой конструкции игнорируются.

Если *условие 1* оказывается ложным, *блок кода 1* игнорируется и выполняется переход к первому предложению `else if`.

Если *условие 2* оказывается истинным, выполняется *блок кода 2*, а все остальные предложения этой конструкции игнорируются.

Если *условие 2* оказывается ложным, *блок кода 2* игнорируется и выполняется переход к следующему предложению `else if`.

Таким образом, выполняется последовательная обработка проверочных выражений предложений `else if`, которая прекращается в одном из следующих случаев:

- если очередное проверочное выражение окажется истинным. В этом случае будет выполнен соответствующий блок кода, а остальные предложения цепочки будут проигнорированы;
- если предложения `else if` будут исчерпаны, то есть ни одно из проверочных выражений не окажется истинным.

Если все проверочные выражения оказываются ложными, происходит переход к предложению `else` и выполнение *альтернативного блока кода*. Этот алгоритм представлен на рис. 7.3.

Количество предложений `else if` в общем случае не ограничено и определяется исключительно числом вариантов выполнения программы, которые требуется предусмотреть.

Предложение `else`, завершающее эту цепочку, определяет набор операций, которые выполняются, если все проверочные выражения оказываются ложными. Если необходимости в таких действиях нет, то предложение `else` просто опускается.

Рассмотрим простой пример использования предложения `else if`. В приведенном ниже сценарии выполняется выбор средства передвижения (переменная `vehicle`) в зависимости от располагаемой суммы, значение которой представлено переменной `income`.

```
if (income >= 1500) {
       vehicle = "Самолет";
}
else if (income >= 1000) { // сумма от 1000 до 1500
       vehicle = "Теплоход";
}
else if (income >= 500) { // сумма от 500 до 1000
       vehicle = "Железная дорога";
}
else if (imcome >= 250) { // сумма от 250 до 500
       vehicle = "Автобус";
}
else { // иное значение
       vehicle = "Автостоп";
}
```

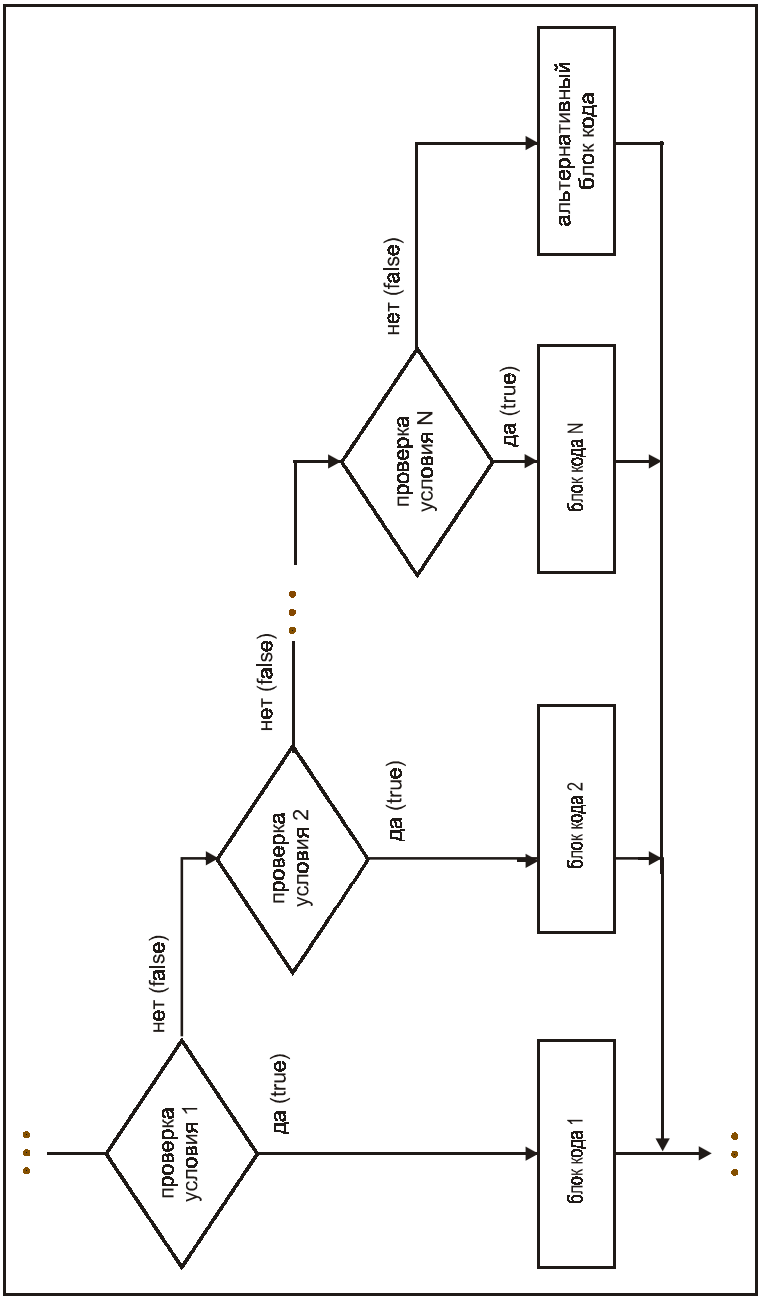


Рис. 7.3. Алгоритм выполнения цепочки предложений `else if`

Листинг 7.2 содержит более сложный сценарий, сравнивающий текущую дату с заданной датой истечения испытательного срока. Если дата истечения еще не наступила, испытательный срок продолжается, в противном случае выводится сообщение об истечении срока. Здесь каждое последующее предложение цепочки условных предложений выполняет проверку факта истечения менее длительных промежутков времени (год, месяц, день). В проверочных выражениях предложений `else if` используется несколько операторов отношений, поскольку при переходе к каждому следующему предложению необходимо проверять равенство более длинных отрезков времени. Так, проверка истечения месяца испытательного срока должна возвращать `true`, только если истек и месяц и год (поскольку испытательный срок может продолжаться больше года). В приведенном сценарии используется встроенный класс `Date`, о котором будет подробно рассказано в *главе 16*.

Листинг 7.2. Проверка окончания испытательного срока

```
var date:Date=new Date();
//Сохраняем текущую дату
var day = date.getDate();//День
var month = date.getMonth()+1;//Месяц (начиная с 1, а не с 0)
var year = date.getFullYear();//Год
//Задаем дату окончания испытательного срока
var expiryDay = 13;//День
var expiryMonth = 4;//Месяц
var expiryYear = 2003;//Год
//Сообщение, выводимое при окончании испытательного срока
var expiryMessage = "Испытательный срок истек " + expiryDay + "." +↵
expiryMonth + "." + expiryYear
//Вывод текущей даты
trace("Сегодня "+day+"."+month+"."+year);
//Проверка окончания испытательного срока
if(year>expiryYear){
       trace(expiryMessage);
}
else if (month>expiryMonth&&year==expiryYear){
       trace(expiryMessage);
}
else if (day>expiryDay&&month==expiryMonth&&year==expiryYear){
       trace(expiryMessage);
}
else trace("Испытательный срок продолжается...");
```

Предложение *switch*

Предложение `switch`, как и предложение `else if`, применяется для организации многовариантного ветвления. Применение предложения `switch` эквивалентно использованию нескольких последовательно расположенных предложений `else if`. Различие состоит в том, что при использовании предложения `else if` всегда выполняется только один блок кода, соответствующий определенному варианту. Предложение `switch` позволяет выполнить все блоки, расположенные после проверочного выражения, которое является истинным. Однако такая особенность предложения `switch` не является непременной. При необходимости предложение `switch` может выполнять в точности те же функции, что и группа предложений `else if`.

Предложение `switch` имеет следующий синтаксис:

```
switch (проверочное выражение){
case выражение 1:
       блок кода 1;
       ...
       break;
case выражение 2:
       блок кода 2;
       ...
       break;
...
case выражение N:
       блок кода N;
       ...
       break;
default:
       альтернативный блок кода;
       ...
}
```

Предложение `switch` включает в свой состав следующие элементы.

- В круглых скобках предложения `switch` находится *проверочное выражение*, которое в результате вычисления может давать любое значение.
- Предложение `case` задает проверяемое выражение *выражение*, значение которого сравнивается со значением проверочного выражения, которое заключено в круглые скобки предложения `switch`. Ключевое слово `case` отделяется от проверяемого выражения с помощью двоеточия.
- Блок кода, размещенный в каждом предложении `case`, выполняется только в том случае, если значение выражения *проверочное выражение* строго равно значению проверяемого выражения *выражение* предложения `case`, то есть если справедливо отношение *проверочное выражение* `===` *выражение*.

- Предложение `break` прекращает выполнение всех следующих после него предложений `case` и осуществляет переход к предложениям, следующим за предложением `switch` (то есть за пределы конструкции `switch`). Использование предложения `break` приводит к тому, что в предложении `switch` выполняется не более одного блока кода.
- Предложение `default` выполняет в составе предложения `switch` ту же функцию, что и предложение `else`. С его помощью задается блок кода, который выполняется, если значение ни одного из проверяемых `case`-выражений не находится в отношении строгого равенства со значением проверочного `switch`-выражения. После ключевого слова `default` ставится двоеточие.

Семантика предложения `switch` заключается в следующем.

В круглых скобках предложения `switch` задается некоторое проверочное выражение. Предложения `case` определяют различные варианты, которые может принять значение проверочного предложения. Эти значения задаются в каждом предложении `case` после двоеточия.

При обработке предложения `switch` производится последовательная проверка строгого равенства проверочного выражения и `case`-выражений. Напомним, что строгое равенство предполагает не только равенство значений, но и типов данных проверяемых операндов.

Если значение одного из `case`-выражений оказывается строго равно значению проверочного выражения, то выполняется блок кода соответствующего предложения `case`. Использование предложения `break` приводит к тому, что все следующие после него предложения и блоки кода игнорируются. Если предложения `break` не обнаружено, то будут выполнены все остальные блоки кода предложения `switch` (включая блок предложения `default`) вплоть до обнаружения предложения `break` или до тех пор, пока все предложения не будут выполнены.

Алгоритм выполнения предложения `switch` представлен на рис. 7.4.

Если ни одно из выражений предложений `case` не равно (строго) проверочному выражению, выполняется блок предложения `default`.

Рассмотрим пример. Допустим, имеются числовые результаты тестирования. Необходимо на основании количества набранных баллов вывести текстовое сообщение о результате:

```
switch (result) {
case 5 :
       trace("Отлично");
       break;
case 4 :
       trace("Хорошо");
       break;
case 3 :
       trace("Удовлетворительно");
       break;
default :
       trace("Неудовлетворительно");
}
```

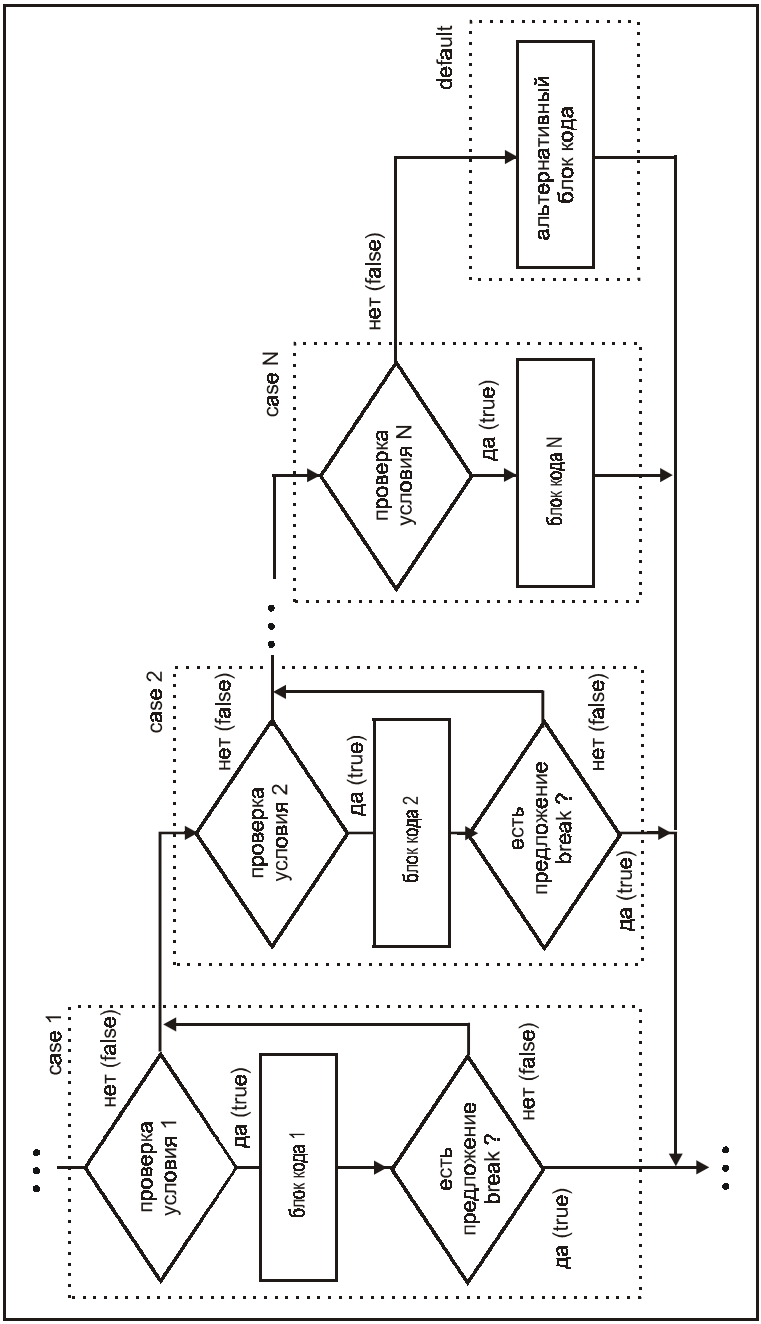


Рис. 7.4. Алгоритм выполнения предложения `switch`

Так, если значение переменной result будет равно 4, то данный код выведет в панель **Output** строку "Хорошо".

Для того чтобы продемонстрировать назначение предложения break, внесем изменения в приведенный выше код:

```
var result:Number = 4;
switch (result) {
case 5 :
       trace("Отлично");
       break;
case 4 :
       trace("Хорошо");
case 3 :
       trace("Удовлетворительно");
default :
       trace("Неудовлетворительно");
}
```

При выполнении этого сценария в панель **Output** будет выведено следующее:

```
Хорошо
Удовлетворительно
Неудовлетворительно
```

В результате удаления предложений break выполнились все блоки кода, начиная с предложения case, выражение которого оказалось равно проверочному (то есть начиная со второго по счету предложения case). Таким образом, при отсутствии предложения break все блоки, следующие за выполняемым блоком, тоже будут выполнены.

Таблица 7.1. *Условные предложения в ActionScript*

Тип предложения	Синтаксис	Описание

if	if (*условие*) {
	блок кода;
	}
		Одновариантное ветвление. Блок кода условного предложения выполняется только в случае истинности условия

else	if (*условие*) {
	блок кода 1;
	}
	else {
	блок кода 2;
	}
		Двухвариантное ветвление. Если условие предложения if оказывается истинно, выполняется его блок кода, в противном случае выполняется блок кода предложения else

***Таблица 7.1** (окончание)*

Тип предложения	Синтаксис	Описание

else if	if (*условие1*) {
	блок кода 1;
	}
	else if (*условие 2*){
	блок кода2;
	}
	...
		Многовариантное ветвление. Выполняется только один из блоков кода при условии истинности проверочного выражения, содержащего его предложения

switch	switch (*проверочное выражение*){
	case *выражение1*:
	блок кода1;
	break;
	case *выражение2*:
	блок кода2;
	break;
	...
	default:
	блок кода;
	}
		Многовариантное ветвление. Блок кода в составе предложения case выполняется только в том случае, если значение его выражения строго равно значению проверочного выражения. Если таких выражений нет, выполняется блок кода предложения default. Если блок кода не содержит предложения break, то в случае его выполнения будут также выполнены все блоки остальных case-предложений вплоть до появления предложения break

Многократное выполнение кода

Для решения очень многих практических задач необходимо обеспечивать возможность многократного выполнения одинакового набора инструкций. Такая ситуация возникает, например, при необходимости задать местоположение нескольким клипам на сцене или при обработке элементов массива. Использование индивидуальных инструкций для обработки каждого элемента данных приведет к непомерному разрастанию программного кода и снижению его эффективности, не говоря о том, что далеко не всегда такая ручная обработка набора элементов представляется возможной. Зачастую приходится иметь дело с ситуацией, когда количество обрабатываемых элементов заранее неизвестно или это количество может изменяться в зависимости от каких-то внешних факторов.

Предложения цикла позволяют гибко и эффективно организовать повторение выполнения сегмента кода и обрабатывать несколько элементов. Предложения цикла, существующие в ActionScript, представлены в обобщенном виде в табл. 7.2, расположенной в конце этого раздела.

Понятие программного цикла

Цикл (loop) — это итеративная программная конструкция, задаваемая специальным предложением, которая вынуждает определенный сегмент кода выполняться требуемое количество раз. Цикл состоит из *заголовка* и *тела*. В заголовке цикла с помощью соответствующего ключевого слова указывается его тип (например, `while` или `for`) и параметры его выполнения. Тело цикла включает блок кода, инструкции которого могут выполняться многократно в зависимости от заданных параметров цикла. ActionScript включает несколько различных типов цикла. Параметры, задаваемые в заголовке, различаются в зависимости от типа цикла, однако в общем случае цикл имеет *счетчик* или итератор и *проверочное выражение*. Принцип действия цикла заключается в том, что блок кода в теле цикла повторно выполняется до тех пор, пока проверочное выражение цикла остается истинным. Каждое выполнение инструкций тела цикла называется *итерацией* (повторением). После каждой итерации происходит модификация значения счетчика цикла. Обычно счетчик используется в проверочном выражении цикла, что дает возможность контролировать количество итераций. Счетчик также можно использовать внутри тела цикла для генерирования различных значений на каждой итерации. Когда после очередной итерации проверочное выражение оказывается ложным, выполнение блока кода в теле цикла прекращается и происходит *выход из цикла*, то есть остановка его работы. Если проверочное выражение всегда остается истинным, то имеет место так называемый бесконечный цикл. Бесконечный цикл всегда вызывает ошибку. Любой цикл, выполняющийся более 15 секунд, считается бесконечным. Его обнаружение приводит к появлению сообщения об ошибке с предложением прекратить действие всех сценариев фильма (см. рис. 7.5).

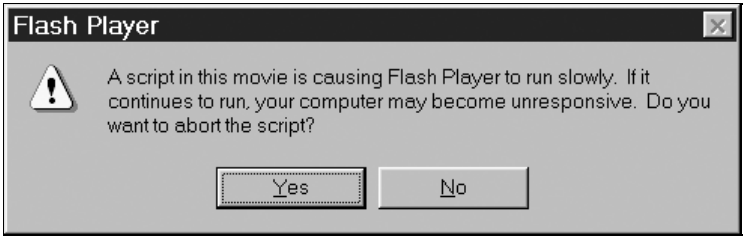


Рис. 7.5. Сообщение об обнаружении бесконечного цикла

Очень важно хорошо понимать, что программный цикл является частью сценария, и сам по себе не может вызывать обновление графического содержимого сцены. Визуальные (и звуковые) изменения, связанные с деятельностью программных циклов, отображаются только после выполнения всех итераций. В связи с этим программные циклы сами по себе не могут выполнять действий, связанных с постепенным визуальным изменением тех или иных параметров объектов. Так, например, здравый смысл подсказывает, что, если в цикле на каждой итерации изменять значение координат объекта, то при выполнении сценария этот объект будет перемещаться. В действительности на протяжении времени выполнения цикла объект будет оставаться неподвижным, а после выполнения последней итерации он переместится в точку,

координаты которой будут получены на последней итерации. Программный цикл выполняется сравнительно короткое время (15-секундный цикл уже считается бесконечным и выдает предупреждающее сообщение), и визуальные результаты его работы отображаются на экране только после завершения работы цикла. В связи с этим программные циклы используются при необходимости обработки нескольких объектов или элементов, не требующей вывода графических (или звуковых) изменений после каждой итерации.

В этой части главы будут рассмотрены различные циклы и принципы их использования.

Цикл *while*

Цикл `while` позволяет реализовать многократное выполнение блока кода до тех пор, пока остается истинным проверочное выражение цикла. Цикл `while` имеет следующий синтаксис:

```
while (проверочное выражение){
блок кода;
...
}
```

Здесь `while` — это ключевое слово.

В круглых скобках задается выражение *проверочное выражение*, которое анализируется перед каждой итерацией цикла.

В теле цикла, заключенном в фигурные скобки, находится блок кода, который может включать в себя неограниченное количество предложений.

Алгоритм выполнения цикла `while` представлен на рис. 7.6 и состоит в следующем:

1. Выполняется оценка проверочного выражения в круглых скобках. Проверочное выражение вычисляется, и результат преобразуется в булев тип.
2. Если в результате оценки проверочного выражения получается значение `true`, выполняются действия, описанные на шаге 3.

   Если в результате оценки проверочного выражения получается значение `false`, происходит выход из цикла и переход к тем предложениям, которые следуют за предложением цикла.
3. Выполняются предложения блока кода в теле цикла.
4. Переход к шагу 1.

Рассмотрим простой пример:

```
var i:Number = 0;
while (i<10) {
        trace(++i);
}
```

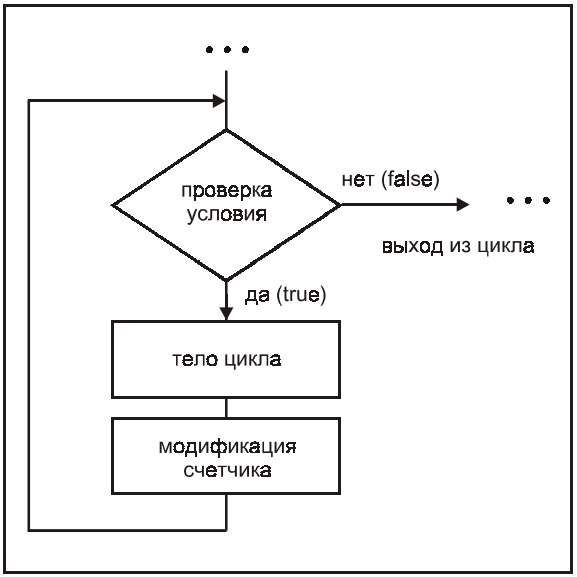


Рис. 7.6. Алгоритм выполнения цикла `while`

В результате выполнения приведенного кода в панель **Output** будет выведен ряд чисел: 1, 2, ..., 10. В этом примере следует обратить внимание на следующие детали. Во-первых, поскольку цикл `while` не предполагает явной инициализации счетчика в заголовке, мы инициализировали переменную `i` перед заголовком цикла.

Во-вторых, в соответствии с проверочным выражением тело цикла будет выполняться до тех пор, пока значение переменной `i` не достигнет 10. В связи с этим модификация счетчика цикла осуществляется непосредственно внутри тела цикла. В данном случае для этого используется оператор инкрементирования. При каждом вызове функции `trace()` значение переменной `i` увеличивается на единицу и выводится в панель **Output**. Поскольку в данном случае используется префиксное инкрементирование, первое выводимое значение — это единица, а не ноль. После очередного выполнения блока тела цикла значение переменной `i` становится равным 10 и очередная оценка проверочного выражения дает значение `false`, после чего осуществляется выход из цикла.

В-третьих, если бы значение переменной не модифицировалось, то проверочное выражение всегда оставалось истинным, и мы получили бы *бесконечный цикл*.

Обратите внимание, что в приведенном примере счетчик используется и в проверочном выражении, и в теле цикла, позволяя нам выводить разные значения на каждой итерации. Такая ситуация встречается очень часто.

Модификация счетчика может выполняться и в заголовке цикла `while` непосредственно в проверочном выражении. В этом случае изменение значения счетчика происходит всякий раз при оценке проверочного выражения.

Следующий код дает такой же результат, как и код, приведенный ранее:

```
var i:Number = 0;
while (i++<10) {
        trace(i);
}
```

Различие между этими двумя вариантами состоит в том, что после выполнения первого сценария значение переменной `i` остается равным 10, а после выполнения второго сценария значение переменной `i` остается равным 11. Это связано с различием префиксного и постфиксного инкрементирования.

Цикл *do while*

Цикл `do while` очень похож на цикл `while` с той лишь разницей, что проверочное выражение находится не перед телом цикла, а после него. Вследствие этого блок кода в теле цикла выполняется, как минимум, один раз.

Цикл `do while` имеет следующий синтаксис:

```
do {
блок кода;
...
} while (проверочное выражение)
```

Здесь `do` и `while` — ключевые слова.

В теле цикла, заключенном в фигурные скобки, находится блок кода, который может включать в себя неограниченное количество предложений.

В круглых скобках задается выражение *проверочное выражение*, которое анализируется после каждой итерации цикла.

Алгоритм выполнения цикла `do while` представлен на рис. 7.7 и состоит в следующем:

1. Выполняются предложения блока кода в теле цикла.
2. Выполняется оценка проверочного выражения в круглых скобках. Проверочное выражение вычисляется, и результат преобразуется в булев тип.
3. Если в результате оценки проверочного выражения получается значение `true`, осуществляется переход к шагу 1.

   Если в результате оценки проверочного выражения получается значение `false`, происходит выход из цикла и переход к тем предложениям, которые следуют за предложением цикла.

Поскольку блок кода в теле цикла `do while` выполняется до оценки проверочного выражения, это позволяет однократно исполнить определенные действия, несмотря на то, что результатом оценки проверочного выражения является `false`. В предыдущем примере с циклом `while` переменную, используемую в качестве счетчика, необходимо было инициализировать до выполнения цикла. В противном случае проверочное выражение сразу бы выдало `false` и произошел бы выход из цикла.

Цикл `do while` позволяет инициализировать счетчик непосредственно в теле цикла. Модифицируем рассмотренный ранее пример:

```
do {
       i==undefined ? i=1 : i++;
       trace(i);
} while (i < 10);
```

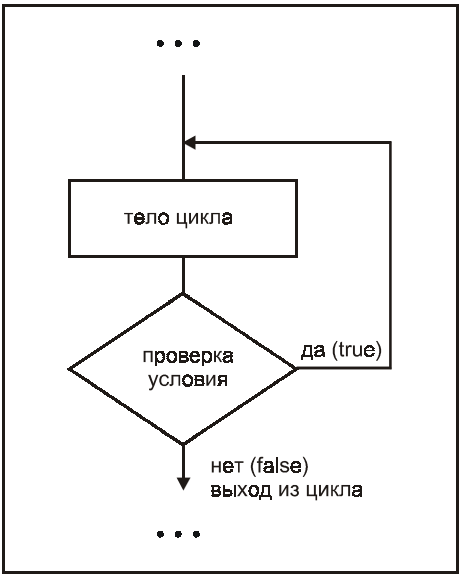


Рис. 7.7. Алгоритм выполнения цикла `do while`

Поскольку переменная счетчика не была объявлена, в теле цикла осуществляется проверка того, имеется у нее значение или нет. Для проверки используется условный оператор. Если значение переменной `i` не определено (при первом выполнении цикла ее значение равно `undefined`), присваиваем ей начальное значение 1. Далее, когда переменная `i` уже имеет конкретное значение, при каждой следующей итерации оно будет инкрементироваться. В результате выполнения этого сценария в панель **Output** будет выведен ряд чисел: 1, 2, ..., 10.

Цикл *for*

Цикл `for`, как и цикл `while`, позволяет многократно выполнять блок кода, размещенный в теле цикла, до тех пор, пока проверочное выражение является истинным. Отличием цикла `for` является то, что его заголовок содержит инициализацию счетчика цикла, проверочное выражение и предложение, модифицирующее счетчик.

Цикл `for` имеет следующий синтаксис:

```
for (инициализация; проверочное выражение; модификация){
блок кода;
...
}
```

Здесь:

- `for` — ключевое слово;
- *инициализация* — это выражение, содержащее объявление переменной, используемой в качестве счетчика цикла. Эта переменная может быть объявлена явно или неявно, то есть с помощью предложения `var` или без него. Инициализирующее выражение выполняется только один раз в самом начале работы цикла перед оценкой проверочного выражения и обычно содержит присваивание, задающее начальное значение счетчика;
- *проверочное выражение* — задает условие, которое оценивается перед каждой итерацией. Если проверочное выражение дает значение `true`, выполняется блок кода внутри тела цикла. Если проверочное выражение дает значение `false`, осуществляется выход из цикла;
- *модификация* — это выражение, модифицирующее счетчик после каждой итерации цикла. Это выражение обычно включает оператор инкрементирования (декрементирования) или составной оператор. Модификация счетчика выполняется после каждой итерации.

Алгоритм выполнения цикла `for` представлен на рис. 7.8 и состоит в следующем:

1. Выполняется инициализация счетчика (выражение *инициализация*).
2. Выполняется оценка проверочного выражения. Проверочное выражение вычисляется, и результат преобразуется в булев тип.

   Если в результате оценки проверочного выражения получается значение `true`, выполняются действия, описанные на шаге 4.

   Если в результате оценки проверочного выражения получается значение `false`, происходит выход из цикла и переход к тем предложениям, которые следуют за предложением цикла.
3. Выполняются предложения блока кода в теле цикла.
4. Выполняется модификация значения счетчика (выражение *модификация*).
5. Переход к шагу 2.

Реализуем рассмотренный ранее пример с использованием цикла `for`.

```
for(var i = 1 ; i<=10; i++){
trace(i);
}
```

Как и в предыдущих случаях, в результате выполнения этого сценария в панель **Output** будет выведен ряд чисел: 1, 2, ..., 10.

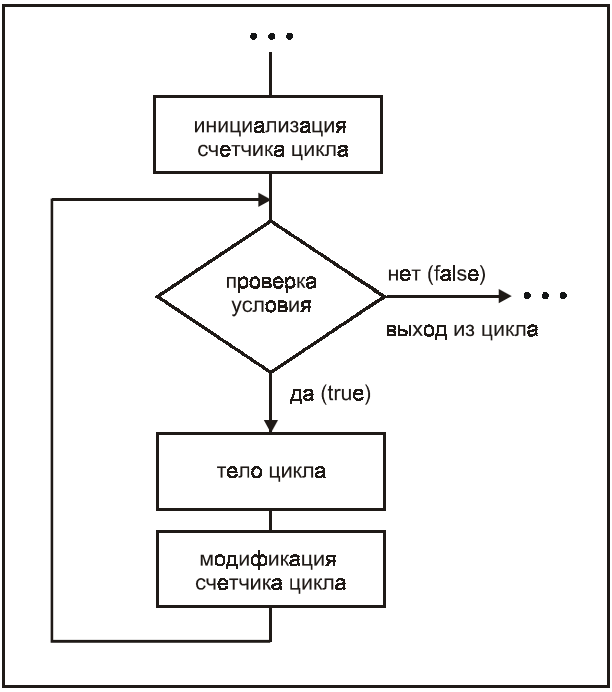


Рис. 7.8. Алгоритм выполнения цикла `for`

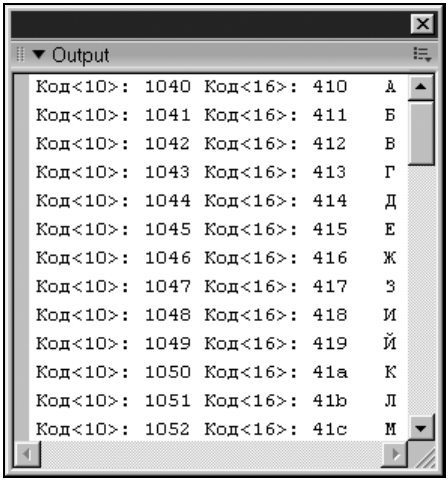


Рис. 7.9. Вывод кодов кириллических символов с помощью цикла `for`

Следующий пример выводит в панель **Output** перечень десятичных и шестнадцатеричных кодов кириллических символов в кодировке Unicode и соответствующие им символы. Подобным образом можно узнать код любого символа этой кодировки.

```
for(var i = 1040 ; i<=1103; i++){
trace("Код<10>: "+i+" Код<16>: " + i.toString(16) +↳
" "+String.fromCharCode(i));
}
```

На рис. 7.9 представлена панель **Output** после выполнения этого сценария.

Цикл *for...in*

Это особый тип цикла, который используется для обработки свойств объектов и элементов массива. Этот цикл последовательно проходит все свойства объекта, для каждого из них выполняя инструкции в теле цикла. Цикл `for...in` имеет следующий синтаксис:

```
for (счетчик in объект){
блок предложений;
...
}
```

Здесь:

- `for` и `in` — это ключевые слова;
- *`счетчик`* — это имя переменной, используемой в качестве счетчика. Переменная *`счетчик`* на каждой итерации принимает строковое значение, совпадающее с именем обрабатываемого свойства. Поскольку на каждой итерации цикла `for...in` происходит обращение к новому свойству, переменная *`счетчик`* всегда хранит имя этого свойства в виде строки, что дает возможность обратиться к свойству и выполнить с ним требуемые действия;
- *`объект`* — это объект, чьи свойства обрабатываются циклом `for...in`.

Следующий сценарий выводит в панель **Output** перечень всех переменных и экземпляров объектов, присутствующих на сцене.

```
for (i in _root){
trace(i);
}
```

В качестве объекта здесь используется `_root`, поскольку все экземпляры и переменные, расположенные на монтажной линейке какого-либо объекта, интерпретируются как свойства этого объекта. Если объект задается глобальным свойством `_root`, цикл `for...in` пройдет все свойства основной монтажной линейки, то есть обработает все экземпляры, расположенные на основной монтажной линейке. Кроме перечня имен переменных и экземпляров в панели **Output** будет также выведен идентификатор `$version`. Это автоматически создаваемая переменная, содержащая номер используемой версии проигрывателя Flash Player.

Попробуем немного модифицировать представленный пример и выполнить с экземплярами на сцене какие-то действия. Допустим, необходимо сообщить всем экземплярам, расположенным на основной монтажной линейке, 50%-ю прозрачность. С этой целью воспользуемся следующим кодом:

```
for (i in _root){
_root[i]._alpha = 50;
}
```

Для задания степени прозрачности мы воспользовались свойством `_alpha` *(о свойствах встроенных классов будет подробно сказано в дальнейшем)*. Следует обратить внимание на то, что переменная-итератор на каждой итерации содержит строку с именем свойства, а не непосредственную ссылку на него. В связи с этим для доступа к свойству мы используем оператор `[]`, который в качестве параметра принимает строку.

Цикл `for...in` не позволяет обрабатывать свойства и методы встроенных объектов и их потомков. Так, например, следующий код вместо того, чтобы вывести перечень всех глобальных свойств, оставляет панель **Output** пустой.

```
for (i in _global){
       trace(i); // Не выводит ничего
}
```

Для того чтобы обработать подобные скрытые от цикла `for...in` элементы, можно воспользоваться глобальной функцией `ASSetPropFlags()`. Отменить скрытие свойств встроенных объектов от цикла `for...in` можно следующим образом:

```
ASSetPropFlags(объект, null, 6, 1);
```

Здесь *`объект`* — это объект, свойства которого должны быть обработаны.

Так, например, следующий код выведет в панель **Output** перечень всех свойств глобального объекта `_global`, в числе которых есть также много недокументированных.

```
ASSetPropFlags(_global, null, 6, 1);
for (i in _global) {
       trace(i);
}
```

Таблица 7.2. *Предложения цикла в ActionScript*

Тип предложения	Синтаксис	Описание

`while`	`while (условие) {`
	`  блок кода;`
	`}`
		Блок кода в теле цикла выполняется до тех пор, пока проверочное условие остается истинным

***Таблица 7.2** (окончание)*

Тип предложения	Синтаксис	Описание

do while	do {
	блок кода;
	}
	while (*условие*)
		Вначале выполняется блок кода, после этого осуществляется оценка проверочного выражения. Повторно блок кода выполняется до тех пор, пока проверочное условие остается истинным

for	for (*инициализация, условие, модификация*)
	{
	блок кода;
	}
		В заголовке цикла выполняется инициализация счетчика, задается проверочное выражение и выражение, модифицирующее счетчик. Тело цикла выполняется до тех пор, пока проверочное условие остается истинным. Модификация счетчика выполняется после каждой итерации

for-in	for (*свойство* in *объект*){
	блок кода;
	}
		Последовательно проходит все свойства указанного в заголовке объекта, на каждой итерации сохраняя название текущего свойства в виде строки в переменной-итераторе *свойство*

Контроль выхода из цикла

В некоторых ситуациях необходимо осуществить выход из цикла еще до того, как выполнилось требуемое количество итераций. Для этих целей в ActionScript используется два специальных предложения: `break` и `continue`.

Предложение *break*

Предложение `break` позволяет завершить выполнение тела цикла и осуществляет выход из цикла. Это предложение используется так же, как и в конструкции `switch`, и имеет следующий синтаксис:

```
break;
```

Предложение `break` размещается в теле цикла и обычно применяется в сочетании с условным предложением. Его можно использовать для того, чтобы совершить выход из бесконечного цикла, или в случае, если в новых итерациях просто больше нет необходимости.

Вернемся к примеру со сценарием, который выводит коды символов в кодировке Unicode. Этот пример рассматривался в разделе, посвященном циклу `for`. Если мы заранее не знаем кода начальной и конечной позиции символов кириллического алфавита, то можно осуществлять перебор всех позиций, начиная с нуля до тех пор, пока не встретится последний символ, то есть буква "я". Значит, задача состоит в том, чтобы последовательно выводить в панель **Output** перечень кодов и символов

до тех пор, пока не будет обнаружен последний требуемый символ. Когда последний символ будет выведен в панель **Output**, произойдет выход из цикла.

```
var i:Number = 0;
while (true) {
i++;
trace("Код<10>: " + i + " Код<16>: " + i.toString(16) + " " +↳
String.fromCharCode(i));
if (String.fromCharCode(i).charAt(0) == "я") {
        break;
        }
}
```

Предложение *continue*

Предложение `continue` позволяет завершить выполнение блока кода в теле цикла на каждой итерации, однако в отличие от предложения `break` не завершает работу цикла, а продолжает его выполнение со следующей итерации. Таким образом, предложение `continue` просто позволяет пропустить следующие за ним предложения тела цикла, не осуществляя выход из цикла.

Предложение `continue` имеет следующий синтаксис:

```
continue;
```

Приведенный фрагмент кода удаляет все клипы с именами `mc1`, `mc2`, ..., `mc10`, горизонтальные координаты которых выходят за границы, заданные переменными `right` и `left`.

```
for (i=10;i>=0;i--){
if (this["mc"+i]._x<right && this["mc"+i]._x>left){
        continue; /* Если клип не выходит за границы, перейти к следующей
        итерации */
        }
        this["mc"+i].removeMovieClip(); /* Удаление клипа, выходящего за
        границы */
}
```

Вложенные циклы

Цикл, находящийся внутри другого цикла, называется *вложенным* (nested). Вложенные циклы используются для обработки многомерных структур, таких как матрицы, многомерные массивы и т. д. При использовании вложенных циклов сначала осуществляется инициализация счетчика внешнего цикла, а затем вложенного. Модификация счетчика внешнего цикла выполняется только после выполнения всех итераций вложенного цикла. Таким образом, для каждой итерации внешнего цикла выполняются все проходы или итерации внутреннего цикла. Например, если внешний цикл выполняет десять итераций, а вложенный — двадцать, то блок предложе-

ний вложенного цикла будет выполнен двести раз — по двадцать итераций на каждое значение счетчика внешнего цикла.

Рассмотрим пример вложенного цикла, обрабатывающего элементы двумерного массива. Двумерный массив — это матрица. Внешний цикл проходит по строкам матрицы, а вложенный — по столбцам. В каждой строке необходимо обработать все элементы, то есть на одну итерацию внешнего цикла приходится столько итераций вложенного цикла, сколько столбцов содержится в матрице. Двумерный массив — это массив, элементами которого также являются массивы. Обратите внимание, что нумерация элементов массива начинается с нуля. О массивах будет подробно рассказано в *главе 9*.

```
var ar1:Array = ["A", "B", "C", "D"]; // Первая строка
var ar2:Array = ["E", "F", "G", "H"]; // Вторая строка
var ar3:Array = ["I", "J", "K", "L"]; // Третья строка
var array2:Array = [ar1, ar2, ar3]; // Двумерный массив
for (i = 0; i < 3; i++) { // Внешний цикл по строкам
       for (j = 0; j < 4; j++) { // Вложенный цикл по столбцам
              trace(array2[i][j]);
       }
}
```

Этот код последовательно выводит в панель **Output** все элементы двумерного массива `array2`.

Циклы монтажной линейки

Цикл монтажной линейки или трехкадровый цикл — это совершенно особый тип цикла, реализуемый исключительно за счет специфики рабочей среды Macromedia Flash. Основываясь на том, что кадры монтажной линейки по умолчанию воспроизводятся последовательно и циклично, а также на том, что каждый ключевой кадр монтажной линейки может содержать сценарий, выполняющийся при вхождении воспроизводящей головки в этот кадр, можно достаточно просто организовать многократное выполнение определенного блока кода.

Цикл монтажной линейки в отличие от обычного программного цикла имеет три важные особенности.

- Все визуальные (и звуковые) изменения, реализуемые в блоке кода цикла монтажной линейки, отображаются непосредственно после каждой итерации.
- Цикл монтажной линейки может выполняться бесконечно, и это не вызовет появление предупреждающего сообщения.
- Временной интервал между выполнением итераций определяется скоростью воспроизведения монтажных линеек (fps), используемой в документе.

В общем виде для создания трехкадрового цикла, как следует из его названия, используется три ключевых кадра.

Каждый из этих кадров содержит сценарий и выполняет следующую роль:

1. В первом кадре выполняется инициализация всех переменных цикла. Здесь задаются начальные значения переменных или свойств объектов, которые будут модифицироваться в ходе выполнения цикла.
2. Во втором кадре находится блок кода, который должен выполняться на каждой итерации цикла. Второй кадр, таким образом, представляет собой тело цикла.
3. Третий кадр используется для модификации значений и зацикливания. Здесь содержится код, осуществляющий изменения тех или иных значений и отправляющий воспроизводящую головку во второй кадр, в результате чего анимация зацикливается между вторым и третьим кадрами и тело цикла выполняется многократно. В третьем кадре также может быть предусмотрена возможность выполнения каких-либо действий перед следующей итерацией, Так, например, можно организовать условие выхода из цикла или повторения всех итераций с самого начала. Для этого можно использовать условное предложение и в случае истинности заданного проверочного выражения либо остановить воспроизводящую головку (выход из цикла), либо отправить ее в первый кадр (запуск цикла сначала).

Для перехода от одного кадра к другому используются глобальные встроенные функции (или методы класса `MovieClip`). Подробнее об этих возможностях будет сказано в *главе 10*. Алгоритм выполнения трехкадрового цикла представлен на рис. 7.10.

Рассмотрим принцип использования трехкадрового цикла на простом примере. Имеется переменная `x`. В первом кадре присваиваем ей начальное значение:

```
// 1-ый кадр
x = 0;//Инициализация
```

Во втором кадре выводим ее значение в панель **Output**:

```
// 2-ой кадр
trace(x); // Вывод значения
```

В третьем кадре выполняем модификацию значения и проверку, не превысило ли текущее значение максимально допустимое. Для этого используем условный оператор. Если значение не превышает 10, переходим к кадру 2. В противном случае переходим к кадру 1, где значение переменной `x` обнуляется, и цикл выполняется снова.

```
// 3-ий кадр
x++; // Модификация значения
//Проверка достижения максимального значения
x<=10 ? gotoAndPlay(2): gotoAndPlay(1);
```

В результате работы такого цикла в панель **Output** будут выводиться значения: 0, 1, ..., 10, 0, 1, ..., 10, ...

Трехкадровый цикл можно применять для создания программной анимации. Такой подход является не самым эффективным, однако для того, чтобы лучше понять принцип использования трехкадрового цикла, рассмотрим еще один пример. Лис-

тинг 7.3 содержит трехкадровый цикл, позволяющий перемещать экземпляр клипа внутри рабочей области размером 550×400 так, чтобы при столкновении со стенкой клип отталкивался и двигался в противоположную сторону. В процессе перемещения клип будет вращаться вокруг своей оси.

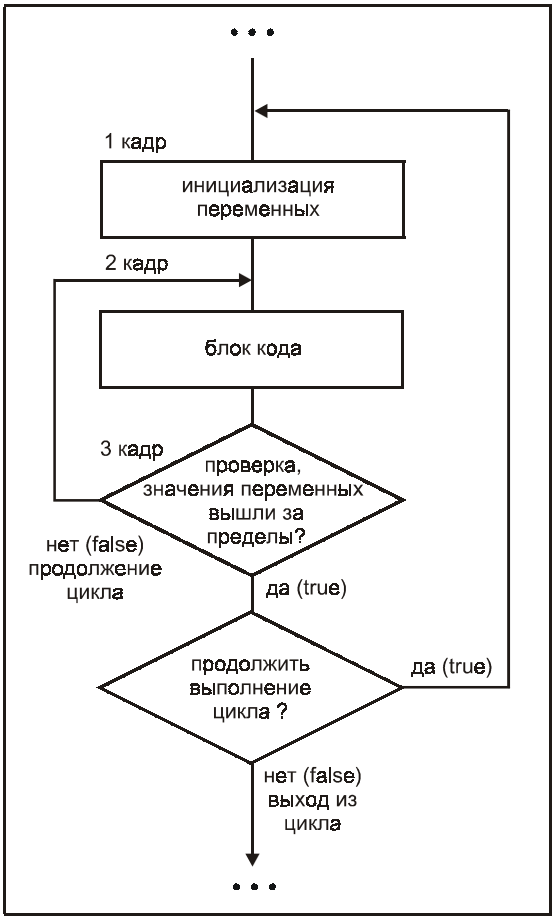


Рис. 7.10. Алгоритм выполнения трехкадрового цикла

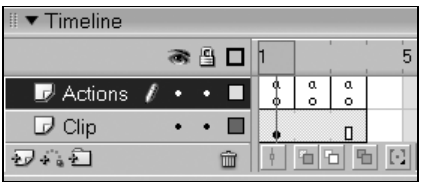


Рис. 7.11. Монтажная линейка. Структура трехкадрового цикла

Для реализации этой задачи используется два слоя. Структура монтажной линейки представлена на рис. 7.11. Слой Actions содержит сценарии, а на слое Clip расположен экземпляр клипа с именем mc.

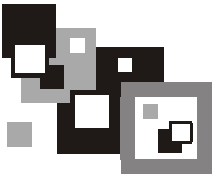
Листинг 7.3. Трехкадровый цикл. Перемещение внутри прямоугольной области

```
// Сценарий 1-го кадра
//Инициализация переменных
x=0; // Начальное значение x
y=400; // Начальное значение y
r = 0; // Начальный угол поворота
incX=5; // Приращение x на каждой итерации
incY=-10; // Приращение y на каждой итерации
incR = 10; // Приращение угла поворота на каждой итерации
// Вспомогательные переменные
w =.5*mc._width; // Половина ширины клипа mc
h = .5*mc._height; // Половина высоты клипа mc

// Сценарий 2-го кадра
// Задаем координаты клипа mc
mc._x = x;
mc._y = y;
mc._rotation = r;
// Сценарий 3-го кадра
// Модификация значений
x+=incX;
y+=incY;
r+=incR;
if (x > 550-w) {// Столкновение с правой границей
       x = 550-w; // Помещаем объект в допустимые границы
       incX = -incX;// Меняем направление приращения
}
if (x < w) {// Столкновение с левой границей
       x = w;// Помещаем объект в допустимые границы
       incX = -incX;// Меняем направление приращения
}
if (y > 400-h) {// Столкновение с верхней границей
       y = 400-h;// Помещаем объект в допустимые границы
```

```
        incY = -incY;// Меняем направление приращения
}
if (y < h) { // Столкновение с нижней границей
        y = h;// Помещаем объект в допустимые границы
        incY = -incY;// Меняем направление приращения
}
// Исключаем полные обороты
if(r>=360) r = 0;
gotoAndPlay(2); // Переход к следующей итерации
```

Глава 8



Пользовательские функции

Машинная программа выполняет то, что вы ей приказали делать,
а не то, что вы бы хотели, чтобы она делала.

Третий закон Грида

Функция является одним из наиболее важных понятий в программировании. Функции играют роль строительных блоков при разработке программ и позволяют рационально и эффективно подойти к процессу разработки. ActionScript включает широкий набор встроенных или предопределенных функций, которые могут быть непосредственно использованы для выполнения различных действий. Однако на практике приходится решать гораздо более широкий диапазон задач, чем те, которые решаются с помощью встроенных функций. В связи с этим возникает необходимость в создании своих собственных, пользовательских функций. Именно этому вопросу посвящается данная глава. После того как будут изучены вопросы создания и применения пользовательских функций, мы перейдем к рассмотрению возможностей встроенных функций и методов предопределенных классов.

Общие сведения о функциях

Функция или подпрограмма — это сегмент кода, к которому можно обратиться из другого участка программы с целью реализации выполняемых в нем действий или вычислений. Вместо многократного выполнения однотипных операций можно создать одну функцию, описывающую все выполняемые действия, и в дальнейшем при необходимости выполнения этих действий обращаться к соответствующей функции. Функции предоставляют возможность многократного использования набора команд, причем после создания функции можно использовать ее возможности, совершенно не заботясь о внутренних деталях выполнения вычислений и действий. Это позволяет повысить эффективность программы за счет уменьшения объема расходуемой памяти, сделать ее структуру более логичной и повысить ее читабельность.

Создание функции называется ее *объявлением*. Для создания функции можно использовать специальное предложение или литерал, то есть буквальное описание выполняемых функцией действий.

Обращение к функции с целью выполнения заложенного в нее сегмента кода называется *вызовом* функции. Для того чтобы выполнить сегмент кода функции, ее необходимо вызвать.

Поскольку функция всегда выполняет только предопределенный набор действий, часто возникает необходимость в выполнении этих действий с разными входными данными. Для этих целей функции можно передать ряд значений, называемых *параметрами*. Таким образом, функция, принимая различные параметры, выполняет с ними одинаковый набор действий и выдает результат, определяемый их значениями. Функция может вообще не принимать параметров. Такая функция называется процедурой.

Результатом работы функции может быть выполнение определенных вычислений, действий или конкретное значение. Если результатом работы функции является конкретное значение, говорят, что функция *возвращает* значение. В этом случае функция может быть использована в составе выражения, при вычислении которого она замещается своим возвращаемым значением.

Создание функции

Все функции ActionScript формально являются экземплярами встроенного класса `Function`, тем не менее создание функций выполняется без помощи конструктора. Для того чтобы создать функцию, можно либо непосредственно объявить ее, либо использовать литерал функции.

Объявление функции

Для объявления функции используется предложение `function`. Объявление функции включает следующие элементы.

- Указание имени функции, с помощью которого функция может быть вызвана в дальнейшем. При задании имен функций следует придерживаться общих правил именования, описанных в *разд. "Имена" главы 3*.
- Перечень параметров, принимаемых функцией. Параметры задаются в виде перечня идентификаторов, разделенных запятыми. ActionScript 2.0 позволяет строго указать тип данных для каждого используемого параметра.
- Набор инструкций, составляющих *тело функции*. Тело функции выполняется всякий раз при ее вызове.

В общем случае при объявлении функции используется следующий синтаксис:

```
function имя функции (параметр1, параметр2, ..., параметрN){
блок кода;
...
}
```

Здесь `function` — это ключевое слово, указывающее интерпретатору, что далее следует определение функции. В круглых скобках через запятую перечисляются параметры функции. Если функция не принимает параметров, то круглые скобки остаются пустыми, но их наличие обязательно! Ключевое слово `function`, имя функции и перечень ее параметров вместе образуют *заголовок функции*.

Блок кода в фигурных скобках составляет тело функции. Блок кода может состоять из неограниченного количества предложений, которые оформляются в соответствии со стандартным синтаксисом.

Рассмотрим пример создания простейшей функции с именем `countdown`, которая не принимает параметров и выводит в панель **Output** ряд чисел: 10, 9, ..., 0.

```
function countdown() {
        for (var i = 10; i >= 0; i--) {
                trace(i);
        }
}
```

Для того чтобы оценить результат работы этой функции, необходимо вызвать ее по имени следующим образом:

```
countdown(); // Выводит 10, 9, ..., 0
```

Использование литерала функции

Синтаксис литерала функции очень похож на синтаксис объявления функции за исключением того, что создаваемая с помощью литерала функция является анонимной, то есть у нее нет имени. В связи с этим литерал функции должен быть сохранен в переменной (или свойстве объекта), с помощью которой можно будет в дальнейшем вызывать эту функцию. Рассмотрим синтаксис литерала функции:

```
function (параметр1, параметр2, ..., параметрN){
блок кода;
...
};
```

Как видно, литерал отличается от объявления функции только отсутствием имени и точкой с запятой после закрывающей фигурной скобки тела функции. С применением литерала функции наш предыдущий пример можно реализовать следующим образом.

Вначале создадим переменную `countdown`, в которой затем будет сохранен литерал функции:

```
var countdown:Function;
```

Обратите внимание, поскольку мы собираемся сохранить в переменной функцию, в качестве типа этой переменной можно указать тип `Function`, определяемый встроенным классом `Function`, на основе которого создаются все функции.

Теперь создадим литерал и присвоим его переменной `countdown` в качестве значения:

```
countdown = function () {
        for (var i = 10; i >= 0; i--) {
                trace(i);
        }
};
```

После этого функция может быть вызвана с помощью имени содержащей ее переменной:

```
countdown(); // Как и раньше, выводит 10, 9, ..., 0
```

На практике литералы функции чаще всего используются при работе с методами обработчиков событий, что позволяет контролировать событие объекта (клипа, кнопки или компонента), не прикрепляя сценарий непосредственно к его экземпляру, и обрабатывать события различных встроенных классов.

Для наглядности приведем еще один пример использования литерала функции. Следующий код выводит ряд чисел в панель **Output** при нажатии на клип с именем `mc`. Для этих целей мы присваиваем вышеприведенный литерал функции свойству клипа `mc`, содержащему ссылку на метод обработчика событий `onPress`. Подробная информация об обработке событий клипов представлена в *главе 11*.

```
mc.onPress = function() {
        for (var i = 10; i >= 0; i--) {
                trace(i);
        }
};
```

Параметры функции

Параметры функции — это данные, которые передаются ей для выполнения заложенных в нее действий. Параметры, принимаемые функцией, указываются в ее заголовке в виде перечня имен, разделенных запятыми. Это так называемые *формальные параметры*, представляющие собой фиктивные переменные. Формальные параметры определяют, какое количество значений будет передано функции и как эти значения будут использованы в теле функции.

При вызове функции в качестве параметров ей передается ряд конкретных значений, которые связываются с формальными параметрами. Это так называемые *фактические параметры*, замещающие соответствующие им формальные параметры, указанные при создании функции.

Важно понимать, что все вычисления в теле функции осуществляются в терминах формальных параметров. Это означает, что при создании функции всюду, где вместо фиксированных значений должны быть использованы произвольные значения, переданные в качестве фактических параметров, необходимо использовать формальные параметры. Объявление функции — это своеобразный шаблон. При вызове функции фактические параметры замещают формальные и тело функции выполняется для фактически переданных значений. Поясним это на простом примере функции, вычисляющей площадь треугольника. Величины основания и высоты треугольника передаются функции в качестве параметров `base` и `height`.

```
function triangleArea(base, height) {
        trace(.5 * base * height);
}
```

Теперь вычислим площадь треугольника с основанием 10 и высотой 20. Для этого вызовем нашу функцию `triangleArea()`, передав ей соответствующие значения как параметры.

```
triangleArea(10, 20); // Выводит 100
```

Следует иметь в виду, что число фактических параметров, передаваемых функции в момент вызова, может не совпадать с числом формальных параметров, указанных при объявлении. Эти особенности будут рассмотрены далее в настоящем разделе.

Контроль типа параметров

Наряду со строгой типизацией переменных ActionScript 2.0 предоставляет возможность строго определить тип данных для значений, принимаемых параметрами функций. Так, например, если известно, что в качестве первого параметра функции передается число, можно при объявлении строго типизировать первый параметр, определив его принадлежность к типу `Number`. Если в дальнейшем при вызове этой функции передать ей параметр иного типа, это вызовет ошибку несоответствия типа (type mismatch), которая, как и при использовании строгой типизации переменных, будет обнаружена в результате автоматической проверки синтаксиса (Check syntax) или в среде тестирования. Контроль типа параметров преследует те же цели, что и контроль типа переменных. Он позволяет избежать возможных ошибок, связанных с тем, что функции будут переданы параметры неверного типа. Если контроль типа параметров не используется, интерпретатор не сможет определить факт передачи функции значений неверного типа, что, скорее всего, повлечет ее некорректную работу или вызовет синтаксическую или иную ошибку.

Для указания типа параметров, принимаемых функцией, используется тот же синтаксис, что и при объявлении переменной.

```
function имя функции (параметр 1:тип, параметр 2:тип, ..., параметр
N:тип){
блок кода;
...
}
```

Следующий пример иллюстрирует создание функции `convertBase()`, преобразующей заданное число в требуемую систему счисления. Функция принимает в качестве параметров два числа. Первый параметр `num` — это само число, которое нужно перевести в иную систему счисления, а второй `radix` — ее основание. Результат преобразования выводится в панель **Output** в виде строки.

```
function convertBase (num:Number, radix:Number){
       trace(num.toString(radix));
}
```

Вызовем функцию `convertBase()` для числа 12 и основания 2:

```
convertBase(12, 2); // Выводит 1100
```

А что произойдет, если передать функции один из параметров в виде строки?

```
convertBase("12", 2); // Ошибка несоответствия типа
```

Если бы параметры функции не были строго типизированы, то последнее предложение вызова не вызвало бы ошибки. Функция, получив в качестве первого параметра строку, просто вывела бы ее без изменений, и в панели **Output** появилось бы значение 12.

Следует отметить, что различные параметры могут иметь разный тип. Кроме того, тип некоторых параметров может быть указан, а для некоторых указание типа может отсутствовать.

Класс *Arguments*. Контроль фактических параметров функции

В ActionScript существует специальный встроенный класс `Arguments`, который позволяет обрабатывать параметры, переданные функции непосредственно в процессе ее выполнения, а также управлять вызовом функции из ее тела.

Объект `Arguments` (то есть экземпляр класса `Arguments`) представляет собой массив, содержащий значения всех фактических параметров, переданных функции при вызове. Объект `Arguments` автоматически создается в момент вызова функции. При этом также создается локальная переменная `arguments`, содержащая ссылку на этот объект. Поскольку фактически переданные параметры размещаются в массиве объекта `Arguments` в виде индексированных элементов, к любому из них можно обратиться из тела функции, используя переменную `arguments` в качестве ссылки на объект `Arguments`, и индекс элемента массива, соответствующий переданному параметру. Такое обращение будет иметь следующий вид:

```
arguments[индекс];
```

Здесь *индекс* — это число или выражение, дающее числовое значение, соответствующее порядковому номеру параметра. Следует иметь в виду, что элементы массива индексируются, начиная с нуля, а не с единицы. При заполнении массива `arguments` вначале размещаются значения параметров, которые были переданы функции раньше.

Примечание

Важно иметь в виду, что объект `Arguments` создается в момент вызова функции. Это означает, что к нему можно обратиться *только* при выполнении этой функции непосредственно из ее тела.

Рассмотрим следующий пример. Имеется некая функция, принимающая 3 параметра:

```
function someFunction (p1, p2, p3){
trace("1-ый параметр: "+ arguments[0]);
trace("2-ый параметр: "+ arguments[1]);
trace("3-ый параметр: "+ arguments[2]);
}
```

Вызовем функцию `someFunction()`, для наглядности придав ей параметры разных типов:

```
someFunction (1, "Dmitri", true);
```

Результатом выполнения этой функции будет следующее сообщение, выведенное в панели **Output**:

```
1-ый параметр: 1
2-ый параметр: Dmitri
3-ый параметр: true
```

Таким образом, объект `Arguments` дает нам мощные возможности обработки фактических параметров, передаваемых функции.

Свойство *arguments.length*

Это свойство содержит значение количества фактических параметров, переданных функции. Такая информация является чрезвычайно полезной в том случае, если количество параметров, передаваемых функции, заранее не известно или число фактических параметров отличается от числа формальных, указанных при объявлении функции.

В качестве примера приведем функцию `multiply()`, которая выводит в панель **Output** результат произведения всех переданных ей параметров. Поскольку количество параметров заранее не известно, необходимо оценить длину массива `arguments` с помощью свойства `length` и после этого в цикле обработать все элементы этого массива, то есть фактические параметры функции. Максимальное значение счетчика в проверочном выражении цикла задается на основании значения свойства `length`. Поскольку индексирование элементов в массиве начинается с нуля, используется строгое неравенство.

```
function multiply (){
var m = 1; /* Создаем локальную переменную для хранения промежуточного
результата умножения */
for(i=0;i<arguments.length; i++){
m *= arguments[i]; /* В цикле перемножаем параметры функции, сохраняя
результат в переменной m на каждой итерации */
       }
trace(m); /* После выхода из цикла переменная содержит конечный
результат */
}
```

Вызовем функцию `multiply()` несколько раз, передавая ей разные параметры:

```
multiply (1, 2, 3, 4); // Выводит 24
multiply (8.45, 15, 39.7, 65, 0); // Выводит 0
multiply (0x0f, 0x0f); // Выводит 225
```

Свойство *arguments.callee*

Данное свойство содержит ссылку на вызываемую функцию и позволяет вызвать функцию во время ее выполнения непосредственно из тела функции. Такая возможность применяется для создания рекурсивных функций, то есть функций, содержащих собственный вызов. Более подробно о рекурсивных функциях будет сказано в соответствующем разделе настоящей главы.

Рассмотрим модель применения данного свойства. Допустим, у нас имеется некая функция `myFunc()`, выполняющая определенные действия и принимающая два параметра `x` и `y`:

```
function myFunc(x, y){
инструкции;
...
}
```

Для того чтобы функция `myFunc()` могла вызвать саму себя, в теле этой функции можно либо поместить обычный вызов:

```
function myFunc(x, y){
myFunc(x, y) // Функция вызывает саму себя по имени
инструкции;
...
}
```

или воспользоваться свойством `arguments.callee`, содержащим ссылку на текущую выполняемую функцию:

```
function myFunc(){
arguments.callee(x ,y) /* Функция вызывает саму себя с помощью свойства
arguments.callee */
инструкции;
...
}
```

Обратите внимание на то, что после обращения к свойству `callee` следуют круглые скобки с перечнем параметров так же, как и при обычном вызове функции.

Зачем же нужно свойство `callee`, если можно вызвать функцию, просто используя ее имя? Дело в том, что у литерала функции имени не существует, и для того, чтобы заставить такую функцию выполняться рекурсивно, используется не имя, а ссылка на функцию, реализуемая с помощью свойства `callee`.

Рассмотрим простой пример на использование свойства `callee`. Ниже представлено объявление функции `count()`, которая принимает два параметра `start` и `end` и выводит в панель **Output** ряд чисел, расположенных с шагом в единицу. При этом первое число ряда задается параметром `start`, а последнее — параметром `end`.

```
function count(start, end) {
        if (start <= end) {
```

```
            trace(start);
            arguments.callee(start + 1, end); /* Вновь вызываем
            функцию count(), увеличивая на единицу значение параметра
            start */
        }
}
```

Вызываем функцию `count()`:

```
count(5, 15); // Выводит 5, 6, 7, ..., 15
```

Свойство *arguments.caller*

Это свойство содержит ссылку на функцию (или метод), вызвавшую текущую функцию. Если функция не вызывается из другой функции, то ее свойство `arguments.caller` имеет значение `null`.

Свойство `caller` позволяет из одной функции, которая вызывается другой функцией, вызывать эту другую функцию и таким образом инициировать и свой повторный вызов. Это может быть использовано при создании рекурсивной функции. Для того чтобы из текущей функции `f1()`, вызванной с помощью другой функции `f2()`, осуществить вызов функции `f2()`, нужно в теле функции `f1()` использовать следующий синтаксис:

```
arguments.caller(параметр1, параметр2, ..., параметрN);
```

Здесь *параметр1*, *параметр2*, ..., *параметр N* — это параметры, передаваемые функции `f2()` при ее вызове.

С помощью свойства `caller` также можно осуществить проверку, из какой функции вызвана текущая функция, и в зависимости от этого выполнить те или иные действия. Рассмотрим это на конкретном примере. Допустим, перед нами стоит задача создания универсальной функции, которая вычисляет площади различных фигур, назовем ее `calculateArea()`. Эта функция должна вызываться из других функций, связанных с конкретными фигурами и принимающих данные, необходимые для вычисления их площадей в качестве параметров:

```
// Функция для круга
function circle(radius) {
       calculateArea(radius); //Вычисляет площадь круга
}
// Функция для прямоугольника
function rectangle(side1, side2) {
       calculateArea(side1, side2); //Вычисляет площадь прямоугольника
}
// Функция для треугольника
function triangle(base, height) {
       calculateArea(base, height); //Вычисляет площадь треугольника
}
```

Таким образом, у нас имеются три функции, принимающие разное количество параметров, которые вызывают одну и ту же функцию для вычисления площади, передавая ей свои параметры. Как же можно заставить одну и ту же функцию выполнять разные вычисления в зависимости от того, какой функцией она была вызвана? Для этого как раз и удобно использовать свойство `caller`. В теле функции `calculateArea()` мы организуем проверку того, из какой функции она вызывается, и в зависимости от этого произведем расчет площади по соответствующей формуле. Ниже приводится код объявления функции `calculateArea()`.

```
function calculateArea() {
if (arguments.caller == circle) {
trace("S круга = " + Math.PI * arguments[0] * arguments[0]);
}
else if (arguments.caller == rectangle) {
trace("S прямоугольника = " + arguments[0] * arguments[1]);
}
else if (arguments.caller == triangle) {
trace("S треугольника = " + .5 * (arguments[0] * arguments[1]));
       }
}
```

Теперь вызовем функции каждой фигуры:

```
circle(10); // Выводит S круга = 314.159265358979
rectangle(5, 12); // Выводит S прямоугольника = 60
triangle(8, 16); // Выводит S треугольника = 64
```

Возвращение значения

До сих пор мы рассматривали функции, которые выполняли определенные действия, но не выдавали значение, полученное в результате произведенных вычислений. Во многих случаях требуется, чтобы результатом работы функции были не просто действия, но и конкретное значение, выдаваемое функцией перед прекращением ее работы. Передача функцией определенного значения, вычисленного в результате ее выполнения, называется *возвращением значения*, а само это значение называется *возвращаемым*.

В примерах, рассмотренных ранее, значение, полученное в результате вычислений, выводилось в панель **Output**. Такой подход, как правило, не имеет практической ценности, поскольку работа с панелью **Output** осуществляется исключительно в среде разработки Flash. Для того чтобы вычисленное функцией значение могло быть в дальнейшем использовано в программе, необходимо либо прямо из функции сохранить его во внешней переменной, либо возвратить это значение перед окончанием работы функции.

Для того чтобы функция возвращала значение, в теле этой функции необходимо разместить предложение `return`. Предложение имеет следующий синтаксис:

```
return выражение;
```

Здесь *выражение* — это выражение, значение которого возвращается в качестве возвращаемого функцией значения. Выражение может иметь значение любого типа. Кроме того, выражение может быть вообще опущено. В этом случае возвращаемым значением функции будет `null`.

Важно иметь в виду, что предложение `return` прекращает выполнение функции и возвращает результат, заданный выражением, указанным после ключевого слова `return`. Это означает, что если при обработке инструкций в теле функции интерпретатор встречает инструкцию `return`, он немедленно прекращает выполнение функции и возвращает требуемое значение. Все предложения тела функции, расположенные после предложения `return`, в случае его выполнения оказываются невыполненными. В связи с этим предложение `return` обычно используется либо в конце тела функции, либо в блоке условного предложения, что позволяет прекратить работу функции при выполнении определенного условия.

Рассмотрим простой пример функции `sum()`, выполняющей сложение двух чисел, переданных в качестве параметров, и возвращающей результат:

```
function sum(op1, op2) {
       return op1 + op2;
}
```

Если функция возвращает значение, то при вызове ее в составе выражения она замещается своим значением. Это дает нам возможность вывести результат вычислений в панель **Output**, непосредственно используя вызов функции `sum()` в качестве параметра функции `trace()`.

```
trace(sum(2, 3)); // Выводит 5
```

Возвращаемое функцией значение можно сохранить в переменной с тем, чтобы в дальнейшем мы могли им воспользоваться:

```
var sumResult:Number = sum(2, 3);
trace (sumResult); // Выводит 5
```

Покажем, что после выполнения предложения `return` функция завершает работу. Для этого в теле нашей функции `sum()` добавим еще одно предложение после предложения `return`.

```
function sum(op1, op2) {
       return op1 + op2;
       trace("Это предложение никогда не будет выполнено");
}
trace(sum(2, 3)); /* Выводит 5. Второе предложение тела функции не
выполняется */
```

Теперь рассмотрим более сложный пример функции `power()`, которая возводит переданное в качестве параметра число `num` в целую степень, заданную вторым параметром `pow`, и возвращает результат. Поскольку степень должна быть задана целым числом, необходимо выполнить проверку на допустимость параметра `pow`. (Код, вы-

полняющий проверку дробности числа, рассматривался *в разд. "Предложение else" главы 7*). Функция представлена в листинге 8.1.

Листинг 8.1. Функция для возведения числа в целую степень

```
function power(num:Number, pow:Number) {
/* Сохраняем переданное в качестве параметра число в локальной переменной
initNum */
var initNum = num;
// Проверка дробности степени
if(String(pow).indexOf(".")!=-1){
//Если степень дробная, возвращаем NaN
        return NaN;
}
//Если степень положительна
else if (pow > 0) {
// Выполняем последовательное умножение числа на само себя
        for (var i = 1; i < pow; i++) {
        num *= initNum;
        }
// Возвращаем результат
        return (num);
}
// Если степень меньше нуля
else if (pow < 0) {
/* Выполняем последовательное умножение числа на само себя. В проверочном
выражении цикла значение параметра pow берется по модулю, так как степень
отрицательна */
        for (var i = 1; i < Math.abs(pow); i++) {
        num *= initNum;
        }
/* Возвращаем результат в соответствии с правилом вычисления
отрицательных степеней */
        return (1/num);
}
// Если степень равна нулю, возвращаем 1
else {
        return 1;
        }
}
```

Проверим работу функции:

```
trace(power(2, 5)); // Выводит 32
trace(power(2, -2)); // Выводит 0.25
trace(power(256, 0)); // Выводит 1
trace(power(1.2, 2)); // Выводит 1.44
trace(power(1.2, 2.2));// Выводит NaN
```

Контроль типа возвращаемого значения

Строгий контроль типов данных, появившийся в ActionScript 2.0, позволяет строго типизировать не только переменные и параметры функций, но и определять тип возвращаемого функцией значения. Если при объявлении функции тип возвращаемого ею значения будет строго определен, то в случае возвращения значения иного типа произойдет ошибка несоответствия типа. В результате обнаружения подобной ошибки в панель **Output** выводится сообщение, представленное на рис. 8.1.

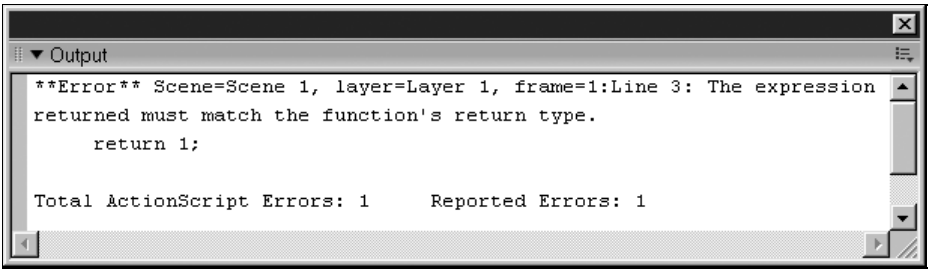


Рис. 8.1. Сообщение о несоответствии типа возвращаемого выражения типу возвращаемого функцией значения

Контроль типа возвращаемого функцией значения преследует те же цели, что и контроль типа значения переменных, — избежать возможных ошибок несоответствия типов и сделать программу более читабельной и понятной. Для указания типа возвращаемого функцией значения используется следующий синтаксис:

```
function имя функции (параметр1:тип, ..., параметрN:тип):тип {
блок кода;
...
}
```

Примечание

Если функция не возвращает значения, в качестве типа возвращаемого значения указывается специальный тип `Void`.

В листинге 8.2 приводится функция `calculate()`, выполняющая четыре арифметических действия над двумя числами, переданными ей в качестве параметров. Третьим параметром является строка, содержащая оператор, определяющий, какое именно действие должно быть выполнено над числами. При объявлении функции строго указаны типы ее параметров и возвращаемого результата.

Листинг 8.2. Функция — арифметический калькулятор

```
function calculate(op1:Number, op2:Number, operator:String):Number {
       switch (operator) {
       case "+" :
              return (op1 + op2);
       case "-" :
              return (op1 - op2);
       case "*" :
              return (op1 * op2);
       case "/" :
              return (op1 / op2);
       }
}
```

Вызовем функцию `calculate()`:

```
trace(calculate(15, 16, "+")); //Выводит 31
trace(calculate(0, 25, "-")); //Выводит -25
trace(calculate(25, 4, "*")); //Выводит 100
trace(calculate(1, 0, "/")); //Выводит Infinity
```

В следующем примере, представленном в листинге 8.3, создается функция, позволяющая переводить рубли в доллары и наоборот. Функция принимает три параметра: `amount` — денежная сумма, которая должна быть переведена, `rate` — курс, определяющий количество рублей за один доллар, `dir` — направление перевода. Последний параметр может принимать два значения (`true` — в рубли, `false` — в доллары).

Листинг 8.3. Простой конвертер валют

```
function convert (amount:Number, rate:Number, dir:Boolean):Number{
       if(dir)return amount*rate; // Перевод в рубли
       else return amount/rate; // Перевод в доллары
       }
trace(convert (1000, 29.7, true)); //Выводит 29700
```

Вызов функции

При создании функции инструкции, содержащиеся в ее теле, сами по себе не выполняются. Для того чтобы инициировать их выполнение, необходимо *вызвать* функцию. В предыдущих примерах мы уже осуществляли вызов функции для того, чтобы оценить результат ее работы. Для этих целей мы применяли один и тот же наиболее часто используемый способ. Однако ActionScript предоставляет несколько различных способов вызова функции, описанию которых и посвящен этот раздел.

Вызов функции по имени

Наиболее часто используемый способ вызова функции предполагает указание ее имени, после которого следует оператор вызова функции `()`. Если функции передаются параметры, то при вызове их фактические значения указываются в круглых скобках через запятую. Этот способ уже многократно использовался в предыдущих примерах, поскольку он является наиболее употребительным и широко используемым. Вызов функции по имени имеет следующий синтаксис:

```
имя функции (параметр1, параметр2, ..., параметрN);
```

Здесь *имя функции* — это идентификатор функции.

Значения *параметр1, параметр2, ..., параметрN* — это фактические параметры, передаваемые функции.

Следует отметить, что вызов функции может выполняться не только в отдельном предложении, но и входить в состав выражения. Всюду, где используется идентификатор функции, за которым следуют круглые скобки, выполняется вызов функции. Если функция возвращает значение, то при необходимости использования возвращаемого значения в выражении можно вызвать эту функцию. Поясним это на примере.

Пусть имеется некая функция `test()`, выполняющая определенные операции и в результате возвращающая булево значение `true` или `false`. Если в зависимости от возвращаемого ей значения мы должны выполнить те или иные действия, можно в составе проверочного выражения условного предложения непосредственно использовать вызов функции `test()`:

```
if(test()){
trace(true); // Если test() возвращает true
}
else {
trace (false); // Если test() возвращает false
}
```

При оценке проверочного выражения происходит вызов функции `test()`, она выполняет свой сегмент кода и возвращает булево значение обратно в выражение.

Если возвращаемое функцией значение нужно сохранить в переменной, то вызов функции можно поместить в правой части присваивания:

```
var testResult:Boolean = test();
```

Аналогично в данном случае осуществляется вызов функции `test()`, и после окончания ее работы возвращаемое значение присваивается переменной `testResult`.

Применение вызова функции по имени — это понятный и логичный способ, использующий наглядный и лаконичный синтаксис. Этот способ используется в большинстве случаев. Тем не менее он имеет несколько существенных недостатков:

- не позволяет динамически генерировать имя функции;
- идентификатор `this`, используемый в теле функции, всегда ссылается на монтажную линейку, содержащую объявление данной функции. Если функция вызывается в качестве метода объекта, то используемый в ее теле идентификатор `this` возвращает ссылку на этот объект;
- не позволяет передать функции произвольное количество параметров в виде массива;
- не позволяет периодически вызывать функцию с определенным интервалом.

При необходимости устранения этих недостатков можно использовать другие способы вызова функции, описанные ниже.

Вызов функции с помощью ссылки

Как было сказано ранее, любая функция является объектом класса `Function`. Это означает, что, как и любой объект, функция передается по ссылке *(см. разд. "Элементарные и ссылочные типы данных" главы 5)*. Соответственно, у нас имеется возможность сохранить ссылку на функцию в переменной и в дальнейшем вызывать эту функцию, используя идентификатор переменной и оператор вызова функции `()`. Рассмотрим простой пример:

```
var t:Function=trace; // Сохраняем ссылку на функцию trace() в переменной
t("ActionScript"); // Выводит в панель Output слово ActionScript
```

Обратите внимание на то, что в первом предложении круглые скобки после идентификатора функции в правой части присваивания не нужны, поскольку мы не вызываем функцию `trace()`, а только сохраняем ссылку на нее в переменной `t`, типом которой является тип `Function`.

Необходимо четко понимать различие между вызовом функции и передачей ссылки на эту функцию. Рассмотрим два следующих предложения:

```
var t = myFunction;
var t = myFunction();
```

Несмотря на внешнее сходство их смысл совершенно различен. В первом предложении мы сохраняем в переменной `t` ссылку на функцию `myFunction()`; при этом переменная получает тип `Function`, определяемый классом `Function`, лежащим в основе любой функции.

Во втором предложении мы вызываем функцию `myFunction()` и присваиваем переменной `t` значение, возвращаемое этой функцией. Тип переменной определяется типом возвращаемого значения. Использование строгой типизации переменных мо-

жет сделать эти предложения более наглядными (предположим, что возвращаемое значение имеет числовой тип):

```
var t:Function = myFunction;
var t:Number= myFunction();
```

Итак, мы выяснили, что ссылка на функцию может храниться в переменной. В *разд. "Адресация переменных" главы 4* обсуждались два способа указания пути к целевой переменной (или объекту): точечный синтаксис и квадратные скобки `[]` (оператор доступа к элементам массива). При использовании точечного синтаксиса для вызова функции с помощью ссылки, хранящейся в переменной, необходимо указать идентификатор этой переменной. Применение квадратных скобок предполагает обращение к переменной как к свойству содержащего ее объекта и позволяет передать строковое представление ее идентификатора. При этом используется следующий синтаксис:

```
объект[имя переменной](параметр1, параметр2, ..., параметрN);
```

Здесь `объект` — это ссылка на объект, вызывающий функцию.

Операнд *имя переменной* — это строка или выражение, дающее строковое значение. Значение в квадратных скобках задает идентификатор переменной, которая содержит ссылку на вызываемую функцию.

В круглых скобках задаются фактические параметры вызываемой функции.

Рассмотрим это на примере:

```
var go_s = gotoAndStop; // Сохраняем ссылку на функцию gotoAndStop()
_root["go_s"](25); // Переход на 25-ый кадр
```

Использование такого подхода при вызове функции позволяет динамически задавать имена переменных, содержащих ссылки на требуемые функции.

В качестве иллюстрации сказанного модифицируем сценарий кнопки-флага, который был представлен в листинге 7.1. Сохраним ссылки на глобальные функции `play()` и `stop()` в переменных `doPlay` и `doStop` соответственно. В качестве флага используем переменную `flag`, которая будет принимать одно из двух строковых значений: `"Play"`, если анимация остановлена, и `"Stop"`, если анимация воспроизводится.

```
var flag = "Stop"; // Начальное значение флага
var doPlay = play; // Сохраняем ссылку на функцию play()
var doStop = stop;// Сохраняем ссылку на функцию stop()
// При нажатии кнопки с именем but...
but.onPress=function(){
if (flag=="Play"){
_root["do"+ flag](); //Вызов функции play()
flag = "Stop"; //Переключаем флаг
       }
```

```
else if(flag=="Stop"){
_root["do"+ flag](); //Вызов функции stop()
flag = "Play"; //Переключаем флаг
       }
}
```

Вызов функции с помощью метода *apply()*

Любая функция, будучи объектом типа `Function`, может вызывать методы класса `Function`. В классе `Function` имеется только два предопределенных метода: `call()` и `apply()`. Методы `call()` и `apply()` класса `Function` формально связаны с вызовом функции, однако для их понимания требуется наличие представления об области видимости функции. Если читатель встретится с трудностями понимания при изучении данного материала, рекомендуем вначале обратиться к *разд. "Область видимости функции" настоящей главы.*

Метод `Function.apply()` используется для передачи функции, произвольного количества параметров в виде массива. Это может быть очень удобно в ситуации, когда количество параметров заранее неизвестно или когда они являются результатом выполнения других функций. Метод `apply()` также позволяет указать объект, определяющий область видимости функции, на который будет ссылаться идентификатор `this`, используемый в теле функции. При вызове функции обычным образом ссылки с применением `this` в теле функции разрешаются относительно объекта, содержащего ее объявление. Метод `apply()` имеет следующий синтаксис:

функция`.apply(`*объект*`, `*массив параметров*`);`

- *функция* — это идентификатор вызываемой функции.
- Параметр *объект* — это ссылка на объект, относительно которого в теле функции будут разрешаться все ссылки с использованием идентификатора `this`. Другими словами, при передаче в качестве данного параметра ссылки на объект, используемый в теле функции, идентификатор `this`, используемый в теле функции *функция*, будет ссылаться на этот объект. Если в теле функции `this` должен возвращать ссылку на объект, содержащий ее объявление (что является обычным поведением при вызове функции без применения метода `apply()`), то в качестве первого параметра методу `apply()` передается значение `null`.
- Второй параметр метода `apply()` — это массив, элементы которого передаются функции в качестве параметров. Для передачи массива, как правило, используется идентификатор переменной, содержащий ссылку на этот массив, хотя можно применить и литерал массива.

Приведем два примера, иллюстрирующих применение данного метода.

Создадим функцию, которая задает значения координат текущего объекта (то есть объекта, содержащего ее объявление) на основании двух чисел, переданных ей в качестве параметров. Для задания координат используются глобальные свойства `_x` и `_y`. Назовем эту функцию `setPosition`.

Ниже приводится код ее объявления.

```
function setPosition(){
this._x=arguments[0];
this._y=arguments[1];
}
```

Обратите внимание, что при объявлении функции мы не указали формальные параметры, которые собираемся ей передать. Обработку фактических параметров мы выполняем с помощью ссылки на массив `arguments`, предполагая, что из переданных функции параметров первые два соответствуют координатам. Наша функция `setPosition()` имеет очень ограниченную область действия, поскольку она может позиционировать только тот объект, на монтажной линейке которого находится ее объявление. Так, если объявление находится в первом кадре основной монтажной линейки, то идентификатор `this` в теле функции будет содержать ссылку на `_root`. Вследствие этого вызов функции `setPosition()` сместит все содержимое рабочей области так, что ее левый верхний угол попадет в переданные в качестве параметров координаты.

```
setPosition(400, 400); /*Смещает рабочую область в точку с координатами
(400; 200)*/
```

Для того чтобы, не изменяя саму функцию, использовать ее для смещения других объектов, воспользуемся методом `apply()`. Допустим, на основной монтажной линейке находится клип с именем `mc`. Вызовем функцию `setPosition()`, передав ей в качестве первого параметра ссылку на `mc`, а в качестве второго — массив, состоящий из двух элементов, соответствующих координатам по *X* и по *Y*.

```
var coords:Array = [400, 200]; // Создаем массив для хранения координат
setPosition.apply(mc, coords); /* Вызываем функцию для клипа mc,
передавая ей массив coords */
```

В результате только клип `mc` смещается в точку с координатами (400; 200).

Рассмотрим еще один пример. Создадим функцию `test()`, которая будет выводить в панель **Output** значения всех переданных ей параметров с указанием их типа.

```
function test() {
        for (i = 0; i < arguments.length; i++) {
                trace(arguments[i]+" : "+typeof(arguments[i]));
        }
}
```

Если мы хотим передать этой функции параметры в виде массива, воспользуемся методом `apply()`. Поскольку теперь нет необходимости в изменении значения `this`, в качестве первого параметра метода `apply()` передадим значение `null`.

```
var parameters:Array = [1, "2", true, ["a", "b", "c"]]; // Массив с
параметрами
test.apply(null, parameters); /* вызываем функцию test(), передавая ей
параметры в виде массива */
```

При выполнении этого кода в панели **Output** получаем следующее:

```
1 : number
2 : string
true : boolean
a,b,c : object
```

Вызов функции с помощью метода *call()*

Метод `call()` по своему назначению очень близок методу `apply()`. Различие заключается лишь в том, что метод `apply()` принимает параметры функции в виде массива, а метод `call()` предполагает непосредственную передачу параметров. Таким образом, назначение метода `call()` является более узким. Он предназначен для того, чтобы указать объект, относительно которого в теле функции будут разрешаться все ссылки с использованием идентификатора `this`.

Метод `call()` имеет следующий синтаксис:

функция`.call(`*объект*`, `*параметр1*`, `*параметр2*`, ..., `*параметрN*`);`

- *функция* — это идентификатор вызываемой функции.
- Параметр *объект* — это ссылка на объект, относительно которого в теле функции будут разрешаться все ссылки с использованием идентификатора `this`. Если значением идентификатора `this`, используемого в теле функции, должна быть ссылка на объект, содержащий ее объявление, то в качестве первого параметра методу `call()` передается значение `null`.
- Параметры с первого по *N*-й — это фактические параметры, передаваемые функции.

Приведенный пример с функцией `test()` и методом `apply()` может быть переписан с применением метода `call()` следующим образом:

```
_root.test.call(null, 1, "2", true, ["a", "b", "c"]); /* вызываем функцию
test(), передавая ей параметры с помощью метода call() */
```

При выполнении этого кода в панели **Output** получаем тот же результат:

```
1 : number
2 : string
true : boolean
a,b,c : object
```

Периодический вызов функции. Глобальные функции *setInterval()* и *clearInterval()*

Для выполнения множества задач необходимо периодически с определенным интервалом времени вызывать требуемую функцию. ActionScript предоставляет несколько различных способов решения этой проблемы.

К их числу можно отнести:

- трехкадровый цикл;
- функции `setInterval()` и `clearInterval()`;
- обработчик событий `onEnterFrame`;
- глобальная функция `getTimer()`.

Данный раздел посвящен изучению глобальных функций `setInterval()` и `clearInterval()`.

Функция *setInterval()*

Данная функция позволяет с заданной периодичностью вызывать требуемую функцию и позволяет передавать этой функции необходимые для работы параметры. Функция `setInterval()` применяется в ситуациях, требующих периодического контроля и изменения состояния объекта, процесса, вычислений и т. д. Функция `setInterval()` имеет следующий синтаксис:

`setInterval(`*функция*`, `*интервал*`, `*параметр1*`, `*параметр2*`, ..., `*параметрN*`);`

- Параметр *функция* — ссылка на вызываемую функцию. Следует иметь в виду, что значением этого параметра является идентификатор функции, то есть ссылка на объект `Function`, а не вызов функции. Поэтому круглых скобок после параметра *функция* быть не должно. Допускается в качестве данного параметра использовать литерал функции.
- Параметр *интервал* — промежуток времени между вызовами функции, заданный в миллисекундах.
- Остальные параметры функции `setInterval()` — это фактические параметры, передаваемые вызываемой функции. Если вызываемая функция не принимает параметров, они не указываются и просто опускаются.

Функция `setInterval()` имеет возвращаемое значение, представляющее собой идентификационный номер интервала. Идентификационный номер — это число, уникальное для каждого вызова функции `setInterval()`. Так, при первом вызове `setInterval()` возвращает 1, при втором — 2 и т. д. Как правило, идентификационный номер интервала сохраняется в переменной, поскольку при необходимости прекратить периодический вызов функции его можно передать в качестве параметра другой глобальной функции `clearInterval()`. Рассмотрим простой пример.

Создадим функцию `messageFunction()`, которая будет выводить в панель **Output** текстовое сообщение, передаваемое ей в качестве параметра `mes`, с порядковым номером вызова функции (переменная `times`) . Для того чтобы при каждом новом вызове порядковый номер, присоединяемый к сообщению, увеличивался на единицу, используем оператор инкрементирования.

```
function messageFunction (mes){
       times==undefined ? times = 1 : times ++;
       trace(mes+" "+times);
}
```

Поскольку при первом вызове функции `messageFunction()` переменная `times` еще не определена, используем условный оператор для выполнения проверки наличия у этой переменной значения. Если значение не определено (при первом вызове), задаем начальное значение 1, если значение определено, увеличиваем его на единицу с помощью инкрементирования.

Для того чтобы функция `messageFunction()` вызывалась периодически с интервалом в 1 секунду (1000 миллисекунд), используем функцию `setInterval()` и сохраним идентификатор интервала в переменной `int`:

```
var int:Number = setInterval(messageFunction, 1000, "Проверка");
```

В панели **Output** с интервалом в одну секунду будут появляться следующие сообщения:

```
Проверка 1
Проверка 2
Проверка 3
...
```

и т. д.

В качестве первого параметра функции `setInterval()` может быть передан не только идентификатор функции, но и литерал, то есть анонимная функция. Предыдущий пример может быть записан следующим образом:

```
var int:Number = setInterval(function(mes){times==undefined ? times = 1↵
:times ++; trace(mes+" "+times);}, 1000, "Проверка");
```

Такая запись является синтаксически верной, однако трудночитаемой.

Следует иметь в виду, что временной интервал вызова функции не всегда соблюдается строго. Если значение параметра *интервал* меньше времени экспонирования кадра, определяемого используемой в фильме скоростью воспроизведения (fps), то временной промежуток между вызовами функции соответствует заданному интервалу настолько, насколько позволяют ресурсы процессора. Если же значение заданного интервала, наоборот, превышает время экспонирования кадра, то функция вызывается с временным промежутком, кратным времени экспонирования. Пусть значение скорости воспроизведения документа составляет 10 fps. При этом время экспонирования одного кадра составляет 0,1 секунды или 100 миллисекунд. Если параметр *интервал* задается меньше или равным 0,1 секунды, проигрыватель Flash Player пытается обеспечить требуемую частоту вызова. Если параметр *интервал* задается больше 0,1 секунды, функция будет вызываться с интервалом, кратным 100 миллисекундам, то есть через 100 или 200, или 300 и т. д. миллисекунд.

Следует иметь в виду, что при применении функции `setInterval()` для вызова метода объекта необходимо использовать следующий синтаксис:

```
setInterval(объект, "метод", интервал, параметр1,..., параметрN);
```

- Парметр *объект* — это ссылка на объект, содержащий объявление метода, который должен быть вызван.

- Параметр *метод* — это строка, содержащая имя метода, который должен вызываться с определенным интервалом. В отличие от вызова простой функции, здесь имя метода обязательно должно передаваться в виде строки.
- Остальные параметры аналогичны параметрам, передаваемым функции `setInterval()` для вызова обычной функции, которые были рассмотрены ранее.

Обновление экрана. Применение функции *updateAfterEvent()*

Если некоторая функция должна периодически вызываться посредством глобальной функции `setInterval()` с интервалом, меньшим, чем время экспонирования кадра, то в этом случае рекомендуется в теле вызываемой функции использовать специальную глобальную функцию `updateAfterEvent()`. Функция `updateAfterEvent()` позволяет обновлять графическую информацию на экране при каждом своем вызове. Такая потребность возникает, если частота выполняемых действий оказывается меньше, чем скорость воспроизведения фильма. Поскольку проигрыватель Flash Player обновляет экран с частотой, равной скорости воспроизведения, используемой в фильме, некоторые движения или трансформации могут выполняться скачкообразно, рывками. Для того чтобы придать таким движениям плавность, используется функция `updateAfterEvent()`. Правда, используя данную функцию для анимации большого количества графики или сложных графических объектов, стоит помнить о том, что это создает дополнительную нагрузку на компьютер и может привести к снижению быстродействия проигрывателя.

Примечание

Функция `updateAfterEvent()` обновляет экран, только будучи вызванной с помощью функции `setInterval()` или посредством некоторых обработчиков событий. Во всех остальных случаях вызов функции `updateAfterEvent()` не дает результатов.

Эффект, оказываемый функцией `updateAfterEvent()`, можно наглядно пронаблюдать на следующем примере. В листинге 8.4 представлена функция `followMouse()`, заставляющая следовать за курсором мыши клип, ссылка на который передается ей в качестве параметра. Для этих целей используются встроенные свойства `_xmouse` и `_ymouse`, содержащие координаты курсора мыши относительно системы координат целевого объекта. Допустим, что в кадре, содержащем объявление функции `followMouse()`, находится экземпляр клипа с именем `mc`, который должен следовать за курсором. Будем вызывать функцию `followMouse()` с помощью функции `setInterval()` каждые 25 миллисекунд. Для плавного движения необходимо в теле функции использовать функцию `updateAfterEvent()`.

Листинг 8.4. Движение за курсором с помощью функции `setInterval()`

```
function followMouse(obj){
obj._x = obj._parent._xmouse;
obj._y = obj._parent._ymouse;
updateAfterEvent();
}
setInterval(followMouse, 25, mc); /* Вызываем функцию followMouse(),
с интервалом 25 миллисекунд, передавая ей ссылку на mc */
```

Чтобы оценить эффект, оказываемый функцией updateAfterEvent(), можно закомментировать ее вызов и посмотреть результат. Движение будет выполняться значительно менее плавно.

Функция *clearInterval()*

Функция clearInterval() позволяет прекратить периодический вызов функции, инициированный с помощью функции setInterval(). Совместно используя функции setInterval() и clearInterval(), можно достаточно гибко управлять периодическим вызовом функции. Функция clearInterval() имеет следующий синтаксис:

```
clearInterval(идентификатор интервала);
```

Здесь идентификатор интервала — это числовой идентификатор, возвращаемый функцией setInterval(), осуществляющей периодический вызов требуемой функции. Вызов функции clearInterval() и передача ей идентификатора интервала прекращает действие соответствующей функции setInterval() или, как говорят, "очищает интервал". Идентификатор интервала передается функции clearInterval() с помощью имени переменной, в которой он был сохранен при вызове функции setInterval(), или просто как целочисленный числовой литерал.

Рассмотрим следующий пример. Имеется кнопка с именем but. Эта кнопка должна двигаться по горизонтали в заданных границах. При нажатии на кнопку but она начинает двигаться. При повторном нажатии она останавливается. Листинг 8.5 содержит функцию moveX(), реализующую перемещение объекта в заданных границах. Функция принимает три параметра. Параметр obj — это объект, который нужно перемещать; параметры left и right задают левую и правую границы соответственно, за которые при перемещении не должна выходить точка регистрации объекта. В листинге 8.5 также приведен сценарий самой кнопки but, где с помощью переменной-флага flag реализуется попеременный вызов функций setInterval() и clearInterval(), что позволяет инициировать периодический вызов функции moveX(), перемещающей кнопку, и очищать интервал, останавливая ее движение.

Листинг 8.5. Применение функций setInterval() и clearInterval() для контроля анимации

```
//Функция для перемещения объекта. Сценарий кадра
function moveX(obj, left, right) {
// При первом вызове начинаем двигаться влево
       if(dir==undefined)dir="left";
// Достижение левой границы
       if (obj._x < left) {
              obj._x = left;
// Смена направления движения
              dir="right";
       }
```

```
// Достижение правой границы
       else if (obj._x > right) {
              obj._x = right;
// Смена направления движения
              dir="left"
       }
// Если движемся вправо, увеличиваем значение координаты X
       if(dir=="right")obj._x+=1;
// Если движемся влево, уменьшаем значение координаты X
       else if(dir=="left") obj._x-=1;
       updateAfterEvent();
}
// Сценарий кнопки
on(press){// При нажатии...
if(flag){
clearInterval(int); // Останавливаем движение
flag=false; //Переключаем флаг
       }
else {
int=setInterval(moveX, 10, but, 175, 375); /* Запускаем движение,
вызывая функцию moveX () через каждые 10 миллисекунд, передавая ей
в качестве параметров ссылку на целевой объект и границы горизонтального
перемещения */
flag = true; //Переключаем флаг
       }
}
```

Вопросы области видимости и времени существования функций

Как и переменные, функции могут быть объявлены в сценариях, прикрепленных к различным монтажным линейкам (основной или монтажной линейке клипа). В связи с этим возникает ряд вопросов, связанных с областью видимости, временем существования функций, а также возможностью их вызова с различных монтажных линеек.

Область видимости функции

Функции имеют свою собственную область видимости, ограниченную предложениями тела функции. Параметры, передаваемые функции, существуют и являются доступными для предложений функции исключительно в пределах этой локальной

области видимости. Таким образом, мы не можем получить доступ к параметрам функции извне.

Переменные, которые объявляются внутри функции с помощью предложения `var`, называются локальными, поскольку они существуют только на протяжении времени выполнения функции и являются доступными только предложениям тела функции. Внешние инструкции не имеют возможности обратиться к локальным переменным. Локальные переменные были подробно описаны в соответствующем разделе *главы 4*.

Наряду с локальной областью видимости у функции имеется еще одна область видимости, на которую непосредственно распространяется действие инструкций, находящихся в теле функции. Этой областью видимости является объект, монтажная линейка которого содержит объявление функции. Все ссылки внутри тела функции разрешаются либо относительно ее локальной области видимости, либо относительно объекта, содержащего ее объявление.

Кроме указанных областей видимости, областью видимости функции также является глобальный объект `_global`. Это так называемая глобальная область видимости.

При разрешении ссылок вначале исследуется локальная область видимости функции, если требуемый элемент в ней не найден, выполняется поиск в объекте, содержащем объявление функции, если и на его монтажной линейке целевой объект (или переменная) не найден, выполняется поиск в объекте `_global`.

Если в теле функции осуществляется обращение к переменной, то при наличии соответствующей локальной переменной ссылка разрешится относительно нее, если же локальной переменной с таким именем не существует, будет выполнен поиск переменной монтажной линейки, содержащей функцию. Рассмотрим пример, иллюстрирующий вопросы приоритета области видимости функции:

```
var k = "Это не локальная переменная"; /* Создаем переменную вне тела
функции */
function f(){
       trace(k);
}
f(); // Выводит "Это не локальная переменная"
```

Теперь создадим внутри функции локальную переменную, имя которой совпадает с именем не локальной переменной:

```
var k = "Это не локальная переменная"; /* Создаем переменную вне тела
функции */
function f(){
       var k = "Это локальная переменная"
       trace(k);
}
f(); // Выводит "Это локальная переменная"
```

Таким образом, мы видим, что при разрешении ссылок, осуществляемых внутри тела функции, приоритет отдается локальной области видимости функции.

Чрезвычайно важно иметь в виду, что идентификатор this при использовании его в теле функции возвращает ссылку на объект, содержащий объявление данной функции, вне зависимости от того, откуда осуществляется ее вызов. Рассмотрим следующую функцию:

```
function moveToPosition(x, y){
this._x=x;
this._y= y;
}
```

Функция moveToPosition() перемещает объект, на монтажной линейке которого находится ее объявление в заданную с помощью параметров точку. Важно отметить, что результат действия этой функции зависит от того, где находится ее объявление, поскольку обращение к свойствам _x и _y, отвечающим за координаты объекта, выполняется с помощью идентификатора this. Так, если поместить код этой функции в кадр основной монтажной линейки, то ее вызов сместит в заданную точку все содержимое кадра:

```
moveToPosition(300, 200); /* Левый верхний угол рабочей области теперь
находится в точке (300; 200) */
```

Для того чтобы идентификатор this в теле функции ссылался не на объект, содержащий объявление функции, а на другой объект, можно использовать методы Function.call() или Function.apply(), рассмотренные ранее в *данной главе*.

Если внутри функции неявным образом создается переменная (то есть без использования предложения var), то эта переменная автоматически прикрепляется к монтажной линейке объекта, содержащего объявление функции. Это можно легко проверить следующим образом:

```
function f(){
       k = "Это не локальная переменная";
}
f();
trace(k); // Выводит "Это не локальная переменная"
```

Данный пример показывает, что переменная, созданная в теле функции без помощи предложения var, остается доступной даже после окончания работы функции. Это оказывается возможным, поскольку данная переменная прикрепляется к монтажной линейке, содержащей функцию, как если бы она была просто на ней объявлена. Если бы при создании переменной в теле функции использовалось предложение var, мы получили бы локальную переменную, которая была бы удалена после выполнения функции:

```
function f(){
       var k = "Это локальная переменная";
}
f();
trace(k); // Выводит undefined, так как переменная локальна
```

Доступ к функции

Несмотря на то, что объявление функции находится на одной монтажной линейке, ее можно вызвать с другой монтажной линейки, используя адресацию точно так же, как это происходит с переменными. Функция, объявленная на одной монтажной линейке, может быть вызвана с другой монтажной линейки с помощью точечного синтаксиса или с помощью оператора `[]`.

Так, допустим, на основной монтажной линейке находится клип `mc` и здесь же объявлена функция `myFunc()`, принимающая три параметра `x`, `y` и `z`. Для того чтобы с монтажной линейки клипа `mc` вызвать функцию `myFunc()`, необходимо использовать один из следующих видов записи:

```
_root.myFunc(x, y, z); // Абсолютная адресация
_root["myFunc"](x, y, z); /* Абсолютная адресация. Применение оператора
"[]" */
_parent.myFunc(x, y, z); // Относительная адресация
_parent["myFunc"](x, y, z); /* Относительная адресация. Применение
оператора "[]" */
```

Примечание

Необходимо четко понимать, что областью видимости функции (помимо ее локальной области видимости) является монтажная линейка, на которой объявлена эта функция. Это справедливо даже в том случае, если вызов функции осуществляется с другой монтажной линейки.

ActionScript предоставляет возможность создания глобальных функций, непосредственно доступных из сценария любой монтажной линейки. Для создания глобальной функции необходимо создать глобальную переменную и сохранить в ней ссылку на функцию. После этого данную функцию можно вызвать из любого места фильма без необходимости применения адресации.

```
function myGlobalFunction(p1, p2, ..., pN){ // Объявляем функцию
...;
}
_global.func = myGlobalFunction; /* Сохраняем ссылку на функцию
в глобальной переменной */
func(p1, p2, ..., pN); // Вызываем глобальную функцию
```

Глобальную функцию также можно создать с помощью литерала функции следующим образом:

```
_global.func = function(p1, p2, ..., pN){
...;
}
```

При создании глобальных функций следует иметь в виду следующие важные особенности.

- Если на монтажной линейке, содержащей вызов глобальной функции, имеется объявление функции с тем же именем, локальная функция перекроет глобальную. В этом случае вызов глобальной функции должен начинаться с идентификатора `_global`.
- В глобальной функции идентификатор `this` дает ссылку на объект `_global`. Для того чтобы `this` ссылался на другой объект, необходимо использовать методы `Function.call()` или `Function.apply()`.

Время жизни функции

Время жизни функции, как и время жизни переменной, ограничено временем существования монтажной линейки, на которой объявлена данная функция. При удалении из документа клипа, содержащего функцию, эта функция перестает быть доступной. Соответственно при выгрузке из проигрывателя SWF-документа все функции, принадлежащие элементам этого документа, удаляются.

Исключение составляют только функции, помещенные в `_global`. Глобальные функции продолжают быть доступными в течение всего времени работы проигрывателя Flash Player, начиная с момента их объявления.

Рекурсивные функции

Рекурсивной функцией является функция, которая вызывает саму себя. Иными словами, рекурсия имеет место, когда в теле функции содержится ее вызов. Для того чтобы функция вызывала себя саму, в ее теле можно использовать либо явный вызов, включающий идентификатор функции и оператор `()`, либо воспользоваться свойством `callee` объекта `arguments`, возвращающим ссылку на текущую функцию. Данное свойство обсуждалось в соответствующем разделе *настоящей главы*. При размещении вызова функции в ее теле теоретически выполнение функции продолжается либо до выполнения определенного условия, инициирующего завершение работы функции с помощью предложения `return`, либо до бесконечности. В действительности проигрыватель Flash Player допускает лишь 255 шагов рекурсии (то есть 255 вызовов функции самой собой), после чего выводит сообщение об ошибке, представленное на рис. 8.2.

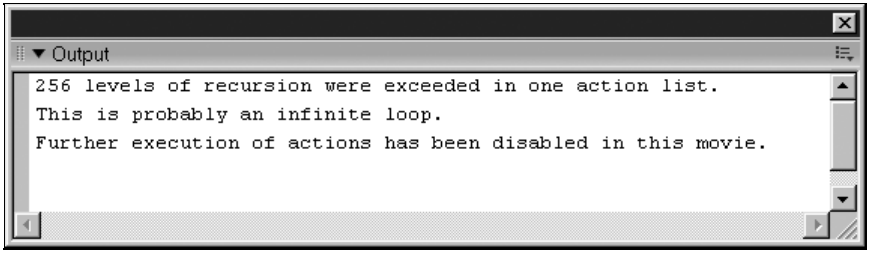


Рис. 8.2. Сообщение о превышении допустимого количества уровней рекурсии

Классический пример рекурсивной функции — вычисление факториала числа. Листинг 8.6 содержит функцию, вычисляющую факториал некоторого числа. Факториал числа 0 равен 1 — этот факт учитывается в условном предложении. Поскольку факториал вычисляется только для неотрицательных целых чисел, проводится проверка того, является ли аргумент функции допустимым значением. Для проверки дробности используется функция `indexOf()`, позволяющая определить наличие оператора десятичной точки. Если точка отсутствует, функция `indexOf()` возвращает значение −1.

Листинг 8.6. Использование рекурсивной функции для вычисления факториала числа

```
function factorial(n:Number){
        //Если аргумент равен нулю, факториал равен 1
        if(n==0){
        return 1;
        }
        /*Если аргумент дробное (содержит десятичную точку) или
        отрицательное число, факториал не вычисляется*/
        else if (String(n).indexOf(".")!=-1||n<0){
                return undefined;
        }
        /*Если аргумент положительное целое число, не меньше 1,
        осуществляется вычисление с использованием рекурсии*/
        else if (n>=1){
                return n*arguments.callee(n-1)};
        }
// вызов функции
trace(factorial(5));//Выводит 120
```

В листинге 8.7 приводится пример функции `findAllChars()`, осуществляющей поиск в заданной строке индексов всех вхождений требуемого символа или сочетания символов. Функции передается два или три параметра: `string` — строка, в которой выполняется поиск, и `charToFind` — символ или набор символов, индекс вхождения которых требуется найти. Третий параметр является необязательным, поэтому он не указан как формальный параметр. Данный параметр позволяет задать индекс, начиная с которого будет осуществляться поиск. Напомним, что индексация символов в строке начинается с нуля. Принцип работы функции состоит в следующем. Если третий параметр не передается функции при вызове, мы искусственно добавляем его со значением 0 в массив параметров `arguments`. Далее мы вводим локальную переменную `index`, используемую для хранения промежуточных результатов поиска, то есть для индекса вхождения искомого символа, полученного на каждой итерации. Для поиска мы используем функцию `indexOf()`, подробно рассмотренную в *разд. "Встроенные свойства и методы для работы со строками" главы 5*. Теперь остается рекурсивно вызывать функцию `findAllChars()`, при каждом вызове в ка-

честве третьего параметра передавая ей индекс символа, следующего за очередным найденным символом. Таким образом мы проходим всю строку до тех пор, пока вхождения искомого символа не будут исчерпаны и функция `indexOf()` не выдаст значение −1. Когда это произойдет, выполнится блок предложения `else`, содержащий предложение `return`, и функция завершит свою работу.

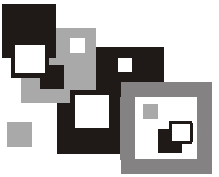
Примечание

Данная функция не будет работать, если исходная строка содержит более 254 вхождений искомого символа. Это связано с ограничением глубины рекурсии, установленным в проигрывателе Flash Player.

Листинг 8.7. Рекурсивный поиск индексов всех вхождений заданного набора символов в строку

```
function findAllChars(string:String, charToFind:String):Number {
/*Если передано только два параметра, сами добавляем третий параметр,
задающий индекс, с которого будет начинаться поиск, и присваиваем ему
значение 0 */
       if (arguments.length == 2) {
              arguments[2] = 0;
// Локальная переменная для хранения индекса вхождения символа
              var index = 0;
       }
// сохраняем индекс очередного вхождения символа
       index = string.indexOf(charToFind, arguments[2]);
/* Если искомый символ обнаружен, выводим индекс его вхождения
и рекурсивно вызываем функцию, установив начальный индекс поиска
на единицу большим, чем индекс обнаруженного вхождения */
       if (index != -1) {
              trace(index);
              arguments.callee(string, charToFind, index + 1);
       }
// Если искомый символ не обнаружен, прекращаем действие функции
       else {
              return;
       }
}
//Вызов функции
findAllChars("Параллелограмм", "л"); // Выводит 4, 5, 7
findAllChars("Параллелограмм", "лл"); // Выводит 4
var str = "Vita brevis ars longa";
findAllChars(str, " "); // Выводит 4, 11, 15
```

Глава 9



Массивы

В любом наборе исходных данных самая надежная величина,
не требующая никакой проверки, является ошибочной.

Третий закон Финэйгла

Перед разработчиком очень часто возникает необходимость в моделировании множеств, состоящих из элементов, значения которых могут обрабатываться единым образом. Такая задача возникает, например, при обработке множества значений координат точек или при обработке набора записей, содержащих информацию о клиентах фирмы. Для эффективного решения подобных задач используются особые программные объекты, называемые массивами.

Понятие массива

Массив представляет собой упорядоченную структуру данных, состоящую из индексированных элементов, в которой каждый отдельный элемент идентифицируется своим положением относительно начального элемента. Значения, хранящиеся в массиве, называются *элементами* массива. Каждый элемент массива имеет свой уникальный числовой индекс, определяющий порядковый номер этого элемента относительно самого первого элемента массива. Поскольку индекс уникален для каждого элемента, с его помощью можно осуществлять обращение к элементу. Важно отметить, что элементами массива в ActionScript могут быть данные любого типа, например, числа, строки, объекты или даже другие массивы; кроме того, один массив может содержать данные разных типов. Массив может быть использован для хранения координат набора точек, списка клиентов фирмы, набора ссылок на клипы основной монтажной линейки и т. п. Применение массивов чрезвычайно удобно при работе с группами однотипных взаимосвязанных данных.

Для создания и работы с массивами в ActionScript имеется специальный встроенный класс `Array`. Любой массив представляет собой объект, являющийся экземпляром этого класса. Класс `Array` включает одно свойство и ряд встроенных методов, предоставляющих возможность выполнения наиболее часто используемых процедур обработки массивов. Поскольку любой массив создается на базе класса `Array`, он автоматически получает доступ к его свойству и методам.

Следует отметить, что массив, являясь объектом, принадлежит к ссылочным данным и передается по ссылке, а не по значению. Это означает, что при сохранении массива в переменной сохраняется только ссылка на него, сами данные с описанием массива не копируются в переменную.

Фактически массив представляет собой контейнер, который содержит набор элементов. Каждый элемент массива имеет свой уникальный индекс, с помощью которого можно осуществлять обращение к этому элементу. Нумерация элементов массива начинается с нуля, то есть первый элемент всегда имеет индекс 0, второй — 1, третий — 2, и т. д. Элементы в массиве не обязательно должны следовать подряд друг за другом. Возможна ситуация, когда элементы располагаются с промежутком, например, в массиве могут находиться элементы с индексами 0, 1, 2 и затем только 120, при этом длина массива будет составлять 121 элемент, а все промежуточные элементы (начиная с элемента с индексом 3 и вплоть до элемента с индексом 119) являются пустыми и принимают значение `undefined`.

Используя индекс элемента, можно извлекать его значение, присваивать значение данному элементу либо выполнять с ним определенные действия. Так, например, если массив в качестве элементов содержит ссылки на клипы, то, обращаясь к его элементам, можно изменять местоположение или другие параметры соответствующих клипов. Это дает мощные средства разработки интерактивности или создания программной анимации.

Создание массива

ActionScript предоставляет несколько различных способов создания массива. Как правило, при создании массива ссылка на него сохраняется в переменной, для того чтобы в дальнейшем к элементам этого массива можно было бы обратиться по имени данной переменной. При сохранении ссылки на массив в переменной она получает тип `Array`, определяемый классом `Array`.

Конструктор массива

Поскольку каждый конкретный массив — это объект, являющийся экземпляром класса `Array`, как и большинство экземпляров классов, он может быть создан с помощью оператора `new` и конструктора `Array()`. Результат создания массива с помощью конструктора зависит от числа и значения передаваемых ему параметров. Рассмотрим все возможные варианты.

Создание пустого массива

Пустой массив — это массив, не содержащий элементов. Для создания пустого массива параметры не передаются. При этом используется следующий синтаксис:

```
var массив:Array = new Array();
```

Создание массива выполняется в правой части выражения присваивания. Ссылка на созданный массив сохраняется в переменной с идентификатором *массив*. Поскольку изначально созданный таким образом массив не содержит элементов, они могут быть добавлены в него позднее.

Создание массива заданного размера

Если конструктору `Array()` в качестве единственного параметра передается число или переменная, содержащая числовое значение, то создается массив, состоящий из соответствующего числа пустых элементов, то есть элементов со значением `undefined`. В этом случае используется следующий синтаксис:

```
var массив:Array = new Array(количество элементов);
```

Здесь *количество элементов* — это число, указывающее число пустых элементов в создаваемом массиве. При создании массива этим способом его свойство `length`, указывающее количество элементов массива, принимает значение, заданное параметром `количество элементов`.

```
var l:Number = 25; // Числовая переменная
var ar:Array = new Array(l); // Создаем массив из 25 пустых элементов
trace(ar.length); // Выводит 25
```

Создание массива с набором элементов

Конструктор позволяет при создании массива сразу же заполнить его набором элементов. Если конструктору передается более одного числового параметра или любое количество нечисловых параметров, то значения этих параметров помещаются в массив в качестве элементов. При этом используется следующий синтаксис:

```
var ar:Array = new Array(элемент1, элемент2, ......, элементN);
```

Здесь параметры *элемент1*, *элемент2*, ..., *элементN* — это литералы данных или переменные, содержащие значения, размещаемые в массиве в качестве его элементов.

Следующий код создает массив, состоящий из трех элементов, имеющих строковые значения: `"Дима"`, `"Лена"`, `"Марк"`.

```
var ar:Array = new Array("Дима", "Лена", "Марк");
```

Литерал массива

В некоторых случаях для большей наглядности или для более компактной записи бывает удобно воспользоваться литералом массива, представляющим собой буквальное описание массива. Литерал массива имеет следующий синтаксис:

```
[элемент1, элемент2, ......, элементN]
```

Здесь *элемент1*, *элемент2*, ..., *элементN* — это литералы данных или переменные, содержащие значения, размещаемые в массиве в качестве его элементов.

Литерал массива, как и в предыдущих случаях, может быть сохранен в переменной (сохраняется не сам массив, а только ссылка на него), для того чтобы к массиву можно было обратиться в дальнейшем.

```
var ar:Array = ["Дима", "Лена", "Марк"];/* Создаем массив из
3-х элементов и сохраняем ссылку на него в переменной ar */
```

При необходимости создания нового пустого массива бывает удобно использовать литерал:

```
var ar:Array = [];
```

Глобальная функция *Array()*

Применение глобальной функции `Array()` практически идентично применению конструктора `Array()`, за исключением того, что для создания нового массива не нужно использовать оператор `new`.

Функция `Array()` позволяет создать новый пустой массив или преобразовать переданные ей параметры в элементы нового массива. Данная функция имеет следующий синтаксис:

```
Array(элемент1, элемент2, ......, элементN);
```

Здесь параметры *элемент1*, *элемент2*, ..., *элементN* — это элементы, которые должны быть помещены в массив. Если при вызове функции `Array()` параметры отсутствуют, то создается пустой массив. Ссылка на массив, созданный с помощью функции `Array()`, может быть сохранена в переменной так же, как и в случае с использованием конструктора.

```
var ar:Array = Array(элемент1, элемент2, ......, элементN);
```

Функция `Array()` может быть использована как функция преобразования для преобразования типа передаваемого ей значения в тип `Array`. Это иллюстрируется в следующем примере.

```
var ar:String="hello";
trace(ar instanceof Array ); // Выводит false
ar = Array(ar); // Преобразуем тип переменной ar
trace(ar instanceof Array ); // Выводит true
```

Используемый здесь оператор `instanceof` осуществляет проверку, соответствует ли тип его левого операнда типу, заданному с помощью правого операнда. Если имеет место соответствие типов, `instanceof` возвращает значение `true`, если типы не совпадают, `instanceof` возвращает значение `false`.

Для выполнения преобразования типа `Array` в другие типы можно использовать соответствующие функции преобразования. В табл. 9.1 представлены результаты преобразования типа данных `Array` в основные элементарные типы.

Таблица 9.1. *Результат преобразования типа данных `Array` в элементарный тип*

Исходный тип данных	Тип, к которому происходит преобразование		
	Number	**String**	**Boolean**
Массив	`NaN`	Строка из строковых значений элементов, разделенных запятыми	`true`

Вывод массива в панель *Output*

При передаче ссылки на массив в качестве аргумента функции `trace()` в панель **Output** будет выведено его строковое представление, то есть перечень всех элементов массива, разделенных запятыми и расположенных в строке в соответствии с их порядковым номером (индексом).

```
trace(["A", "B", "C", "D", "E"]) //Выводит A, B, C, D, E
var myArray:Array=[1, 2, 3, 4, 5];
trace (myArray); //Выводит 1, 2, 3, 4, 5
```

Для того чтобы вывести значение конкретного элемента, необходимо в качестве аргумента функции `trace()` использовать обращение к этому элементу.

Обращение к элементам массива

Для обращения к элементу массива используется имя переменной, содержащей ссылку на массив, и селектор, представляющий собой индекс элемента, к которому требуется обратиться, заключенный в квадратные скобки. Обращение к элементу массива имеет следующий синтаксис:

массив[индекс]

Здесь *массив* — это имя переменной, содержащей ссылку на массив, `индекс` — это число или выражение, возвращающее число, указывающее индекс элемента, к которому идет обращение.

Создадим массив из четырех элементов:

```
var myArray:Array = new Array("A", "B", "C", "D");
```

Для того чтобы вывести значение второго элемента в панель **Output**, используем следующее предложение:

```
trace(myArray[2]); // Выводит C
```

Напомним, что индексация элементов массива начинается с нуля.

Для того чтобы вывести значение всех элементов массива, можно было бы написать еще три аналогичных строки кода, однако такой подход является чрезвычайно неэффективным. При необходимости обработки нескольких элементов массива значительно удобнее прибегнуть к использованию цикла:

```
for (i=0; i<4; i++){
        trace(myArray[i]);
}
```

В результате выполнения этого сценария в панели **Output** выводятся значения всех элементов массива:

```
A
B
C
D
```

Обратите внимание, что в проверочном выражении цикла в качестве ограничителя значения счетчика используется значение длины массива.

Если требуется обработать все элементы массива, можно также прибегнуть к циклу `for.....in`. Данный цикл не требует явного указания длины массива, поскольку его счетчик сам выполнит проход по всем элементам. Правда, в этом случае элементы обрабатываются, начиная с конца массива. Продолжая предыдущий пример, рассмотрим применение цикла `for.....in`:

```
for (i in myArray){
       trace(myArray[i]);
}
```

В результате в панели **Output** появится следующее:

```
D
C
B
A
```

Поскольку индекс элемента массива может быть задан в виде любого выражения, возвращающего числовое значение, можно динамически генерировать индекс для обращения к требуемому элементу:

```
trace(myArray[90/45+1]); // Выводит D, значение элемента с индексом 3
var x:Number = 12;
var y:Number = 6;
trace(myArray[(x-y)/y]); // Выводит B, значение элемента с индексом 1
```

Операции над элементами массива

При работе с данными, хранящимися в массиве, необходимо осуществлять модификацию их значений, добавлять, упорядочивать, сортировать и удалять данные. Для выполнения наиболее часто используемых, стандартных операций класс `Array` предусматривает ряд возможностей, которые будут рассмотрены в настоящем разделе.

Определение количества элементов в массиве

Количество элементов, содержащихся в массиве, называется размером или длиной массива. Поскольку индексация элементов массива начинается с нуля, длина массива всегда на единицу больше значения индекса его последнего элемента.

Свойство *Array.length*

Это единственное свойство, определенное в классе `Array`. Свойство `length` указывает количество элементов в массиве. Для обращения к этому свойству, как и к свойству любого объекта, используется точечный синтаксис или оператор `[]`. Значение, возвращаемое свойством `length`, всегда на единицу больше индекса послед-

него элемента массива. Количество элементов в любом массиве может быть определено с помощью одного из следующих вариантов синтаксиса:

```
массив.length
массив["length"]
```

Здесь *массив* — это идентификатор массива.

```
var myArray:Array = new Array("A", "B", "C", "D");
myArray.length; // Возвращает 4
```

Свойство `length` можно использовать не только для определения длины массива, ему также можно присваивать числовое значение, что позволяет добавлять в массив пустые элементы или удалять имеющиеся.

Если свойству `length` присваивается значение, меньшее, чем количество элементов массива, то его длина уменьшается и соответствующее количество элементов удаляется, начиная с конца массива.

```
myArray.length = 2;
trace(myArray); // Выводит A, B
trace(myArray.length); // Выводит 2
```

Если свойству `length`, наоборот, присваивается значение, большее, чем количество элементов массива, то длина массива увеличивается, а свободные места заполняются пустыми элементами (элементами со значением `undefined`). Поместим следующие строки после приведенного кода:

```
myArray.length = 4;
trace(myArray); // Выводит A, B, undefined, undefined
trace(myArray.length); // Выводит 4
```

Свойство `length` часто применяется в проверочном выражении циклов `for` или `while` при необходимости перебора элементов массива. Это позволяет обработать все элементы массива, не зная заранее его размер. Следующий сценарий выводит в панель **Output** значения всех элементов массива `someArray` вне зависимости от его длины.

```
for (i=0;i<someArray.length; i++){
       trace(someArray[i]);
}
```

Присваивание значения элементу массива

Присваивание значения элементу массива осуществляется так же, как и присваивание значения переменной. Слева от оператора присваивания находится элемент массива, справа — присваиваемое ему значение в виде литерала или выражения. Отметим, что разные элементы одного и того же массива могут иметь значения различных типов.

В следующих предложениях выполняется присваивание значений разных типов элементам массива multiArray:

```
multiArray[0] = 3.14; // Число
multiArray[1] = "Dmitri I. Albert"; // Строка
multiArray[2] = true; // Булев тип
multiArray[3] = gotoAndStop; // Функция
multiArray[4] = [1, 2, 3, 4, 5]; // Массив
multiArray[5] = {}; // Объект
```

Если значение присваивается уже существующему элементу массива, его прежнее значение будет переопределено:

```
multiArray[0] = 3.14;
multiArray[0] = 2.72;
trace(multiArray[0]); // Выводит 2.72
```

Если значение присваивается несуществующему элементу, то есть элементу, индекс которого превышает значение length-1, то этот элемент будет создан автоматически. При этом произойдет увеличение длины массива, в который будет добавлено соответствующее количество пустых элементов, и обновление значения свойства length. В этом легко убедиться на примере:

```
var let:Array = new Array("a", "b", "c");
trace(let.length); // Выводит 3
let[25] = "z";
trace(let.length); // Выводит 26
```

Все элементы с индексами 3—24 включительно являются пустыми, то есть содержат значение undefined. Для того чтобы проверить это, можно вывести массив в панель **Output**:

```
trace(let); // Выводит a, b, c, undefined, ......, undefined, z
```

Добавление элементов в массив

Добавление элементов в массив может быть выполнено различными способами. Часть из них уже была рассмотрена, остальные реализуются с помощью методов класса Array.

Следует еще раз отметить, что при добавлении в массив новых элементов его длина автоматически увеличивается, предоставляя возможность для размещения любому количеству элементов.

Присваивание значения свойству *length*

Для того чтобы добавить любое количество пустых элементов в конец массива, то есть после элемента, индекс которого имеет значение length-1, можно присвоить свойству length данного массива значение, равное length+N, где N — это количество добавляемых элементов. Такая возможность рассматривалась ранее.

Присваивание значения элементу массива

Добавить новый элемент в массив можно также, присвоив значение элементу, индекс которого превышает значение `length-1`. Этот подход был также рассмотрен ранее.

Заполнение массива в цикле

Если в массив требуется поместить большой набор данных, значения которых подчиняются определенной закономерности, это очень удобно выполнять с помощью цикла. В следующем фрагменте кода создается массив `alphabeth`, который в цикле заполняется символами прописных букв русского алфавита. Следует иметь в виду, что в кодировке Unicode символы прописных букв русского алфавита начинаются с кода 1040 и следуют подряд за исключением буквы Ё, код которой равен 1025. В связи с этим для простоты мы исключим эту букву и, таким образом, длина заполненного массива составит 32, а не 33 элемента.

```
var alphabeth:Array = new Array(); //Создаем пустой массив
var alphabethLength:Number = 32; //Количество элементов массива
for (i=0; i<alphabethLength; i++){ //Заполнение массива
alphabeth[i] = String.fromCharCode(1040+i);
}
```

Аналогично можно поместить в массив экземпляры клипов, имена которых содержат порядковый номер. В следующем примере создается массив `clips`, который в цикле заполняется динамически созданными экземплярами символа типа MovieClip с идентификатором связи (linkage) `mc`. На этот раз при создании нового пустого массива воспользуемся его литералом. Для создания экземпляров здесь используется глобальная функция `attachMovie()`, подробно описанная в *главе 12*. Обратите внимание, что создание экземпляров и размещение их в массиве последовательно выполняется в одном и том же цикле.

```
var num:Number = 25;// Количество создаваемых экземпляров
var clips:Array = []; //Массив для экземпляров клипов
for (i=0; i<num; i++){
attachMovie("mc", "mc"+i, i);// Создание экземпляров
clips[i]=this["mc"+i]; //Добавление экземпляров в массив
}
```

Метод *Array.push()*

Данный метод добавляет новые элементы в конец массива и возвращает обновленное значение его свойства `length`.

Метод `push()` имеет следующий синтаксис:

массив.push(*элемент1*, *элемент2*, ..., *элементN*);

Здесь *массив* — это идентификатор массива, в который добавляются элементы.

Параметры *элемент1*, *элемент2*, ..., *элементN* — это элементы, которые добавляются в конец массива. Данный метод позволяет добавить в массив любое количество элементов (в том числе и нулевое). Параметры могут иметь любой тип и задаваться как литералы данных или выражения. При передаче методу push() выражений в качестве параметров, они вычисляются, и результаты вычислений помещаются в массив в качестве новых элементов. Если при вызове метода push() параметры отсутствуют, то никаких изменений в массиве не происходит.

```
var myArray:Array = ["a", "b", "c", "d"];
myArray.push("e", "f", String.fromCharCode(103));
trace(myArray.length); //Выводит 7
trace(myArray); //Выводит a,b,c,d,e,f,g
```

После добавления элементов в массив метод push() возвращает новую длину массива.

```
var newLength = myArray.push("h");
trace(newLength); // Выводит 8
```

Метод *Array.unshift()*

Метод unshift() действует очень похоже на предыдущий метод push(), но в отличие от него добавляет новые элементы не в конец массива, а в начало. При этом элементы, расположенные в начале массива, изменяют свои индексы и "сдвигаются" к концу массива, освобождая место для новых элементов. Как и предыдущий метод, метод возвращает новую длину массива.

Метод unshift() имеет синтаксис, похожий на синтаксис метода push():

массив.unshift(*элемент1*, *элемент2*, ..., *элементN*);

Здесь *массив* — это идентификатор массива, в который добавляются элементы.

Параметры *элемент1, элемент2, ..., элементN* — это элементы, которые добавляются в начало. Параметры могут иметь любой тип и задаваться как литералы данных или выражения. При передаче методу unshift() выражений в качестве параметров они вычисляются, и результаты вычислений помещаются в массив в качестве новых элементов. Если при вызове метода unshift() параметры отсутствуют, то никаких изменений в массиве не происходит.

```
var myArray:Array = [4, 5, 6, 7];
myArray.unshift(1, 2, "Flash".indexOf("s"));
trace(myArray.length); // Выводит 7
trace(myArray); // Выводит 1,2,3,4,5,6,7
```

Важно отметить, что применение метода unshift() увеличивает индексы существующих элементов, "сдвигая" их к концу массива.

Метод *Array.splice()*

Это многофункциональный метод, позволяющий выполнять сразу несколько различных действий с элементами массива. К числу выполняемых этим методом операций относятся следующие:

- удаление элементов из произвольного участка массива;
- добавление новых элементов в произвольный участок массива;
- одновременное удаление произвольного количества элементов, начиная с заданного индекса, и добавление в этот участок массива произвольного количества новых элементов.

Метод `splice()` имеет следующий синтаксис:

массив`.splice(`*индекс*`, `*удаление*`, `*элемент1*`, `*элемент2*`, ....., `*элементN*`)`

Здесь *массив* — это идентификатор массива, в который добавляются или из которого удаляются элементы. Рассмотрим назначение параметров метода `splice()`:

- параметр *индекс* служит для задания индекса элемента массива, начиная с которого (включая его самого) будет выполнено добавление элементов в массив или удаление элементов из массива. Этот параметр задается как число или выражение, возвращающее число. Если значение параметра *индекс* превышает значение индекса последнего элемента массива, то при добавлении элементов они будут просто добавлены в конец массива, а при удалении массив останется без изменений;
- параметр *удаление* задает количество элементов массива, которые должны быть удалены. Удаление элементов выполняется, начиная с элемента с индексом *индекс*, включая данный элемент. Этот параметр задается как число или выражение, возвращающее число. Если значение параметра *удаление* равно нулю, удаления элементов не происходит. Если этот параметр опущен, происходит удаление всех элементов массива, начиная с элемента с индексом *индекс*;
- параметры *элемент1*, *элемент2*, ..., *элементN* определяют элементы, которые должны быть добавлены в массив. Добавление элементов происходит, начиная с индекса, заданного параметром *индекс*. При этом индексы всех последующих элементов массива увеличиваются в соответствии с количеством добавляемых элементов. Следует отметить, что количество удаляемых и добавляемых элементов может и не совпадать. Если добавлять новые элементы в массив не требуется, данные параметры не указываются.

По окончании работы метод `splice()` возвращает измененный массив.

Рассмотрим несколько примеров, иллюстрирующих применение данного метода:

Создадим массив из восьми элементов:

```
var myArray:Array = ["a", "b", "c", "d", "e", "f", "g", "h"];
```

Удалим из него три элемента, начиная с элемента `"e"`, имеющего индекс 4:

```
myArray.splice(4, 3);
```

Теперь массив состоит из следующих элементов:

```
["a", "b", "c", "d", "h"]
```

В этом легко убедиться с помощью функции trace():

```
trace(myArray); // Выводит a,b,c,d,h
```

Теперь добавим в массив три новых элемента, поместив их между элементами "b" и "c" с индексами 1 и 2 соответственно:

```
myArray.splice(1, 0, "Э", "Ю", "Я");
trace(myArray); // Выводит a,Э,Ю,Я,b,c,d,h
```

А теперь удалим только что добавленные элементы, заместив их другими:

```
myArray.splice(1, 3, "1", "2", "3", "4", "5");
trace(myArray); // Выводит a,1,2,3,4,5,b,c,d,h
```

И, наконец, вернем массив в первоначальное состояние. Поскольку для этого нужно удалить элементы с индексами 1—5 и добавить новые элементы в другой участок массива, удалим 11 элементов, начиная с индекса 1, и добавим вместо них 6 новых:

```
myArray.splice(1, 11, "b", "c", "d", "e", "f", "g");
trace(myArray); // Выводит a,b,c,d,e,f,g
```

Метод *Array.concat()*

Этот метод позволяет добавлять произвольное количество элементов в конец массива, а также объединять несколько массивов, создавая новый массив.

Метод concat() имеет следующий синтаксис:

массив.concat(*элемент1, элемент2, ..., элементN*)

Здесь *массив* — это идентификатор исходного массива.

Параметры *элемент1, элемент2, ..., элементN* — это данные, которые добавляются в массив в качестве элементов. Параметрами метода concat() могут быть другие массивы. В этом случае массив, возвращаемый методом concat(), будет состоять из элементов исходного массива и элементов массивов, переданных ему в качестве параметров.

Результатом выполнения метода concat() является новый массив. Исходный массив и массивы, переданные данному методу в качестве параметров, остаются без изменений. Все рассмотренные выше методы класса Array непосредственно изменяли сам исходный массив.

Если метод concat() вызывается без передачи параметров, он возвращает копию исходного массива.

Создадим два массива:

```
var letters = ["A", "B", "C", "D"];
var figures = [1, 2, 3];
```

Для того чтобы добавить новые элементы в конец массива с помощью метода concat(), необходимо поступить так, как показано в следующем примере:

```
letters=letters.concat("E", "F");
trace(letters); // Выводит A,B,C,D,E,F
```

Поскольку метод не изменяет исходный массив, а возвращает новый массив, состоящий из элементов исходного массива, и элементов, переданных в качестве параметров, здесь применяется операция присваивания. Полученный таким образом результат эквивалентен применению метода `push()`.

Теперь попробуем объединить два массива и сохранить ссылку на массив, полученный в результате объединения в переменной `mixed`:

```
var letters = ["A", "B", "C", "D", "E", "F"];
var figures = [1, 2, 3];
var mixed = letters.concat(figures);
trace(mixed); // Выводит A,B,C,D,E,F,1,2,3
```

Можно убедиться в том, что исходные массивы остались без изменений:

```
trace(letters); // Выводит A,B,C,D,E,F
trace(figures); // Выводит 1,2,3
```

Аналогично выполняется объединение трех и более массивов. Результат в виде нового массива может быть сохранен в переменной:

```
var let1 = ["A", "B", "C"];
var let2 = ["D", "E", "F"];
var let3 = ["G", "H", "I"];
var letters = let1.concat(let2, let3);
trace(letters); // Выводит A,B,C,D,E,F,G,H,I
```

Если одним из элементов объединяемого массива является массив (вложенный массив), то после объединения в новом массиве данный элемент не изменяется. Таким образом, метод `concat()` не разбивает элементы вложенных массивов.

```
var let1 = ["A", "B", "C"];
var let2 = ["D", [1, 2, 3], "F"];
var letters = let1.concat(let2);
trace(letters[3]); // Выводит D
trace(letters[4]); /* Выводит 1, 2, 3, так как пятый элемент массива
letters - это тоже массив */
```

Обратиться к элементу вложенного массива можно следующим образом:

```
trace(letters[4][0]) // Выводит 1
```

Подробнее о вложенных массивах будет сказано в *разд. "Многомерные массивы" в конце данной главы.*

Строковое представление массива

Класс `Array` содержит два метода `toString()` и `join()`, позволяющие сформировать строку, состоящую из элементов исходного массива. Полученная таким образом строка в дальнейшем может быть использована для различных целей. Следует отметить, что оба рассматриваемых метода не изменяют исходный массив.

Метод *Array.toString()*

Этот метод создает строковые представления для каждого элемента массива (включая пустые элементы со значением `undefined`) и конкатенирует их в единую строку, где строковые представления разных элементов разделяются запятыми. Полученная таким образом строка возвращается в качестве результата. Следует отметить, что метод `toString()` не изменяет исходный массив, а просто формирует строку на основании значений его элементов. Данный метод используется в ситуациях, требующих преобразования данных, имеющих тип `Array` в тип `String`.

Метод имеет следующий синтаксис:

массив.toString()

Здесь *массив* — это исходный массив.

Метод .toString() не имеет параметров.

Рассмотрим несколько примеров:

```
var nArray = [1, 2, 3, 4, 5];
nArray.toString()// Возвращает строку "1, 2, 3, 4, 5"
var bArray = [true, true, false, true, false];
var b = bArray.toString(); // Сохраняем строку в переменной b
trace(b); // Выводит true,true,false,true,false
```

Теперь создадим массив, состоящий из трех пустых элементов:

```
var ar = new Array(3);
var str = ar.toString();
trace(str); // Выводит undefined,undefined,undefined
```

Отметим, что вложенные массивы тоже разбиваются на элементы, которые в результирующей строке разделяются запятыми:

```
var nested = [1, 2, 3, ["a", "b", "c"], 4, 5];
nested.toString(); //Возвращает строку "1,2,3,a,b,c,4,5"
```

Метод *Array.join()*

Метод `join()` выполняет практически ту же задачу, что и метод `toString()`, однако действует более гибко, поскольку позволяет задать разделитель, с помощью которого строковые значения исходного массива будут отделяться друг от друга в результирующей строке. Как и метод `toString()`, метод `join()` не изменяет исходный массив. Метод `join()` имеет следующий синтаксис:

массив.join(*разделитель*)

Здесь *массив* — это исходный массив.

Параметр *разделитель* — это строка, или выражение, возвращающее строку, состоящее из одного или нескольких символов, с помощью которых значения элементов массива будут разделены в результирующей строке. Если параметр *разделитель*

опущен, то в качестве разделителя будут использованы запятые, это даст результат, аналогичный результату применения метода `toString()`.

```
var userInfo = ["Петров", "812-876-87-99", "petrov@home.ru"];
trace(userInfo.join(" | ")); // Петров | 812-876-87-99 | petrov@home.ru
```

Разделитель может быть передан методу `join()` с помощью идентификатора переменной:

```
var ar = [1, 2, 3, 4, 5];
var del = "-->";
trace(ar.join( del )); // Выводит 1-->2-->3-->4-->5
```

При необходимости в качестве разделителя может быть использован управляющий символ. Так, например, если требуется вывести элементы массива в панель **Output** так, чтобы каждый элемент размещался на отдельной строке, в качестве разделителя можно использовать Escape-последовательность символа перевода строки `\n`:

```
trace(ar. join("\n"));
```

Эта строка кода выводит в панель **Output** следующее:

```
1
2
3
4
5
```

Следует отметить, что при работе с массивами, содержащими вложенные массивы, элементы вложенных массивов в результирующей строке разделяются запятыми вне зависимости от заданного разделителя:

```
var nested = [1, 2, 3, ["a", "b", "c"], 4, 5];
var del = " ";// Пробел в качестве разделителя
trace(nested.join( del )); //Выводит 1 2 3 a,b,c 4 5
```

Сортировка массива

При работе с элементами массива очень часто возникает необходимость в упорядочивании их размещения в соответствии с определенным критерием. Так, например, перед выводом элементов массива, содержащего информацию о клиентах, имеет смысл отсортировать их в алфавитном порядке по фамилии, сумме заказа или иной категории. Это позволит значительно легче обработать представленную информацию. Класс `Array` предоставляет весьма удобные возможности сортировки массивов, позволяющие выполнить наиболее часто встречающиеся задачи.

Метод *Array.reverse()*

Данный метод позволяет инвертировать или обратить порядок следования элементов в массиве. В результате применения метода первый элемент массива становится по-

следним, второй — предпоследним и т. д. Изменение порядка следования элементов, как правило, применяется к уже отсортированному массиву для изменения порядка сортировки. Метод `reverse()` имеет следующий синтаксис:

```
массив.reverse()
```

Здесь *массив* — это исходный массив, элементы которого требуется инвертировать.

Данный метод не принимает параметров. Его возвращаемым значением является измененный массив. Следует иметь в виду, что метод `reverse()` изменяет сам оригинал массива, а не его копию. Следующий пример иллюстрирует возможности данного метода.

```
var myArray:Array = [1, 2, 3, 4, 5, 6, 7];
r=myArray.reverse();
trace (r); // 7,6,5,4,3,2,1
```

Метод *Array.sort()*

Данный метод предоставляет широкие возможности сортировки массивов и позволяет достаточно просто решить наиболее часто встречающиеся задачи, связанные с сортировкой.

Вызов метода *sort()* без параметров

Метод `sort()` выполняет различные действия в зависимости от числа и значения передаваемых ему параметров. Вначале рассмотрим использование этого метода без параметров. Его вызов имеет следующий синтаксис:

```
массив.sort()
```

Здесь *массив* — это исходный массив, элементы которого должны быть отсортированы.

При вызове без параметров метод `sort()` сортирует элементы массива по возрастающей на основании значений Unicode-кодов их строковых представлений. В результате сортировки изменяется порядок следования элементов в исходном массиве. Следует отметить, что элементы массива сортируются, как строки, поэтому при определении порядка следования элементов применяются те же правила, что и при сравнении строк. В связи с этим, например, в результате сортировки массива, состоящего из числовых элементов, элемент со значением 10 будет помещен перед элементом со значением 9, так как строковое значение "10" меньше, чем строковое значение "9". При сортировке также учитывается регистр символов. Если в исходном массиве имеется несколько элементов с одинаковым значением, после сортировки они размещаются последовательно, причем их местоположение относительно остальных элементов определяется в соответствии со строковым представлением их значения так же, как если бы это был один элемент.

Рассмотрим несколько примеров:

```
var letters = ["В", "а", "д", "Б", "г", "д"];
trace(letters.sort());// Выводит Б,В,а,г,д,д
```

Поскольку коды прописных букв предшествуют кодам строчных, первым элементом результирующего массива является "Б", а не "а". Обратите внимание на повторяющийся элемент "д" в исходном и отсортированном массивах.

```
var numbers = [2, 10, 4, 3, 6, 5, 1];
trace(numbers.sort()); //Выводит 1,10,2,3,4,5,6
```

Метод `sort()` без параметров может быть использован в наиболее простых ситуациях, требующих сортировки строк только в возрастающем направлении.

Настройки сортировки метода *sort()*

Для того чтобы реализовать более гибкий подход к сортировке, методу `sort()` могут быть переданы параметры, задающие опции сортировки. Эти параметры могут быть заданы как десятичные представления двоичных чисел или с помощью специальных буквенных констант класса `Array`. При этом используется следующий синтаксис:

массив`.sort(`*опция1* | *опция2* |,.....,| *опция5*`)`

Здесь *массив* — это исходный массив, элементы которого должны быть отсортированы в соответствии с заданными настройками.

Параметры *опция1*, *опция2*,....., *опция5* — это опции сортировки, которые задаются с помощью десятичного представления двоичного 4-битного числа или с помощью специальных констант класса `Array`. При использовании более одной опции параметры необходимо разделять с помощью оператора побитового ИЛИ — `|`.

Если требуется использовать одновременно несколько опций, то значения их параметров необходимо суммировать либо с помощью побитового оператора ИЛИ, либо арифметически. Так, например, вместо передачи выражения, объединяющего три различные опции:

массив`.sort(`*1* | *2* | 4`)`

можно сразу передать число, в котором нулевой, первый и второй биты будут установлены:

```
массив.sort(7)
```

Вместо использования числовых значений параметров опций можно использовать специальные константы — статические свойства класса `Array`. Класс `Array` содержит набор констант, хранящих числовые значения, соответствующие каждой представляемой ими опции. Использование констант приводит к более громоздкой записи, но зато избавляет от необходимости помнить числовые значения всех опций. Кроме того, константы класса `Array` выводятся во всплывающей подсказке по коду, что значительно облегчает процесс их набора. Перечень опций сортировки, их числовых значений и соответствующих им констант класса `Array` приводится в табл. 9.2.

Таблица 9.2. *Встроенные опции сортировки*

Константа	Числовое значение		Назначение
	Десятичное	Двоичное	
Array.CASEINSENSITIVE	1	00001	Сортировка без учета регистра
Array.DESCENDING	2	00010	Сортировка по убыванию
Array.UNIQUESORT	4	00100	Исключение возможности сортировки массива, содержащего повторяющиеся элементы
Array.RETURNINDEXEDARRAY	8	01000	Получение нового массива, состоящего из отсортированных индексов элементов исходного массива
Array.NUMERIC	16	10000	Сортировка числовых значений

Рассмотрим более подробно назначение и принципы работы опций сортировки.

- Опция `Array.CASEINSENSITIVE` позволяет выполнять сортировку элементов массива без учета регистра их символов. Вызов метода `sort()` с этим параметром приводит к изменению исходного массива. Очевидно этот параметр оказывает влияние на сортировку только буквенных элементов. Рассмотрим следующий пример:

```
var letters = ["В", "а", "Б", "е", "г", "д"];
letters.sort(Array.CASEINSENSITIVE);
```

  Теперь массив `letters` имеет следующий вид:

```
["а", "Б", "В", "г", "д", "е"];
```

  Сравните этот результат с результатом вызова метода `sort()` без параметров в приведенном примере.

- Опция `Array.DESCENDING` выполнять сортировку элементов массива по убыванию, в то время как по умолчанию метод `sort()` использует порядок сортировки по возрастанию. Вызов метода `sort()` с этим параметром приводит к изменению исходного массива. Рассмотрим следующий пример:

```
var letters = ["В", "а", "Б", "е", "г", "д"];
letters.sort(Array.DESCENDING);
```

  Теперь массив `letters` имеет следующий вид:

```
["е", "д", "г", "а", "В", "Б"]
```

  По умолчанию сортировка выполняется с учетом регистра. Для того чтобы выполнить сортировку по убыванию и без учета регистра, необходимо объединить обе опции:

```
letters.sort( Array.CASEINSENSITIVE | Array.DESCENDING);
```

Теперь массив letters имеет следующий вид:

```
["е", "д", "г", "В", "Б", "а"]
```

Опции сортировки могут быть также заданы с помощью чисел. Следующие два вызова метода эквивалентны предыдущему.

```
letters.sort(1 | 2);
letters.sort(3);
```

- Опция Array.UNIQUESORT. Если при вызове метода sort() с данным параметром исходный массив содержит два или более элементов с одинаковым значением, то метод sort() не изменяет исходный массив и просто возвращает значение 0. Если все элементы массива уникальны, этот параметр не оказывает влияния на результат сортировки.

```
var letters = ["В", "а", "д", "Б", "е", "г", "д"];
letters.sort(Array.UNIQUESORT); //Возвращает 0
```

Поскольку элемент "д" встречается в исходном массиве дважды, сортировка не выполняется и метод возвращает значение 0. Исходный массив остался без изменений.

Если теперь убрать повторяющийся элемент, результат сортировки будет такой же, как и при вызове метода sort() без параметров.

```
var letters = ["В", "а", "Б", "е", "г", "д"];
letters.sort(Array.UNIQUESORT);
```

Теперь массив letters имеет следующий вид:

```
["Б", "В", "а", "г", "д", "е"]
```

- Опция Array.RETURNINDEXEDARRAY . Метод sort(), вызванный с данной опцией, возвращает не массив, состоящий из отсортированных элементов, а массив, состоящий из индексов элементов исходного массива, отсортированных в соответствии с порядком сортировки данных элементов. Таким образом, в результате возвращается массив, содержащий индексы исходных элементов, расположенные в таком порядке, как если бы эти элементы были отсортированы. В этом случае сам исходный массив не подвергается никаким изменениям, а возвращаемый массив лишь отражает результат возможной сортировки. Поясним это на примере:

```
var letters = ["в", "а", "б", "е"];
var indexedArray = letters.sort(Array.RETURNINDEXEDARRAY)
```

Массив indexedArray представляет собой следующее:

```
[1,2,0,3]
```

Здесь первый элемент (1) указывает, что первым в отсортированном массиве будет расположен элемент с индексом 1 ("а"), второй элемент (2) указывает на местоположение в отсортированном массиве элемента с индексом 2 ("б") и т. д.

- Опция Array.NUMERIC — применяется для сортировки массивов, состоящих из числовых элементов. Применение этой опции приводит к изменению исходного

массива. Использование опции Array.NUMERIC позволяет избежать проблем, возникающих при сравнении строковых представлений чисел. Проиллюстрируем это на примере:

```
var numbers = [2, 10, 4, 3, 6, 5, 1];
numbers.sort(Array.NUMERIC);
```

Массив теперь имеет следующий вид:

```
[1, 2, 3, 4, 5, 6, 10]
```

Если отсортировать данный массив без применения параметра Array.NUMERIC результирующий массив будет иметь иной вид:

```
[1, 10, 2, 3, 4, 5, 6, 10]
```

Примечание

Следует помнить, что, объединяя несколько опций при вызове метода sort(), можно выполнять сортировку с учетом каждой из них.

Применение пользовательской функции сравнения

Опции метода sort() позволяют успешно реализовать стандартные виды сортировки. Тем не менее иногда этих возможностей оказывается недостаточно для решения конкретных специфических задач. В этом случае можно прибегнуть к еще одной возможности, предоставляемой методом sort(). В качестве первого параметра методу sort() может быть передана ссылка на пользовательскую функцию, которая будет использована для определения порядка сортировки при сравнении элементов массива. Такая функция называется *функцией сравнения* (compare function). Во время выполнения метода sort() функция сравнения вызывается для каждой пары элементов массива, в котором выполняется сортировка для определения их взаимного размещения. В результате выполнения метода sort() элементы массива упорядочиваются в соответствии с правилами, заданными в функции сравнения.

При создании функции сравнения необходимо руководствоваться следующими правилами.

- Функция сравнения принимает два параметра, представляющие два сравниваемых элемента массива. Во время выполнения метода sort() функция сравнения вызывается для каждой пары элементов массива, в котором выполняется сортировка, для определения их взаимного размещения.
- Функция сравнения должна возвращать отрицательное, положительное или нулевое значение.
  - Если возвращаемое функцией сравнения значение оказывается меньше нуля, то элемент, переданный этой функции в качестве первого параметра, в отсортированном массиве будет расположен перед элементом, представленным вторым параметром.
  - Если возвращаемое значение оказывается больше нуля, то элемент, представленный первым параметром, в отсортированном массиве будет расположен следом за элементом, представленным вторым параметром.

- Если возвращаемое значение оказывается равно нулю, то элементы считаются равными.

Вызов метода `sort()` с применением пользовательской функции сравнения имеет следующий синтаксис:

массив`.sort(`*функция сравнения*`,` *опция1* `|` *опция2* `|,.....,|` *опция5*`)`

Обратите внимание, что при необходимости можно использовать и функцию сравнения, и встроенные опции. Также следует отметить, что методу `sort()` передается только ссылка на функцию сравнения, поэтому оператор вызова функции `()` не используется.

Для начала создадим простую функцию сравнения `descend`, позволяющую задать порядок сортировки по убыванию:

```
function descend (a, b){
return (b - a);
}
```

Рассмотрим принцип действия этой функции.

Параметры `a` и `b` — это пара элементов массива. Поскольку перед нами стоит задача выполнения сортировки по убыванию, необходимо проверить, значений какого параметра меньше, и в зависимости от этого вернуть отрицательное, положительное число или число 0 в случае равенства значений параметров. Для этого достаточно вернуть результат вычисления выражения `b - a`. Убедимся, что это действительно так.

Если `a` оказывается больше `b`, данное выражение возвращает отрицательное значение, и значит, в отсортированном массиве элемент `a` будет расположен перед элементом `b`.

Если `a` оказывается меньше `b`, выражение `b - a` возвращает положительное значение, и значит, в отсортированном массиве элемент `a` будет следовать за элементом `b`.

Если же значения `a` и `b` оказываются равны, выражение `b - a` возвращает нулевое значение.

Проверим нашу функцию в действии:

```
var numbers = [2, 5, 12, 10, 4, 3, 6, 5, 1];
numbers.sort(descend);
```

Массив `numbers` после сортировки имеет следующий вид:

```
[12, 10, 6, 5, 5, 4, 3, 2, 1]
```

Рассмотрим еще один пример сортировки массива с применением функции сравнения. Допустим, у нас имеется массив, содержащий набор дат, заданных в качестве строк вида:

```
"чч.мм.гггг" //число.месяц.год
```

Создадим исходный массив:

```
var dates:Array = ["26.08.1979", "05.02.1992", "12.01.2004",↳
"22.07.1998","10.10.1999","12.02.2004", "05.02.1992", "06.02.1992",↳
"27.07.2004"]
```

Необходимо отсортировать даты, хранящиеся в этом массиве, по возрастанию. В листинге 9.1 представлена функция сравнения `compareDates()`, выполняющая эту задачу. Поскольку каждый элемент исходного массива хранит дату в виде единой строки, при сравнении элементов эти строки разбиваются на составляющие (число, месяц, год) с помощью метода `String.split()` и затем преобразуются в числа с помощью функции `parseInt()`. Выделенные таким образом составляющие дат сравниваются, начиная с года. При выполнении сравнения элементов дат и возвращении результата используется подход, рассмотренный ранее.

Листинг 9.1. Функция сравнения для сортировки дат по возрастанию

```
function compareDates(date1, date2) {
/*Помещаем составляющие элементы дат в массивы. Первый элемент каждого
массива хранит число, второй - месяц, третий - год*/
       var a:Array = date1.split(".");
       var b:Array = date2.split(".");
//Преобразуем элементы дат в числа
       for (i=0; i<3;i++){
       a[i]=parseInt(a[i]);
       b[i]=parseInt(b[i]);
       }
//Проверка года (элемент с индексом 2)
       if (a[2] != b[2]) {
              return (a[2] - b[2]);
       }
//Проверка месяца (элемент с индексом 1)
       else if (a[1] != b[1]) {
              return (a[1] - b[1]);
       }
//Проверка числа, а также обработка ситуации равенства двух дат
       else {
              return (a[0] - b[0]);
       }
}
```

Выполним сортировку:

```
dates.sort(compareDates);
```

Теперь выведем в панель **Output** результат. Для вывода каждой даты с новой строки применим метод `Array.join()`, передав ему в качестве разделителя символ перевода строки.

```
trace(dates.join("\n"));
```

В панели **Output** выводится:

```
26.08.1979
05.02.1992
05.02.1992
06.02.1992
22.07.1998
10.10.1999
12.01.2004
12.02.2004
27.07.2004
```

Метод *Array.sortOn()*

Метод `sortOn()` применяется для сортировки массивов, элементы которых являются объектами. Данный метод позволяет выполнять сортировку массива, состоящего из объектов, по определенному свойству объекта или *полю* (field). Если все объекты, хранящиеся в массиве, имеют одинаковый набор свойств, то метод `sortOn()` может легко упорядочить их по значению любого из свойств или, другими словами, по любому полю. Так, если в каждом из объектов массива имеются свойства `имя`, `возраст`, `пол`, то элементы данного массива могут быть отсортированы в соответствии с любым из этих полей, то есть по имени, по возрасту или по принадлежности к полу. Метод `sortOn()` также позволяет выполнять сортировку сразу по нескольким полям. В этом случае поле, указанное первым, имеет наивысший приоритет, следующее меньший и т. д. Кроме того, при сортировке элементов может быть задана одна или несколько стандартных опций сортировки, используемых методом `sort()`, описание которых приводилось в предыдущем разделе. Следует отметить, что в отличие от метода `sort()`, метод `sortOn()` не поддерживает возможность использования пользовательской функции сравнения.

При сортировке по одному или нескольким полям без указания специальных опций сортировки действуют те же правила сортировки, что и при использовании метода `sort()` без параметров.

Сортировка по одному полю

Для того чтобы отсортировать элементы массива по значению одного поля, используется следующий синтаксис:

`массив``.sortOn(`*`свойство`*`)`

Здесь *`массив`* — это массив, элементы которого требуется отсортировать.

Параметр *свойство* — это имя свойства объектов, на основании значений которого будет выполнена сортировка. Данный параметр должен быть задан строковым литералом (то есть заключен в кавычки) либо выражением, возвращающим строку.

В случае если один или несколько объектов в сортируемом массиве не содержат свойства, заданного параметром *свойство*, то данное свойство считается равным `udefined`, а сами объекты последовательно размещаются в отсортированном массиве. Рассмотрим следующий пример.

Имеется массив с названием `clients`, хранящий данные об условных клиентах фирмы. Каждый клиент — это элемент массива, представляющий собой объект со свойствами `sName`, `nAge`, `sGender`, предназначенными для хранения данных об имени, возрасте и поле данного клиента. В данном случае используем литерал объекта, имеющий следующий синтаксис:

```
{свойство1:значение, свойство2:значение, свойствоN:значение}
```

Итак, создадим исходный массив:

```
var clients:Array = new Array
(
{sName:"Олег", nAge:32, sGender:"муж"},
{sName:"Эйприл", nAge:24, sGender:"жен"},
{sName:"Алла", nAge:30, sGender:"жен"},
{sName:"Егор", nAge:25, sGender:"муж"},
{sName:"Марк", nAge:2, sGender:"муж"},
{sName:"Ольга", nAge:55, sGender:"жен"},
{sName:"Егор", nAge:15, sGender:"муж"},
{sName:"Стив", nAge:57, sGender:"муж"},
{sName:"Егор", nAge:9, sGender:"муж"}
)
```

Для наглядности при создании массива мы поместили каждый элемент на отдельной строке, что синтаксически является вполне корректным.

Отсортируем элементы этого массива по именам, то есть по полю `sName`.

```
clients.sortOn(["sName"]); // Сортировка по имени
```

Теперь выведем значения элементов отсортированного массива:

```
for(i=0;i<clients.length;i++){
trace(clients[i].sName+ " " +clients[i].nAge+" "+clients[i].sGender);
}
```

В панели **Output** появится следующее:

```
Алла 30 жен
Егор 9 муж
Егор 25 муж
```

```
Егор 15 муж
Марк 2 муж
Олег 32 муж
Ольга 55 жен
Стив 57 муж
Эйприл 24 жен
```

Обратите внимание, что три объекта с одинаковым значением свойства `sName` "Егор" в отсортированном массиве располагаются без учета значения свойства `nAge`. Для того чтобы при сортировке анализировались значения нескольких полей, необходимо передать их в качестве параметров методу `sortOn()`.

Сортировка по нескольким полям

При выполнении сортировки по нескольким полям элементы сортируются по полю, имеющему наибольший приоритет, затем отсортированные таким образом элементы сортируются по следующему полю, переданному методу `sortOn()` в качестве параметра, не нарушая при этом порядка сортировки по полю с большим приоритетом. При необходимости выполнения сортировки по нескольким полям эти поля передаются методу `sortOn()` в виде массива, первый элемент которого задает поле сортировки с наивысшим приоритетом и каждый последующий задает поле сортировки с меньшим приоритетом. В этом случае вызов метода `sortOn()` имеет следующий синтаксис:

массив.sortOn([*свойство1*, *свойство2*, ..., *свойствоN*])

Обратите внимания на то, что поля для сортировки передаются методу `sortOn()` в качестве единственного параметра, представляющего собой массив. Массив полей сортировки может быть передан методу как литерал либо с помощью идентификатора переменной, содержащей ссылку на этот массив.

Рассмотрим пример сортировки по нескольким полям. Для этого вернемся к рассмотренному выше массиву `clients`. Отсортируем расположенные в нем элементы по полу и по имени клиентов. Полями сортировки в этом случае будут выступать свойства `sGender` и `sName`.

```
clients.sortOn(["sGender", "sName"]);// Сортировка по полу и по имени
for(i=0;i<clients.length;i++){
trace(clients[i].sName+ " " +clients[i].nAge+" "+clients[i].sGender);
}
```

В панели **Output** будет выведен следующий результат:

```
Алла 30 жен
Ольга 55 жен
Эйприл 24 жен
Егор 15 муж
Егор 25 муж
Егор 9 муж
```

```
Марк 2 муж
Олег 32 муж
Стив 57 муж
```

Обратите внимание, что первым элементом массива, переданного методу `sortOn()` в качестве параметра, выступает свойство `sGender`, поэтому при сортировке приоритет отдается полу и только потом среди отсортированных по полу элементов выполняется сортировка по имени.

Теперь попробуем отсортировать элементы по полу и затем по возрасту:

```
clients.sortOn(["sGender", "nAge"]);// Сортировка по полу и по возрасту
```

При выводе результата в панели **Output** отобразится следующее:

```
Эйприл 24 жен
Алла 30 жен
Ольга 55 жен
Егор 15 муж
Марк 2 муж
Егор 25 муж
Олег 32 муж
Стив 57 муж
Егор 9 муж
```

Как видно, сортировка по возрасту среди отсортированных по полу клиентов выполняется не совсем корректно. Так, возраст 9 оказывается больше, чем возраст 57. Этому факту можно найти очень простое объяснение, вспомнив, что по умолчанию (то есть без использования встроенных опций сортировки) метод `sortOn()`, как и метод `sort()`, сортирует строковые представления элементов в соответствии с кодовыми значениями их символов. Поскольку символ "9" имеет кодовое значение, большее, чем символ "5", мы получаем такой, на первый взгляд, неожиданный результат.

Для указания правила сортировки в подобных ситуациях можно прибегнуть к использованию встроенных опций метода `sortOn()`.

Сортировка с применением встроенных опций

Метод `sortOn()`, как и метод `sort()`, позволяет указать одну или несколько опций, определяющих порядок сортировки элементов. Перечень и подробное описание назначения этих опций приводились в *разд. "Метод Array.sort()" данной главы*. При вызове метода `sortOn()` с указанием опций сортировки используется следующий синтаксис:

```
массив.sortOn([свойство1, свойство2, ......, свойствоN], опция1 | ..... | опцияN)
```

Попробуем отсортировать массив `clients` по убыванию значений по полу и по возрасту с применением опций `Array.DESCENDING` и `Array.NUMERIC`:

```
clients.sortOn(["sGender", "nAge"], Array.DESCENDING | Array.NUMERIC);
```

При выводе результата в панели **Output** отобразится следующее:

```
Стив 57 муж
Олег 32 муж
Егор 25 муж
Егор 15 муж
Егор 9 муж
Марк 2 муж
Ольга 55 жен
Алла 30 жен
Эйприл 24 жен
```

Извлечение части массива

Метод *Array.slice()*

Этот метод класса `Array` по своему назначению очень похож на одноименный метод класса `String`, описанный в *главе 5*. Этот метод позволяет извлечь набор последовательно расположенных элементов массива и возвращает их, как новый массив. При этом исходный массив остается без изменений.

Метод `slice()` имеет следующий синтаксис:

массив`.slice(`*начальный индекс*`, `*конечный индекс*`)`

Здесь *массив* — это исходный массив, из которого извлекаются элементы.

- Параметр *начальный индекс* задает индекс первого элемента извлекаемого набора. Этот параметр может принимать любое положительное целочисленное значение и задается в виде литерала или выражения, возвращающего строку. Если значение параметра *начальный индекс* положительно, отсчет индексов элементов начинается с начала массива. Если данный параметр отрицателен, отсчет индексов начинается с конца массива, причем последний (слева направо) элемент массива имеет индекс −1.
- Параметр *конечный индекс* задает индекс элемента, следующего за последним элементом извлекаемого набора. Тип и диапазон принимаемых значений параметра *конечный индекс* аналогичны приведенным характеристикам параметра *начальный индекс*. Если при вызове метода `slice()` данный параметр опускается, то метод возвращает массив, состоящий из набора элементов, начиная с элемента, заданного параметром *начальный индекс*, и заканчивая последним элементом исходного массива (то есть элементом с индексом `length-1`).

Примечание

Следует иметь в виду, что при задании значений параметров метода `splice()` элемент с индексом, заданным первым параметром, должен находиться ближе к началу массива, чем элемент с индексом, заданным вторым параметром. Отсчет элементов в массиве выполняется слева направо. Так, например, можно задать параметры –10 и –1,

но нельзя задать параметры –1 и –10, так как элемент с индексом –10 находится ближе к началу массива, чем элемент с индексом –1. В противном случае метод возвращает пустой массив.

Если оба параметра метода `slice()` опущены, метод возвращает копию исходного массива.

Следует помнить, что метод `slice()` не изменяет исходный массив (как это делает метод `splice()`), а возвращает новый массив, состоящий из элементов исходного массива.

Рассмотрим несколько примеров:

```
var myArray:Array = ["A", "B", "C", "D", "E", "F"];
trace(myArray.slice(2,5)); // Выводит C,D,E
trace(myArray.slice(2)); // Выводит C,D,E,F
trace(myArray.slice(2,-1)); // Выводит C,D,E
trace(myArray.slice(-4,-1)); // Выводит C,D,E
trace(myArray.slice(-5)); // Выводит B,C,D,E,F
myArray.slice(-2,1); /* Возвращает пустой массив, так как символ
с индексом -2 следует за символом с индексом 1 */
```

Удаление элементов массива

Для того чтобы удалить элемент или группу элементов из массива, можно воспользоваться несколькими способами. Часть из них уже была рассмотрена в предыдущих разделах данной главы. К числу рассмотренных возможностей относятся:

- свойство `length`, позволяющее удалить произвольное количество элементов, расположенных в конце массива;
- метод `splice()`, позволяющий удалить произвольное количество последовательно расположенных элементов из любого участка массива.

Рассмотрим другие возможности, предоставляемые методами класса `Array`.

Метод *Array.pop()*

Метод `pop()` является антагонистом метода `push()`. Он удаляет последний элемент массива, возвращая значение этого элемента.

Метод `pop()` не принимает параметров и имеет следующий синтаксис:

массив`.pop()`

Здесь *массив* — это идентификатор массива, из которого удаляется последний элемент.

Рассмотрим пример применения метода `pop()`.

```
var myArray:Array = ["a", "b", "c", "d", "e", "f", "g", "h"];
var lastElement = myArray.pop();
trace(lastElement); // Выводит h
trace(myArray); // Выводит a,b,c,d,e,f,g
```

Метод *Array.shift()*

Метод `shift()`, как и следует из его названия, является антагонистом метода `unshift()`. Данный метод удаляет первый элемент массива и возвращает значение этого элемента. В результате удаления первого элемента все остальные элементы массива "смещаются" к его началу.

Метод `shift()` не принимает параметров и имеет следующий синтаксис:

```
массив.shift()
```

Здесь *массив* — это идентификатор массива, из которого удаляется первый элемент.

Рассмотрим пример:

```
var myArray:Array = ["a", "b", "c", "d", "e", "f", "g", "h"];
firstElement = myArray.shift(2);
trace(firstElement); // Выводит a
trace(myArray); // Выводит a,b,c,d,e,f,g
```

Ассоциативные массивы

Как уже было отмечено, любой массив является экземпляром класса `Array` и представляет собой объект. Объекты могут иметь свойства, а значит, и массив как объект может обладать набором свойств, в которых можно хранить значения любых типов. Такая возможность фактически позволяет добавлять в массив элементы, которые идентифицируются не с помощью числового индекса, а по имени или *ключу*. Массивы, состоящие из элементов, индексированных ключами, называются *ассоциативными*. Каждый элемент ассоциативного массива представляет собой пару: имя (ключ) и значение.

Для добавления элемента в ассоциативный массив используется один из двух стандартных способов добавления свойства объекта:

```
var animalHash = new Array();
//Использование оператора точки
animalHash.cat = "Пушистый"; // Добавляем элемент cat
//Использование оператора квадратных скобок
animalHash["dog"] = "Верный"; // Добавляем элемент dog
```

В приведенном фрагменте кода создается массив `animalHash`, в который затем добавляются два элемента, индексированных ключами `cat` и `dog`. Фактически мы просто добавили в объект `animalHash` два новых свойства: свойство `cat` со строковым значением `"Пушистый"` и свойство `dog` со строковым значением `"Верный"`. Имя свойства выступает в качестве ключа элемента, с помощью которого к данному элементу можно будет обращаться для извлечения или модификации его значения. Обратите внимание, что при использовании оператора `[]` имя свойства задается, как строка, то есть заключается в кавычки.

Теперь мы можем обратиться к элементу ассоциативного массива с помощью оператора точки или квадратных скобок.

```
trace(animalHash["cat"]); // Выводит Пушистый
trace(animalHash.dog); // Выводит Верный
```

Следует отметить, что свойство `length` и все встроенные методы класса `Array` не работают с ассоциативными массивами.

```
trace(animalHash.length); /* Выводит 0, так как в массиве нет элементов с
числовыми индексами */
trace(animalHash.shift()); /* Выводит undefined, так как в массиве нет
элементов с числовыми индексами */
```

При попытке вывести ассоциативный массив в панель **Output** там будет выведена пустая строка. Два следующих предложения дадут один и тот же результат:

```
trace(animalHash.sortOn()); // Выводит пустую строку
trace([]); // Выводит пустую строку
```

Для обработки элементов ассоциативного массива или другими словами свойств объекта типа `Array` может быть использован цикл `for...in`:

```
for (i in animalHash){
trace(animalHash[i]);
}
```

В панель **Output** выводится следующее:

```
Пушистый
Верный
```

Следует отметить, что массив одновременно может содержать и обычные элементы, и элементы, индексированные ключами, то есть свойства. В этом случае применение свойства `length` и методов класса `Array` будет оказывать влияние только на элементы, индексированные числами.

```
var animalHash = new Array();
animalHash["dog"] = "Верный"; // Именованный элемент
animalHash["cat"] = "Пушистый"; // Именованный элемент
animalHash[0] = "Грозный" // Нумерованный элемент
trace(animalHash.length); /* Выводит 1, так как в массиве есть только
один элемент с числовым индексом */
```

Правда, цикл `for...in` может обрабатывать и именованные, и нумерованные элементы массива:

```
for (i in animalHash){
trace(animalHash[i]);
}
```

В панель **Output** выводится следующее:

```
Грозный
Пушистый
Верный
```

Многомерные массивы

До настоящего момента мы рассматривали только одномерные массивы, то есть массивы, элементы которых можно расположить в одну строку или в один столбец. Элементы любого одномерного массива можно отобразить следующим образом:

```
элемент1 элемент2......элементN
```

или

```
элемент1
элемент2
......
элементN
```

В действительности многие практические задачи требуют более сложной организации данных. В этом случае используются многомерные массивы, позволяющие размещать элементы в нескольких измерениях. В большинстве случаев использование многомерных массивов ограничивается двумерным массивом, представляющим собой матрицу, в которой элементы расположены по строкам и столбцам, и каждый элемент может быть однозначно идентифицирован по номеру своей строки и столбца. Элементы двумерного массива могут быть представлены следующим образом:

```
элемент11 элемент12 ...... элемент1N
элемент21 элемент22 ...... элемент2N
......        ......        ......
элементN1 элементN2 ...... элементNN
```

Первое число в индексе элемента указывает его строку, второе число — его столбец.

В ActionScript нет возможности непосредственно создать многомерный массив, указав его размерность. В связи с этим многомерные массивы реализуются как массивы, элементы которых, в свою очередь, также являются массивами. Такие массивы называются *вложенными* (nested).

Рассмотрим следующую ситуацию. Имеется набор трехмерных точек, положение каждой из которых в пространстве задается тройкой координат (абсцисса — *X*, ордината — *Y* и аппликата — *Z*). Для хранения этих точек удобно использовать двумерный массив, каждый элемент которого представляет одну точку и, в свою очередь также являясь массивом, состоящим из трех элементов, хранит координаты этой точки. Перед созданием такого массива вначале необходимо создать массивы координат для каждой точки, которые затем будут вложены в новый массив:

```
var point1:Array = [1, 2, 3]; // 1-я точка
var point2:Array = [4, 5, 6]; // 2-я точка
```

```
var point3:Array = [7, 8, 9]; // 3-я точка
var point4:Array = [10, 11, 12]; // 4-я точка
var point5:Array = [13, 14, 15]; // 5-я точка
```

Теперь создадим двумерный массив для хранения точек:

```
var coordinates:Array = new Array (point1,point2,point3,point4,point5);
```

Массив `coordinates` представляет собой матрицу. Каждая строка этой матрицы представляет одну точку, а в столбцах матрицы размещаются координаты этой точки. Для иллюстрации этого рассмотрим более наглядную форму создания массива `coordinates`.

```
var coordinates:Array = new Array (
[1, 2, 3]; // 1-я точка point1
[4, 5, 6]; // 2-я точка point2
[7, 8, 9]; // 3-я точка point3
[10, 11, 12]; // 4-я точка point4
[13, 14, 15]; // 5-я точка point5
)
```

Для того чтобы обратиться к элементу многомерного массива, используется несколько индексов. Первый (левый) индекс указывает на элемент наружного массива, второй индекс указывает на элемент вложенного массива, заданного первым индексом, и т. д. Рассмотрим несколько примеров обращения к координатам точек, хранящихся в массиве `coordinates`.

```
trace(coordinates[0][0]); //Выводит 1, X-координата точки 1
trace(coordinates[3][1]); //Выводит 11, Y-координата точки 4
trace(coordinates[4][2]); //Выводит 15, -координата точки 5
```

Как мы видим, первый индекс задает требуемую точку, а второй — координату этой точки. Напомним, что индексирование элементов в массиве начинается с нуля. Массив `coordinates` позволяет компактно хранить информацию о координатах и предоставляет удобные возможности доступа к этой информации. Так, поскольку каждая точка имеет три координаты, все вложенные массивы состоят из трех элементов. При этом индекс 0 соответствует координате *X*, индекс 1 соответствует координате *Y* и индекс 2 соответствует координате *Z*. Такая унификация дает широкие возможности автоматической обработки координат.

Модификация значения элемента многомерного массива выполняется так же, как и в одномерном, за исключением того, что для обращения к элементу используется несколько индексов. Для того чтобы модифицировать значение координаты какой-либо точки, можно просто присвоить ей новое значение:

```
coordinates[2][0] = 100; //X-координата точки 3 теперь имеет значение 100
coordinates[2][1] = 200; //Y-координата точки 3 теперь имеет значение 200
```

Для обработки координат многомерного массива удобно использовать вложенные циклы. Рассмотрим это на примере двумерного массива `coordinates`. Внешний цикл проходит по элементам массива `coordinates` (по строкам), то есть по точкам,

а вложенный цикл проходит по элементам вложенного массива (по столбцам), то есть по координатам этих точек. Обратите внимание на составление проверочных выражений в заголовках циклов. Поскольку внешний цикл проходит по строкам, мы ограничиваем максимальное значение его счетчика количеством элементов двумерного массива (то есть `coordinates.length`). Вложенный цикл проходит по столбцам, поэтому максимальное значение его счетчика ограничивается количеством элементов в каждом вложенном массиве или строке (то есть `coordinates[i].length`, где `I` — это текущая строка). Приведенный код выводит в панель **Output** значения всех элементов массива `coordinates` в виде сплошного списка.

```
for (i = 0; i < coordinates.length; i++) {
       for (j = 0; j < coordinates[i].length; j++) {
              trace(coordinates[i][j]);
       }
}
```

Использование матриц позволяет просто и эффективно выполнять множество различных операций, таких как, например, трехмерные преобразования точек. Эта тема будет подробно рассмотрена в *главе 26*. Поскольку класс `Array` не имеет встроенных возможностей, позволяющих автоматически сформировать матрицу из набора заданных элементов, это необходимо делать самому разработчику. Рассмотренный на примере массива `coordinates` способ формирования матрицы (двумерного массива) является надежным и простым, однако он подходит только для тех ситуаций, когда число элементов в матрице невелико. Если элементов много, лучше прибегнуть к автоматическому формированию матрицы с помощью пользовательской функции.

В листинге 9.2 представлена функция `matrix()`, которая в качестве параметров принимает требуемые размеры матрицы и перечень ее элементов и возвращает полученную матрицу в виде двумерного массива. Функция `matrix()` имеет только два формальных параметра: `r` — число строк (row) матрицы и `c` — число столбцов (column) матрицы. Однако при вызове функции ей может быть передано произвольное число фактических параметров. Все параметры, начиная с третьего, интерпретируются как элементы, которые должны быть помещены в будущую матрицу. Причем порядок их передачи влияет на то, какое место в матрице будет занимать элемент. Формирование матрицы выполняется по строкам сверху вниз и слева направо. Если размер матрицы, определяемый первыми двумя параметрами функции `matrix()`, оказывается больше, чем количество переданных функции элементов, то все недостающие элементы будут иметь значение `undefined`. Если же количество элементов превышает размер матрицы, выдается соответствующее сообщение, и функция `matrix()` возвращает пустой массив.

Листинг 9.2. Формирование матрицы с заданным числом строк и столбцов

```
function matrix(r:Number, c:Number):Array {
/* Проверяем, если количество переданных элементов превышает размер
матрицы, выводим сообщение об ошибке и возвращаем пустой массив.
Количество элементов матрицы вычисляется как произведение числа строк
на число столбцов */
```

```
        if (r * c < arguments.length - 2) {
                trace("**ОШИБКА** Количество переданных элементов
                превышает указанный размер матрицы");
                return [];
        }
/* Создаем новый пустой массив, который будет использован для хранения
элементов сформированной матрицы, и сохраняем ссылку на него в локальной
переменной array */
        var array:Array = new Array();
// По строкам...
        for (var i = 0; i < r; i++) {
/* Для каждой строки создаем новый пустой массив с именем rowN, где N -
номер текущей строки */
                this["row" + i] = new Array();
// По столбцам...
                for (var j = 0; j < c; j++) {
// Заполняем массив rowN элементами строки с номером N
                        this["row" + i].push(arguments[j + 2 + i * c]);
                }
/* Добавляем массив rowN с элементами текущей строки в конец массива
array. Таким образом, после выхода из цикла массив array оказывается
заполненным массивами rowN, каждый из которых содержит элементы N-ой
строки матрицы, то есть массив array теперь представляет собой матрицу*/
                array.push(this["row" + i]);
        }
// Возвращаем полученную матрицу
        return array;
}
```

Проверим действие нашей функции. Создадим матрицу из пятнадцати элементов размером 5×3 и сохраним ссылку на нее в переменной `myMatrix`:

```
myMatrix=matrix(5, 3, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
```

Выведем элементы получившейся матрицы в панель **Output**:

```
trace(myMatrix.join("\n"));
```

В панели **Output** видим следующий результат:

```
1,2,3
4,5,6
7,8,9
```

```
10,11,12
13,14,15
```

Рассмотрим еще одну важную процедуру, выполняемую с матрицами, — транспонирование. Транспонирование матрицы заключается в том, что ее строки и столбцы меняются местами. Между элементами исходной и транспонированной матрицы имеется следующая зависимость:

$el_{ij} = el_{ji}^{T}$,

здесь индекс `i` указывает номер строки элемента, а индекс `j` — номер его столбца.

Таким образом, при транспонировании матрицы элемент ее `i`-ой строки и `j`-го столбца становится элементом `j`-ой строки и `i`-го столбца.

В листинге 9.3 представлена функция `transpose()`, выполняющая транспонирование матрицы, переданной данной функции в качестве параметра. Данная функция не проверяет свой параметр `matrix` на корректность (то есть не проводится оценка того, является ли `matrix` матрицей). При желании читатель сам может попытаться организовать такую проверку в качестве упражнения.

Листинг 9.3. Транспонирование матрицы

```
function transpose(matrix:Array):Array {
/* Создаем новый пустой массив array, который будет использован для
хранения элементов транспонированной матрицы */
       var array:Array = new Array();
//Транспонируем матрицу, переданную в качестве параметра
//По столбцам...
       for (var j = 0; j < matrix[0].length; j++) {
//Массив для хранения j-ой строки транспонированной матрицы
              this["row" + j] = new Array();
//По строкам
              for (var i = 0; i < matrix.length; i++) {
//Заполняем массивы rowj для j-ой строки транспонированной матрицы
                     this["row" + j].push(matrix[i][j]);
              }
//Помещаем строки транспонированной матрицы в массив array
              array.push(this["row" + j]);
       }
//Возвращаем транспонированную матрицу
       return array;
}
```

Проверим действие данной функции. В качестве исходной матрицы возьмем матрицу, созданную функцией `matrix()` в предыдущем примере.

```
matrixT = transpose(myMatrix);
```

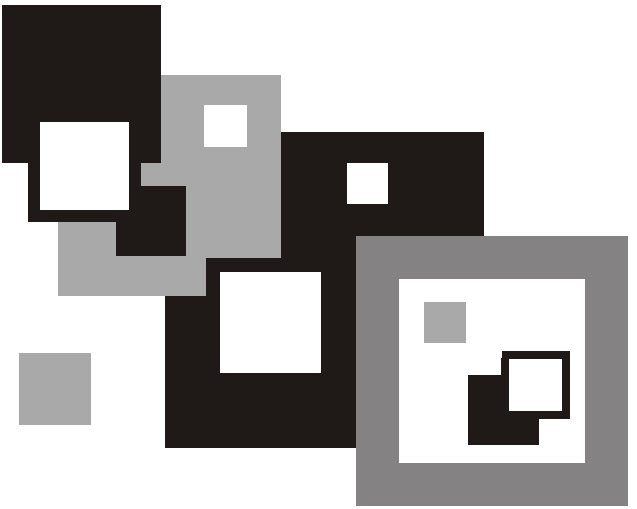
Выведем результат в панель **Output**:

```
trace(matrixT.join("\n"));
```

В панели **Output** выводится следующее:

```
1,4,7,10,13
2,5,8,11,14
3,6,9,12,15
```

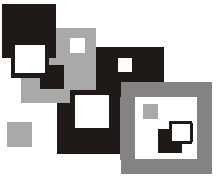
В *главе 26* также будет описан метод, реализующий умножение матриц.



ЧАСТЬ II

Базовые встроенные классы ActionScript

Глава 10



Класс *Button*. Работа с кнопками

Чем проще инструкция (например, "нажмите здесь"),
тем труднее найти, где же все-таки надо нажимать.

Второй закон кухонного беспорядка

Кнопка (Button) — это интерактивный элемент, служащий для инициирования определенных действий и обеспечивающий интерактивное взаимодействие с пользователем. Характерной особенностью кнопки является возможность изменения ее графического представления при взаимодействии с курсором мыши. В рабочей среде кнопки реализуются как экземпляры символа типа Button. Функциональность кнопок обеспечивается встроенным классом Button. Любая кнопка, созданная в рабочей среде, является объектом типа Button и обладает всеми возможностями, предоставляемыми классом `Button`. Несмотря на то, что кнопка является объектом типа Button, создание экземпляра (то есть конкретной кнопки) осуществляется не программным образом с помощью конструктора (как создается, скажем, массив), а исключительно в рабочей среде на базе символа типа Button. Таким образом, кнопка создается в рабочей среде, но ее контроль осуществляется программным образом.

Начиная с шестой версии Flash (Flash MX) муви-клипы (movie clips) получили в свое распоряжение все возможности, которые до этого были монополизированы кнопками. В связи с этим при необходимости использования кнопки почти всегда можно реализовать ее функциональность с помощью клипа, то есть не прибегая к применению символа типа Button. В настоящее время по своей функциональности кнопка фактически является подмножеством клипа, обладая лишь ограниченным набором его возможностей. Тем не менее в реализации некоторых возможностей кнопок имеется своя специфика, и в некоторых ситуациях применение кнопок является более простым и удобным, чем использование клипов, позволяющих добиться того же результата. В настоящей главе будут рассмотрены вопросы использования кнопок, специфичные именно для символов типа Button. Возможности, общие для кнопок и клипов, будут подробно описаны в *главах 11* и *12*.

При создании символа типа Button на его монтажной линейке задаются три состояния, определяющие внешний вид кнопки при отсутствии взаимодействия, при наведении на нее курсора мыши и в момент нажатия. Кроме того, в отдельном кадре задается активная область, в которой кнопка реагирует на данные действия. Кадры монтажной линейки кнопки не воспроизводятся последовательно, как это происходит с монтажными линейками других символов. Вместо этого при осуществлении

определенного воздействия на экземпляр кнопки (например, наведения курсора мыши или нажатия ее левой кнопки) отображается соответствующий кадр ее монтажной линейки. Символ типа Button может содержать экземпляры любых других символов, а также событийный звук. В кнопке нельзя использовать слой-маску. Так же, как и у муви-клипа, монтажная линейка кнопки не зависит от родительской монтажной линейки. Экземпляру символа типа Button может быть присвоен идентификатор (instance name) и назначен сценарий, выполняемый в результате наступления определенного события, связанного с кнопкой или клавиатурой. Применение кнопок всегда связано с созданием сценария, поскольку кнопка является интерактивным элементом. Воздействие на кнопку, кроме изменения ее внешнего вида, должно приводить к выполнению определенных действий, определяемых в сценарии кнопки.

Создание кнопки в рабочей среде

Монтажная линейка символа типа Button в общем случае содержит четыре кадра (рис. 10.1), первые три из которых задают внешний вид кнопки в соответствующем состоянии, а четвертый используется для задания области реагирования кнопки.

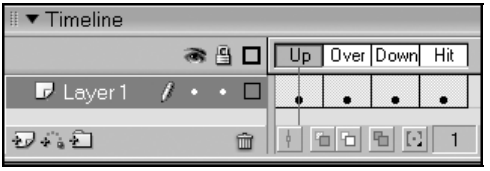


Рис. 10.1. Монтажная линейка символа типа Button

В отличие от кадров других монтажных линеек, кадры монтажной линейки кнопки идентифицируются не номерами, а названиями, определяющими их назначение.

- **Up** — кадр содержит изображение кнопки в неактивном состоянии, то есть когда курсор мыши находится вне области реагирования кнопки.
- **Over** — кадр содержит изображение кнопки при наведении на нее курсора мыши, то есть это активное состояние, при котором курсор находится в области реагирования, но кнопка мыши не нажата.
- **Down** — кадр содержит изображение кнопки в момент нажатия левой кнопки мыши.
- **Hit** — этот кадр является служебным и содержит область реагирования кнопки. В конечном фильме содержимое кадра **Hit** не отображается. В качестве области реагирования кнопки должна быть использована векторная область (объект рабочего или наложенного уровня, являющийся формой). Ее цвет не имеет значения, поэтому в целях минимизации конечного объема файла в качестве цветового заполнения нужно использовать сплошной (solid) цвет. По этой же причине желательно избегать использования в кадре **Hit** контуров (обводок). Форма области реагирования может быть совершенно произвольной и либо совпадать с формой

самой кнопки, либо выходить за ее пределы, или, наоборот, находиться внутри области, занимаемой кнопкой. Область реагирования также может быть размещена отдельно от кнопки или представлять собой несколько несвязанных областей. Следует, однако, помнить, что реакция кнопки осуществляется только при воздействии курсора мыши на ее активную область.

Примечание

Если кадр **Hit** не будет создан, его роль возьмет на себя ближайший к нему кадр, то есть расположенные в нем векторные формы будут использованы в качестве области реагирования кнопки. Однако если кадр **Hit** будет создан, но останется пустым, то кнопка работать не будет, поскольку она будет лишена области реагирования.

Необходимо иметь в виду, что если несколько кнопок расположены друг над другом, то реагировать на действия пользователя будет только самая верхняя из них. Это также относится к ситуации, когда одна кнопка находится внутри другой. Если их активные области пересекаются, то функционировать будет только кнопка, расположенная выше.

Кнопка может содержать неограниченное число слоев.

Для создания кнопки нужно выполнить следующие действия.

1. Создать новый символ типа Button (**Modify>Convert to Symbol...** или **Insert>New Symbol...**).
2. Войти в среду его редактирования.
3. Создать необходимое число слоев, присвоив им соответствующие названия (**Insert>Timeline>Layer**). Следует иметь в виду, что в кнопках нельзя использовать слой-маску, однако можно поместить в кнопку экземпляр муви-клипа, содержащего маску.
4. Создать изображение кнопки в неактивном состоянии, соответствующем положению **Up** (отжатая кнопка), размещая разные элементы изображения в предназначенных для них слоях.
5. Создать новый ключевой кадр в положении **Over** (кнопка при наведении) (команды контекстного меню кадра **Insert Keyframe** или **Insert Blank Keyframe**) и поместить в него изображение кнопки в соответствующем состоянии (или изменить скопированное из предыдущего кадра изображение).
6. Повторяя эти действия, создать два следующих кадра; в кадре **Down** задать вид нажатой кнопки, а в кадре **Hit** — область реагирования.
7. Выйти из среды редактирования и поместить на сцену экземпляр созданной кнопки, вытащив его из окна библиотеки документа.
8. Не все кадры монтажной линейки кнопки обязательно должны быть ключевыми. Если внешний вид кнопки или ее отдельного элемента, размещенного в одном из слоев, не изменяется в различных состояниях, достаточно использовать один-единственный ключевой кадр и поместить после него простые кадры, соответствующие тем состояниям кнопки, где не происходит никаких изменений.

Особый тип кнопки представляет собой *пустая кнопка*. Пустая кнопка не содержит графических объектов и, соответственно, не видна в конечном фильме. Такая кноп-

ка позволяет создать прозрачную область произвольной формы, выполняющую функции кнопки. Используя этот прием, можно заставить любой статический объект (например, растровое изображение) или анимированный ролик, например, баннер, вести себя как кнопка, "накрыв" его сверху пустой кнопкой.

Для того чтобы создать пустую кнопку, необходимо оставить первые три кадра ее монтажной линейки пустыми, а в четвертый кадр **Hit** поместить форму (объект рабочего или наложенного уровня), используемую в качестве области реагирования. В рабочей среде такая кнопка отображается прозрачным бирюзовым цветом, но в конечном фильме она будет не видна.

Кнопка как объект *AS*

Как уже было сказано, любая кнопка, то есть экземпляр символа Button, является экземпляром встроенного класса `Button`. В связи с этим каждая кнопка обладает свойствами и методами (точнее, единственным методом), определенными в классе `Button`. Это означает, что каждый экземпляр кнопки имеет одинаковый набор свойств, значения которых могут быть различны для разных экземпляров. Доступ к каждому из этих свойств может быть получен из любого экземпляра кнопки. В связи с тем, что свойства и метод класса `Button` представляют подмножество свойств и методов класса `MovieClip`, их подробное описание можно найти в *главе 11*. Здесь же ограничимся только кратким перечнем свойств и методов (в количестве одного), определенными в классе `Button` (табл. 10.1).

Таблица 10.1. *Свойства и методы класса* `Button`

Свойство	Описание
`Button._alpha`	Степень прозрачности экземпляра кнопки
`Button.enabled`	Определяет, является ли кнопка активной
`Button._focusrect`	Определяет, отображается ли желтая рамка у кнопки, находящейся в фокусе
`Button._height`	Высота кнопки в пикселах
`Button._highquality`	Определяет степень сглаживания графики в документе (глобальное свойство)
`Button.menu`	Связывает объект `ContextMenu` с экземпляром кнопки
`Button._name`	Содержит имя экземпляра кнопки
`Button._parent`	Ссылка на объект, содержащий экземпляр кнопки
`Button._quality`	Определяет качество отображения графики в документе (глобальное свойство)
`Button._rotation`	Угол поворота экземпляра кнопки в градусах
`Button._soundbuftime`	Количество секунд перед началом воспроизведения потокового звука (глобальное свойство)

Таблица 10.1 *(окончание)*

Свойство	Описание
`Button.tabEnabled`	Определяет, включен ли данный экземпляр кнопки в автоматический порядок переноса фокуса
`Button.tabIndex`	Указывает индекс экземпляра кнопки в используемом порядке переноса фокуса
`Button._target`	Содержит полный путь к экземпляру кнопки в нотации с косой чертой (устаревший способ адресации)
`Button.trackAsMenu`	Определяет, используется ли данным экземпляром кнопки режим меню (`trackAsMenu`)
`Button._url`	Содержит URL документа SWF, в котором создана кнопка
`Button.useHandCursor`	Определяет тип указателя курсора, используемого при взаимодействии с данным экземпляром кнопки
`Button._visible`	Определяет, является ли кнопка видимой
`Button._width`	Ширина кнопки в пикселах
`Button._x`	*X*-координата точки регистрации кнопки
`Button._xmouse`	*X*-координата курсора мыши относительно начала координат объекта, содержащего кнопку
`Button._xscale`	Масштаб кнопки по горизонтали
`Button._y`	*Y*-координата точки регистрации кнопки
`Button._ymouse`	*Y*-координата курсора мыши относительно начала координат объекта, содержащего кнопку
`Button._yscale`	Масштаб кнопки по вертикали
Метод	**Описание**
`Button.getDepth()`	Возвращает значение уровня (глубины) стека, на котором находится экземпляр кнопки

Имя экземпляра

Являясь объектом, экземпляр кнопки должен обладать уникальным идентификатором (именем), с помощью которого можно обращаться к данной кнопке и осуществлять контроль производимых ею действий. Такой идентификатор, называемый именем экземпляра (instance name), может быть задан явно или в противном случае создается автоматически. Для того чтобы указать имя экземпляра, необходимо выделить экземпляр кнопки на сцене, а в поле **Instance Name**, расположенном в левом верхнем углу панели **Properties** (рис. 10.2), ввести набор символов, отвечающий требованиям к идентификаторам, описанным в *разд. "Имена" главы 3*. Если имя экземпляра не будет указано в поле **Instance Name**, идентификатор будет создан автоматически в процессе публикации и будет состоять из слова "`instance`" и порядкового

номера объекта. Номер указывается в соответствии с количеством анонимных объектов, созданных ранее.

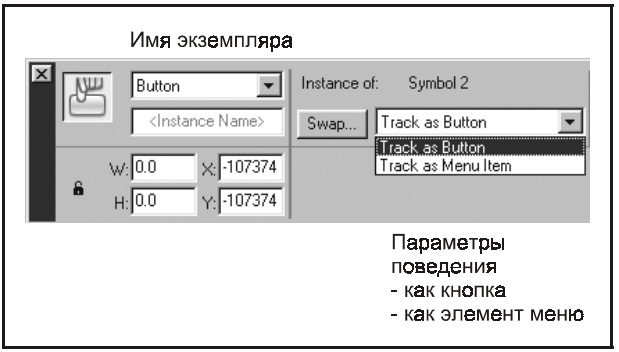


Рис. 10.2. Параметры экземпляра символа типа Button

Используя имя экземпляра можно обращаться к свойствам кнопки, вызывать ее метод, а также осуществлять обработку событий кнопки. При этом используется стандартный точечный синтаксис. Так, например, следующий фрагмент кода повернет кнопку с именем `but` на 30° по часовой стрелке и сделает ее прозрачной на 50%:

```
but._rotation = 30;
but._alpha = 50;
```

Имя экземпляра кнопки требуется задавать в случае, когда обращение к кнопке выполняется из внешнего сценария (не из сценария самой кнопки). Если сценарий кнопки прикреплен непосредственно к ее экземпляру и обращение к кнопке из внешних сценариев не выполняется, имя экземпляра можно не указывать (однако Flash все равно создаст его автоматически).

Режимы кнопки

Кнопка реагирует на курсор мыши, только когда он взаимодействует с областью реагирования кнопки. При этом у кнопок существует два режима, определяющих особенности их функционирования. Эти режимы называются **Track as Button** и **Track as Menu Item**. Их назначение состоит в следующем.

- **Track as Button** — если кнопка мыши нажата до вхождения в активную область кнопки, то кнопка не изменит состояние при наведении курсора, то есть не прореагирует.
- **Track as Menu Item** — состояние кнопки изменится при наведении курсора с нажатой кнопкой мыши, даже если нажатие кнопки мыши произошло до вхождения в активную область кнопки, то есть кнопка прореагирует.

Различия в использовании первого или второго режима особенно проявляются при обработке событий кнопки. Эти различия подробно описаны в табл. 10.2.

Для установки режима кнопки необходимо выделить требуемый экземпляр и в выпадающем списке **Options for Buttons** панели **Properties** выбрать нужное значение (см. рис. 10.2).

Обработка событий кнопки

В ActionScript определен ряд событий, которые автоматически генерируются при взаимодействии пользователя с кнопкой. Для того чтобы в ответ на воздействие курсора мыши кнопка могла выполнять определенные действия, необходимо определить, какое событие будет инициировать выполнение действий и описать сами действия, которые должны быть выполнены при наступлении данного события.

События кнопки

События кнопки связаны с различными вариантами воздействия на область реагирования кнопки курсором мыши в сочетании с использованием *левой* кнопки мыши. Нажатие других кнопок мыши не обрабатывается. Кроме того, для кнопок определено особое событие, позволяющее контролировать нажатие любой клавиатурной клавиши. В табл. 10.2 представлен перечень всех событий кнопки. Любой экземпляр кнопки может обнаружить наступление любого из этих событий и в качестве реакции на него выполнить указанный блок кода. Обратите внимание, что наступление некоторых событий происходит по-разному, в зависимости от используемого режима кнопки

Таблица 10.2. *События кнопки*

Событие	Описание	
	Режим Track as Button	**Режим Track as Menu Item**
`press`	Кнопка мыши нажата при нахождении курсора в области реагирования кнопки	
`release`	Кнопка мыши нажата и затем отпущена при нахождении курсора в области реагирования кнопки	Кнопка мыши отпущена в области реагирования кнопки вне зависимости от местонахождения курсора в момент ее нажатия
`rollOver`	Курсор входит в область реагирования кнопки. Если в момент вхождения кнопка мыши оказывается нажатой, данное событие генерируется в момент отпускания кнопки мыши при нахождении ее курсора в активной области кнопки	Курсор входит в область реагирования кнопки при ненажатой кнопке мыши
`rollOut`	Курсор выходит за пределы области реагирования кнопки при ненажатой кнопке мыши	

Таблица 10.2 *(окончание)*

Событие	Описание	
	Режим Track as Button	**Режим Track as Menu Item**
`dragOut`	Курсор выходит за пределы области реагирования кнопки при нажатой кнопке мыши	
`releaseOutside`	Курсор выходит за пределы области реагирования кнопки при нажатой кнопке мыши, после чего кнопка мыши отпускается вне области реагирования	Событие не генерируется
`dragOver`	Курсор выходит за пределы области реагирования кнопки при нажатой кнопке мыши, после чего вновь возвращается в область реагирования, кнопка мыши остается нажатой	Курсор входит в область реагирования кнопки при нажатой кнопке мыши
`keyPress`	Нажатие любой заданной клавиши клавиатуры	

Всякий раз, когда пользователь выполняет одно из описанных в данной таблице действий, происходит генерирование соответствующего события. Однако факт наступления события сам по себе не может инициировать выполнение определенных действий. Для этого требуется использование особой инструкции, называемой *обработчик событий* (event handler), определяющей блок кода, который выполняется при наступлении конкретного события. До тех пор, пока экземпляру кнопки не будет назначен обработчик событий, любые манипуляции с этой кнопкой не будут иметь никакого результата.

Обработчик событий кнопки

Для того чтобы заставить кнопку выполнять требуемые действия при наступлении заданного события, необходимо снабдить эту кнопку обработчиком событий и блоком кода, который должен выполняться в момент перехвата события. Создать обработчик событий кнопки можно двумя способами.

- Первый способ заключается в том, что сценарий, содержащий обработчик событий, прикрепляется непосредственно к экземпляру кнопки, которая должна реагировать на эти события. Применение данного подхода предполагает создание индивидуального сценария для каждой кнопки, используемой в документе, и распределение этих сценариев непосредственно по соответствующим кнопкам. Это приводит к рассредоточению программного кода по различным элементам документа и, как следствие, к усложнению и запутанности структуры программного кода. Плюсом такого подхода является более компактная запись и простота.
- Второй способ заключается в использовании методов обработчиков событий. Использование методов обработчиков событий не требует прикрепления сценария кнопки непосредственно к ее экземпляру. Обработка событий кнопки может

осуществляться из сценария, расположенного в любой части документа (чаще всего на основной монтажной линейке). Это позволяет добиться централизации программного кода, более простой и ясной структурной организации программы.

Поскольку использование методов обработчиков событий — это более общий подход, который широко применяется не только к кнопкам, но также и к другим объектам, он будет подробно описан в *главе 11*. Здесь же мы пока рассмотрим вопросы прикрепления обработчика кнопок непосредственно к их экземплярам. Информация о размещении сценариев в различных элементах структуры документа содержится в *разд. "Носители сценариев AS" главы 3*.

Обработчик событий кнопки называется `on()` и имеет следующий синтаксис:

```
on(событие){
блок кода;
......
}
```

Здесь *событие* — это название события, наступление которого повлечет за собой выполнение блока кода, ограниченного фигурными скобками. Рассмотрим простой пример. Следующий сценарий кнопки при нажатии выводит в панель **Output** соответствующее сообщение.

```
on(press){
trace("Нажатие кнопки");
}
```

Блок кода обработчика событий, как и блок кода функции, может состоять из неограниченного количества предложений, которые оформляются в соответствии со стандартными правилами синтаксиса предложений.

Примечание

Необходимо помнить, что любой сценарий, прикрепляемый к кнопке, всегда начинается с обработчика событий `on`. Все инструкции, выполняемые в результате воздействия на кнопку, должны быть помещены в блок кода обработчика событий кнопки, ограниченный фигурными скобками. Ни одно предложение (или даже символ) в сценарии кнопки за исключением заголовка обработчика событий и комментариев, не может находиться вне блока обработчика событий. Нарушение этого правила вызовет ошибку.

При необходимости один обработчик может перехватывать сразу несколько событий. В этом случае один и тот же блок кода обработчика событий будет выполнен при наступлении любого из указанных в его заголовке событий. События указываются через запятую. При этом используется следующий синтаксис:

```
on(событие1, событие2, ......, событие8){
блок кода;
......
}
```

В следующем примере сообщение в панель **Output** выводится при уходе курсора мыши за пределы области реагирования кнопки вне зависимости от того, была нажата ее левая кнопка или нет.

```
on(rollOut, dragOut){
trace("Уход за пределы кнопки");
}
```

Для того чтобы одна и та же кнопка выполняла различные действия в зависимости от типа перехваченного события, можно в одном сценарии кнопки использовать несколько обработчиков, каждый из которых содержит свой блок кода. В этом случае используется следующий синтаксис:

```
on(событие1, событие2, ......, событие8){
блок кода1;
......
}
on(событие1, событие2, ......, событие8){
блок кода2;
......
}
......
on(событие1, событие2, ......, событие8){
блок кода 8;
......
}
```

Листинг 10.1 содержит сценарий кнопки, в котором выполняется обработка всех событий кнопки, связанных с мышью. При воздействии на кнопку с данным сценарием в панели **Output** выводится сообщение с названием перехваченного события.

Листинг 10.1. Перехват событий кнопки

```
on (press) {
       trace("Событие press");
}
on (release) {
       trace("Событие release");
}
on (releaseOutside) {
       trace("Событие releaseOutside");
}
on (rollOver) {
```

```
        trace("Событие rollOver");
}
on (rollOut) {
        trace("Событие rollOut");
}
on (dragOver) {
        trace("Событие dragOver");
}
on (dragOut) {
        trace("Событие dragOut");
}
```

Если в одном и том же сценарии кнопки различные обработчики осуществляют перехват одинаковых событий, то при наступлении данных событий выполняются все блоки кода этих обработчиков.

Контроль нажатия клавиатурной клавиши

Обработчик `on()` позволяет перехватывать событие `keyPress`, которое в отличие от всех остальных событий кнопок не связано с мышью. Данное событие генерируется в момент нажатия указанной клавиатурной клавиши на протяжении всего времени, пока клавиша остается нажатой. При этом используется следующий синтаксис:

```
on(keyPress "клавиша"){
блок кода;
.......
}
```

Здесь *клавиша* — это символ на клавиатуре, соответствующий клавише (или сочетанию клавиш), или специальная клавиатурная константа. При указании клавиши с помощью символа следует помнить, что этот символ должен иметь код в пределах кодировки ASCII. Это означает, что можно использовать символы "z", "s", "g", но нельзя использовать символы "я", "ы", "п", даже несмотря на то что они находятся на одних и тех же клавишах. Событие `keyPress` также позволяет контролировать нажатие сочетаний клавиш с применением клавиши <Shift>, дающих символы из кодировки ASCII. К числу таких символов относятся, например, (*), ("), ({), а также все прописные буквы, поскольку они получаются при нажатии сочетания буквенной клавиши и клавиши <Shift>.

Рассмотрим несколько примеров:

```
on(keyPress "r"){
trace("Нажата клавиша <r>");
}
```

```
on(keyPress "R"){
trace("Нажаты клавиши <Shift> + <r>");
}
on(keyPress "*"){
trace("Нажаты клавиши <Shift>+<*> или <*> на дополнительной клавиатуре");
}
```

Клавиатурные константы позволяют контролировать нажатие клавиш, не имеющих символов, таких как, например, <Backspace>, <Escape>, <Left> и т. д. В табл. 10.3 представлен перечень существующих клавиатурных констант:

Таблица 10.3. *Клавиатурные константы события* `keyPress`

<BACKSPACE>	<CAPSLOCK>	<CONTROL>	<DELETEKEY>	<DOWN>	<END>
<ENTER>	<ESCAPE>	<HOME>	<INSERT>	<LEFT>	<PGDN>
<PGUP>	<RIGHT>	<SHIFT>	<SPACE>	<TAB>	<UP>

Следующие сценарии кнопки выводят в панель **Output** информацию о нажатой клавише, указанной с помощью константы.

```
on(keyPress "<Right>"){
trace("Нажата клавиша <Right> - стрелка вправо");
}
on(keyPress "<Enter>"){
trace("Нажата клавиша <Enter>");
}
```

Список клавиатурных констант выводится во всплывающей подсказке по коду, появляющейся после того, как будут набраны название обработчика и открывающая круглая скобка.

Примечание

В связи с тем, что в среде тестирования документа имеются свои собственные горячие клавиши, которые имеют приоритет над клавишами, используемыми в документе, это может явиться причиной некорректной работы. Так, например, клавиша <Enter> в среде тестирования является эквивалентом команд **Control>Play** или **Control>Stop**, в то время как в документе данная клавиша может использоваться для совершенно иных действий. Для того чтобы элиминировать воздействие горячих клавиш среды тестирования, их нужно отключить с помощью команды главного меню среды тестирования **Control> Disable Keyboard Shortcuts**.

Следует иметь в виду, что один обработчик может обрабатывать нажатие только одной клавиши. При необходимости контролировать нажатие нескольких клавиш нужно либо применять несколько обработчиков, либо использовать объект `Key`.

Область видимости обработчика событий кнопок

Под областью видимости обработчика событий кнопок понимается набор переменных и объектов, к которым можно обратиться из блока кода обработчика непосредственно, без применения адресации. В связи с тем, что кнопки могут быть размещены не только на основной монтажной линейке, но и на монтажных линейках клипов, при написании сценария кнопки необходимо четко представлять, какие объекты находятся в области непосредственного воздействия данного сценария.

Областью видимости обработчика событий кнопок `on()` является монтажная линейка, содержащая экземпляр кнопки, которой назначен сценарий. Это означает, что все команды, используемые в блоке кода обработчика, непосредственно распространяются только на монтажную линейку, содержащую кнопку. Так, если к кнопке прикреплен следующий сценарий:

```
on(press){
_x += 10;
_y += 10;
}
```

то при нажатии на эту кнопку объект, на монтажной линейке которого она расположена, сместится на 10 пикселов вправо и вниз. Если кнопка находится на монтажной линейке клипа, то сместится клип вместе со всем своим содержимым, включая и саму кнопку. Если кнопка находится на основной монтажной линейке, то сместится все ее содержимое (естественно, вместе с кнопкой). Поскольку в приведенном примере обращение к свойствам `_x` и `_y` осуществляется непосредственно (то есть без явного указания на целевой объект, содержащий данные свойства), то целевой объект определяется, исходя из области видимости обработчика событий кнопки. В связи с этим в приведенном выше сценарии выполняется обращение к свойствам `_x` и `_y` не самой кнопки, а родительского по отношению к кнопке объекта. Вместо такой неявной ссылки на целевой объект более правильно (хотя в данном случае эквивалентно) использовать идентификатор `this`:

```
on(press){
this._x += 10;
this._y += 10;
}
```

В этом сценарии `this` возвращает ссылку на объект, содержащий кнопку.

Если внутри блока кода обработчика событий кнопки `on()` содержится объявление переменной, то эта переменная прикрепляется к монтажной линейке, на которой находится данная кнопка.

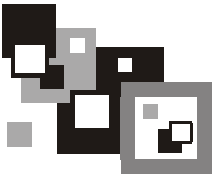
Для того чтобы из сценария кнопки обратиться к объекту, лежащему вне области видимости кнопки, необходимо использовать адресацию точно так же, как и при работе с переменными. При этом важно помнить, что все относительные ссылки,

используемые в блоке обработчика событий кнопки, разрешаются относительно монтажной линейки, содержащей эту кнопку, а не относительно самой кнопки. Для того чтобы из сценария кнопки обратиться к самой кнопке, необходимо задать ей имя и использовать его при обращении. Так, чтобы при нажатии кнопки сдвинуть только саму кнопку, а не родительский по отношению к ней объект, можно присвоить экземпляру этой кнопки имя `but` и прикрепить к ней следующий сценарий:

```
on(press){
but._x += 10;
but._y += 10;
}
```

Теперь при каждом нажатии на кнопку только она будет смещаться на 10 пикселов вправо и вниз.

Глава 11



Обработка событий клипов. Событийная модель ActionScript

Существует четыре типа людей:
— кто сидит спокойно и ничего не делает;
— кто говорит о том, что надо сидеть спокойно, и ничего не делать;
— кто делает;
— кто говорит о том, что надо делать.

Закон Ранэмона

Как уже говорилось в *главе 3*, любые команды ActionScript выполняются не сами по себе, а только в результате наступления определенного события. Таким событием может являться вхождение воспроизводящей головки в кадр, к которому прикреплен сценарий, либо выполнение определенных действий, связанных с пользователем (например, нажатие кнопки или перемещение курсора мыши), или с процессами, происходящими внутри фильма (например, окончание загрузки клипа).

Ряд встроенных объектов, таких как `Button`, `MovieClip`, `Mouse`, `Key`, `Stage` и др., в процессе воспроизведения фильма отслеживают различные процессы (как системные, так и связанные с действиями пользователя). Каждый из них является ответственным за оповещение о наступлении определенного события или набора событий. Когда в фильме выполняется определенное действие, ответственный за обнаружение этого действия встроенный объект генерирует соответствующее событие и оповещает о нем всех заинтересованных объектов.

В ходе воспроизведения фильма может происходить множество различных событий, однако далеко не каждое из них влечет за собой выполнение программного кода. Для того чтобы наступление события инициировало выполнение сценария, необходимо использовать обработчик событий — специальную инструкцию, определяющую, какие действия необходимо выполнить при обнаружении данного события.

Вопросы обработки событий являются чрезвычайно важными, в связи с чем описание событийной модели, используемой во Flash, авторы решили выделить в отдельную главу. В предыдущей главе мы уже коснулись вопроса обработки событий кнопок и описали события, которые генерируются классом `Button`. В настоящей главе эта тема получит продолжение в виде раздела, описывающего использование клипов в качестве кнопок. Здесь будет подробно рассмотрена тема обработки событий клипов, а также более широкие вопросы использования механизмов обработки и контроля событий.

Обработчик событий *on()*. Использование клипов в качестве кнопок

В *главе 10* были рассмотрены вопросы создания кнопки с использованием символа типа Button. Функциональность такой кнопки реализуется с помощью обработчика событий `on()`. В действительности для создания кнопки можно также использовать символ типа MovieClip с применением того же самого обработчика событий `on()`.

Создание клипа-кнопки

Объект `MovieClip` так же, как и объект `Button`, обладает возможностью обрабатывать набор событий, которые генерируются при выполнении определенных действий или во время протекания определенных процессов, связанных с экземплярами клипов. Начиная с шестой версии Flash (Flash MX) клипы получили возможность контроля событий, которые до этого могли обрабатываться исключительно обработчиком событий кнопок. Это существенно упростило решение многих практических задач и, в частности, привело к тому, что клипы практически полностью взяли на себя роль кнопок.

Экземпляру символа типа MovieClip, так же как и экземпляру символа типа Button, может быть назначен сценарий с использованием обработчика событий `on()`, который прикрепляется непосредственно к экземпляру клипа. В результате этого экземпляр клипа начинает функционировать как кнопка, реагируя на воздействия курсора и нажатие левой кнопки мыши. Такой клип называется *клип-кнопка*. Область реагирования клипа-кнопки совпадает с формой самого экземпляра. Если муви-клип содержит анимацию формы, то область реагирования такого клипа будет изменяться в соответствии с трансформацией входящих в его состав форм.

Для того чтобы внешний вид клипа-кнопки изменялся при воздействии курсора так же, как это происходит с обычной кнопкой, на его монтажной линейке должны находиться три кадра, описывающие его внешний вид при отсутствии взаимодействия, в момент наведения курсора мыши и в момент нажатия левой кнопки мыши. Для того чтобы данные кадры интерпретировались не как обычные кадры монтажной линейки, а как источники графики, описывающие состояние клипа-кнопки, их необходимо снабдить метками `_up`, `_over` и `_down` соответственно. В результате при наведении курсора на такой клип будет отображаться содержимое кадра с меткой `_over`, при нажатии левой кнопки мыши будет отображаться содержимое кадра с меткой `_down`, а в отсутствие взаимодействия будет отображаться содержимое кадра с меткой `_up`. Однако такое поведение клипа будет иметь место только после того, как на него будет наведен курсор. До наведения анимация монтажной линейки клипа будет непрерывно выполняться, как и анимация обычной монтажной линейки. Для того чтобы воспроизведение монтажной линейки клипа-кнопки не начиналось самопроизвольно, ее необходимо остановить с помощью глобальной функции `stop()` или метода `stop()` класса `MovieClip`.

Если внешний вид клипа-кнопки не должен изменяться при наведении и нажатии, то его монтажная линейка может состоять из одного-единственного кадра.

Выполнение описанных выше действий позволяет заставить клип практически полностью имитировать функциональность кнопки за исключением того, что область реагирования клипа-кнопки совпадает с его собственной формой, а область реагирования обычной кнопки задается в ее четвертом кадре **Hit** и может быть абсолютно произвольной. Однако существует возможность, позволяющая задать клипу-кнопке область реагирования произвольной формы и местоположения. Для этого необходимо применить свойство `hitArea` класса `MovieClip`. Данное свойство позволяет использовать в качестве области реагирования обратившегося к нему клипа другой клип. При этом используется следующий синтаксис:

```
клип-кнопка.hitArea = клип-область;
```

Здесь *клип-кнопка* — это имя экземпляра клипа-кнопки, для которого задается произвольная область реагирования, *клип-область* — это имя экземпляра клипа, исполняющего роль области реагирования. Если клип-область не существует, в качестве области реагирования будет использован сам клип-кнопка. С помощью свойства `hitArea` область реагирования клипа-кнопки может быть изменена в любой момент. Присваивание ссылки на новый клип свойству `hitArea` автоматически изменяет область реагирования клипа-кнопки, осуществляющего обращение к данному свойству. Следует отметить, что клип, используемый в качестве активной области, не обязательно должен быть видимым. Даже если его свойство `_visible` установлено в значение `false`, он все равно исполняет роль активной области.

Итак, в качестве резюме перечислим набор действий, которые должны быть выполнены для создания клипа-кнопки:

1. Прикрепление сценария с использованием обработчика событий `on()` к экземпляру символа типа `MovieClip`.
2. Если внешний вид кнопки должен меняться, как реакция на действия пользователя, необходимо создать на монтажной линейке клипа три кадра и назначить им метки `_up`, `_over` и `_down`.
3. Если область реагирования клипа-кнопки не должна совпадать с его графическим содержимым, необходимо создать новый клип и присвоить ссылку на него свойству `hitArea` клипа-кнопки.

С помощью обработчика событий `on()` клип так же, как и кнопка, может обрабатывать все стандартные события кнопки, описанные в табл. 10.2.

Область видимости обработчика событий *on()* клипа

Как было сказано в *главе 10*, областью видимости обработчика событий кнопки является монтажная линейка, на которой расположена эта кнопка. Тем не менее при использовании обработчика событий `on()` с клипом этот обработчик имеет иную область видимости.

Следует иметь в виду, что при использовании обработчика событий `on()` в клипах областью его видимости является не монтажная линейка, содержащая экземпляр клипа (как это происходит с кнопками), а монтажная линейка самого клипа. Это означает, что все относительные ссылки, используемые в блоке кода обработчика, разрешаются относительно монтажной линейки клипа, и все команды непосредст-

венно распространяются только на его монтажную линейку. Поскольку областью видимости обработчика событий `on()`, прикрепленного к клипу, является монтажная линейка этого клипа, применение идентификатора `this` в блоке кода обработчика возвращает ссылку на сам клип. Следующий сценарий клипа с именем `mc`, расположенного на основной монтажной линейке, выводит в панель **Output** строку, содержащую ссылку на себя.

```
//Сценарий клипа
on(press){
trace(this); /* Выводит _level0.mc Здесь _level0 - это идентификатор,
представляющий собой ссылку на _root */
}
```

Если внутри блока кода обработчика событий `on()` содержится объявление переменной, то эта переменная прикрепляется к монтажной линейке клипа, содержащего данный обработчик.

```
//Сценарий клипа mc
on(press){
var clipVar = 25; /* Переменная clipVar прикреплена к монтажной линейке
клипа mc */
}
```

Ниже приводятся два примера, иллюстрирующие различия в областях видимости обработчика `on()`, используемого в кнопках и клипах.

```
//Сценарий кнопки
on(press){
this._x += 10; // Сдвигает объект, на монтажной линейке которого
this._y += 10; // находится кнопка, содержащая этот сценарий
var clipVar = 25; /* Переменная clipVar прикреплена к монтажной линейке,
содержащей данную кнопку */
}
//Сценарий клипа
on(press){
this._x += 10; // Сдвигает сам клип, содержащий этот сценарий
this._y += 10;
var clipVar = 25; /* Переменная clipVar прикреплена к монтажной линейке
данного клипа */
}
```

Несмотря на то, что два приведенных сценария формально абсолютно идентичны, они оказывают различное действие в зависимости от своего местоположения. Если сценарий прикреплен к кнопке, при нажатии смещается объект, содержащий кнопку, вместе с самой кнопкой. Если тот же самый сценарий прикреплен к экземпляру муви-клипа, то при нажатии на него смещаться будет только сам клип. Это различие необходимо хорошо помнить и понимать.

Для того чтобы из сценария клипа обратиться к другой монтажной линейке, необходимо использовать адресацию. Так, например, для того чтобы из сценария клипа mc, расположенного на основной монтажной линейке, остановить ее воспроизведение, необходимо указать путь к _root, используя абсолютную или относительную адресацию:

```
//Сценарий клипа, расположенного на основной монтажной линейке
on(press){
// Абсолютная адресация
_root.stop(); // Останавливает анимацию основной монтажной линейки
// То же самое можно выполнить, используя относительную адресацию:
/*_parent.stop(); // Останавливает анимацию монтажной линейки, на которой расположен mc (в данном случае это - _root) */
}
```

События клипов

Кроме событий, обрабатываемых с помощью обработчика событий on(), существует целый ряд событий, которые генерируются объектом MovieClip и некоторыми другими объектами и могут обрабатываться с помощью обработчика событий клипов onClipEvent(). Эти события мы в дальнейшем будем называть *событиями клипов*.

При использовании обработчика onClipEvent() сценарий должен быть непосредственно прикреплен к клипу. Такой сценарий называется сценарием клипа. Синтаксис обработчика onClipEvent() за исключением названия идентичен синтаксису обработчика on().

```
onClipEvent(событие){
блок кода;
.....
}
```

Здесь *событие* — это название события, наступление которого повлечет за собой выполнение блока кода, ограниченного фигурными скобками.

Примечание

Следует помнить, что любой сценарий, прикрепленный к клипу, всегда начинается с обработчика событий on() или onClipEvent(). Все инструкции, выполняемые в результате воздействия на клип, должны быть помещены в блок кода обработчика событий клипа, ограниченный фигурными скобками. Ни одно предложение (или даже символ) в сценарии клипа, за исключением заголовка обработчика событий и комментариев, не может находиться вне блока обработчика событий. Нарушение этого правила вызовет ошибку на этапе компиляции.

Как и в случае с кнопками, прикрепление сценария непосредственно к экземпляру клипа с помощью обработчика onClipEvent() — это не единственный способ обработки событий клипов. Для этих целей также можно использовать методы обработ-

чиков событий, что позволяет поместить сценарий, выполняющий обработку клипа, в любом месте документа. Применение методов обработчиков событий будет рассмотрено далее в этой главе.

Итак, рассмотрим события, которые могут обрабатываться обработчиком `onClipEvent()`.

Событие *enterFrame*

Данное событие генерируется периодически с частотой, равной скорости воспроизведения, используемой в документе. Так, например, если в документе используется скорость воспроизведения 12 кадров в секунду (fps), то событие `enterFrame` генерируется 12 раз в секунду. Событие `enterFrame` является более совершенной альтернативой трехкадровому циклу, описанному в *главе 7*, поскольку позволяет реализовать многократное выполнение определенного набора инструкций с частотой, соответствующей скорости анимации, используемой в документе, и при этом не требует создания нескольких кадров и распределения по ним программного кода. Следующий сценарий, прикрепленный к клипу, периодически увеличивает исходное значение переменной `i` и выводит результаты в панель **Output** с частотой, равной скорости воспроизведения документа.

```
onClipEvent(enterFrame){
if(i == undefined) i = 0;
trace(i ++); // Выводит 0, 1, 2, 3, ......
}
```

Обратите внимание на первое предложение в блоке кода, заключенном в фигурные скобки. В связи с тем, что переменная `i` не была объявлена заранее, попытка ее инкрементирования выдаст значение `NaN`. Если же инициализировать ее внутри блока кода обработчика непосредственным образом, присвоив ей начальное значение, то это присваивание будет выполняться при каждом последующем выполнении кода и постоянного приращения значения не получится. В связи с этим мы выполняем проверку, определяя, имеется ли у переменной `i` значение, и если значения нет (`i` равна `undefined`), то мы задаем начальное значение, в противном случае в следующем предложении данное значение будет увеличиваться на единицу.

Следует отметить, что событие `enterFrame` генерируется периодически вне зависимости от количества кадров, расположенных на монтажной линейке, а также от того, выполняется ли ее воспроизведение или воспроизводящая головка остановлена. Это дает широкие возможности создания программной анимации. В качестве иллюстрации рассмотрим следующий простой пример. Приведенный сценарий клипа заставляет его постепенно изменять свою прозрачность от полной непрозрачности к полной прозрачности и наоборот. За счет использования события `enterFrame` эта программная анимация прозрачности будет выполняться вне зависимости от состояния анимации всех монтажных линеек документа.

```
onClipEvent(enterFrame){
/* При первом выполнении этого сценария происходит инициализация
переменной inc, используемой для задания шага изменения прозрачности */
```

```
if(inc == undefined) inc = 5;
/* Если прозрачность достигла предельного значения, изменяем направление
модификации */
if(this._alpha<= 0 || this._alpha >= 100) inc =-inc;
this._alpha += inc; // Модификация значения прозрачности
}
```

Необходимо отметить, что событие `enterFrame` не генерируется одновременно с событиями `load` и `unload`. Это обстоятельство оказывает существенное влияние на порядок выполнения блоков кода, распределенных по разным элементам документа.

Примечание

Следует запомнить, что блок кода в обработчике события `enterFrame` выполняется перед выполнением сценария монтажной линейки клипа, к которому прикреплен данный обработчик, и перед выполнением сценария кадра родительской монтажной линейки, содержащего данный клип. Это правило имеет место только в том случае, если клип уже загружен в проигрыватель, поскольку при первом появлении клипа в фильме событие `enterFrame` не генерируется.

Событие *load*

Событие `load` генерируется, как только клип впервые появляется на сцене, то есть сразу после его загрузки. Это событие обычно используется для инициализации переменных клипа или задания различных параметров, определяющих его начальное состояние. Если клип создается непосредственно в рабочей среде, то при воспроизведении фильма событие `load` генерируется при вхождении воспроизводящей головки в кадр, где впервые появляется данный клип. При вхождении воспроизводящей головки в кадр, где находится клип с обработчиком события `load`, имеет место следующий порядок выполнения кода.

1. Вначале выполняется сценарий кадра основной монтажной линейки, в котором находится клип с обработчиком.
2. После этого генерируется событие `load` и выполняется блок кода обработчика данного события, прикрепленного к клипу.
3. Затем выполняется код, расположенный на монтажной линейке самого клипа.
4. Событие `load` также генерируется в результате динамического создания клипа. Важно понимать, что появление клипа и обработка события `load` должны выполняться в одном и том же кадре, в связи с этим применение обработчика `onClipEvent()` позволяет избежать возможных трудностей, связанных с тем, что клип появляется в фильме раньше или позже, чем выполняется обработка его события `load`. Такая ситуация может иметь место при использовании метода обработчика события `onLoad`.

Рассмотрим простой пример использования данного события. Приведенный ниже сценарий клипа, задает ему 50%-ную прозрачность при первом появлении в фильме:

```
onClipEvent(load){
this._alpha = 50;
}
```

Событие *unload*

Событие `unload` генерируется при удалении клипа из фильма. Это событие возникает непосредственно после того, как клип удаляется с монтажной линейки, на которой он был расположен. Причиной удаления клипа может служить появление нового кадра, в котором этот клип отсутствует, либо его выгрузка из проигрывателя. Если удаление клипа связано с появлением нового кадра, то при этом имеет место следующий порядок выполнения кода.

1. В последнем кадре, содержащем клип, выполняется сценарий кадра.
2. Выполняется сценарий монтажной линейки клипа.
3. Выполняется блок кода обработчика события `unload`.
4. Происходит переход к следующему кадру и выполнение его сценария.

Событие `unload` используется для выполнения определенных действий в качестве реакции на удаление клипа. Так, например, после удаления клипа можно вернуть заданные с его помощью значения глобальных переменных в исходное состояние.

Событие *data*

Событие `data` генерируется при загрузке внешних данных с помощью методов `MovieClip.loadVariables()`, `MovieClip.loadMovie()` (или одноименных глобальных функций) или метода `MovieClipLoader.loadClip()`.

При использовании метода `MovieClip.loadVariables()` событие `data` генерируется после окончания загрузки всех переменных из внешнего источника (серверного сценария или текстового файла).

При использовании методов `MovieClip.loadMovie()` или `MovieClipLoader.loadClip()` событие `data` генерируется периодически по мере загрузки кадров внешнего SWF-документа. Для того чтобы проследить возникновение данного события, можно в среде тестирования использовать режим имитации загрузки (**View>Simulate Download**). Событие `data` генерируется после загрузки очередного кадра внешнего документа, однако частота наступления этого события не может превышать скорость воспроизведения документа, в котором находится обработчик события `data`. Событие `data` генерируется не более одного раза для каждого нового кадра. Таким образом, количество обнаружений события `data` прямо пропорционально числу кадров загружаемого документа и скорости воспроизведения документа, содержащего обработчик события `data`, и обратно пропорционально скорости соединения.

Событие *mouseMove*

Событие mouseMove генерируется непрерывно во время перемещения курсора мыши в пределах окна проигрывателя Flash Player независимо от смены кадров монтажной линейки. Частота генерирования данного события зависит от ресурсов процессора. Данное событие используется при необходимости выполнения действий, связанных с перемещением курсора мыши. Поскольку данное событие может быть обнаружено в промежутке между сменой кадров, а Flash выполняет обновление (перерисовку) содержимого сцены только при смене кадров, зачастую требуется выполнить принудительное обновление экрана. Для этих целей применяется глобальная функция updateAfterEvent(), которая может быть помещена в блок кода обработчика пользовательского события. Использование функции updateAfterEvent() позволит изменять содержимое сцены в соответствии с действиями, реализуемыми в сценарии, сразу же при обнаружении события mouseMove, а не дожидаться перехода к следующему кадру. Это обеспечивает более быстрый отклик системы на выполняемые пользователем действия.

Событие *mouseDown*

Данное событие генерируется при каждом нажатии левой кнопки мыши во время нахождения курсора в пределах окна проигрывателя. В отличие от события press, которое генерируется при нажатии кнопки мыши только над активной областью кнопки или клипа, содержащего обработчик этого события, событие mouseDown генерируется при нажатии кнопки над любой точкой внутри окна Flash Player.

Событие *mouseUp*

Данное событие генерируется всякий раз, когда левая кнопка мыши отпускается в момент нахождения курсора в пределах окна проигрывателя. Зачастую события mouseDown и mouseUp следуют друг за другом и в сочетании с событием mouseMove позволяют контролировать различные состояния мыши. Практическое применение обработки событий мыши демонстрируется в листинге 1.11, иллюстрирующем возможность создания пользовательского курсора.

Листинг 11.1. Пользовательский курсор

```
/* Как только клип-курсор загружен, помещаем его в координаты мыши и
отключаем отображение стандартного курсора */
onClipEvent(load){
this._x = _parent._xmouse;
this._y = _parent._ymouse;
Mouse.hide();
}
/* При перемещении курсора мыши (отображение которого отключено) клип
следует за ним */
onClipEvent(mouseMove){
```

```
this._x = _parent._xmouse;
this._y = _parent._ymouse;
updateAfterEvent();
}
// При нажатии левой кнопки мыши увеличиваем размер клипа в два раза
onClipEvent(mouseDown){
this._xscale = 200;
this._yscale = 200;
}
// При отпускании левой кнопки мыши устанавливаем прежний размер
onClipEvent(mouseUp){
this._xscale = 100;
this._yscale = 100;
}
// При выгрузке или удалении клипа возвращаем стандартный курсор
onClipEvent(unload){
Mouse.show();
}
```

Событие *keyDown*

Данное событие применяется для обнаружения нажатия любой клавиатурной клавиши. Если для обработки данного события используется обработчик `onClipEvent()`, то блок его кода непрерывно выполняется на протяжении времени нажатия любой клавиатурной клавиши, если окно проигрывателя находится в фокусе. Если же данное событие контролируется с помощью метода обработчика события `onKeyDown`, то для выполнения его блока кода необходимо, чтобы в момент нажатия клавиатурной клавиши в фокусе находилось не только окно проигрывателя, но и сам клип, использующий данный обработчик. Для передачи фокуса требуемому клипу можно воспользоваться методом `setFocus()` класса `Selection`. Вопросы организации порядка передачи фокуса будут подробно рассматриваться в *главе 14*.

Следует отметить, что данное событие не позволяет контролировать нажатие какой-то конкретной клавиши. Для решения этой задачи используются возможности, предоставляемые классом `Key`. Вопросы программного контроля клавиатуры будут подробно рассмотрены в *главе 17*.

Событие *keyUp*

Данное событие генерируется при отпускании нажатой клавиатурной клавиши. Если для обработки данного события используется обработчик `onClipEvent()`, то блок его кода выполняется сразу после отпускания нажатой клавиатурной клавиши, если окно проигрывателя находится в фокусе. Если же данное событие контролируется с

помощью метода обработчика события onKeyDown, то для выполнения его блока кода необходимо, чтобы при отпускании клавиши в фокусе находилось не только окно проигрывателя, но и сам клип, использующий данный обработчик.

Область видимости обработчика *onClipEvent()*

Областью видимости обработчика событий клипов onClipEvent() является монтажная линейка клипа, к которому прикреплен данный обработчик (так же как и у обработчика on(), используемого с клипом). Это означает, что все относительные ссылки, используемые в блоке кода обработчика, разрешаются относительно монтажной линейки клипа, и все команды непосредственно распространяются только на его монтажную линейку. Идентификатор this при его использовании в блоке кода обработчика onClipEvent() возвращает ссылку на клип, к которому прикреплен данный обработчик. Таким образом, при необходимости выполнения определенных действий с клипом для обращения к этому клипу из его собственного сценария можно использовать идентификатор this или вообще опустить указание целевого объекта. Если целевой объект в блоке обработчика onClipEvent() не указан явным образом, интерпретатор будет считать, что выполняется обращение к монтажной линейке клипа, к которому прикреплен данный обработчик.

Следующий сценарий клипа останавливает воспроизведение его собственной монтажной линейки при нажатии левой кнопки мыши в любой точке окна проигрывателя:

```
onClipEvent(mouseDown){
stop(); // Неявная ссылка клипа на себя
//this.stop(); Явная ссылка клипа на себя
}
```

Для того чтобы из сценария клипа обратиться к другой монтажной линейке, необходимо использовать адресацию. Изменим предыдущий сценарий, так чтобы нажатие кнопки мыши останавливало воспроизведение родительской по отношению к данному клипу монтажной линейки:

```
onClipEvent(mouseDown){
_parent.stop(); Остановка монтажной линейки, на которой находится клип
}
```

Применение методов обработчиков событий

К настоящему моменту у читателя уже имеется представление о большей (и основной) части событий кнопок и клипов и о возможностях контроля этих событий с помощью обработчиков on() и onClipEvent(). Мы выяснили, что обработчик on() — это универсальное средство, позволяющее контролировать одинаковый набор событий как для кнопок, так и для клипов. Обработчик событий onClipEvent() — это более специфическая инструкция, позволяющая контролировать набор событий исключительно для объектов MovieClip. Несмотря на существенные различия, эти

два обработчика имеют одно сходство — и тот и другой прикрепляются непосредственно к экземплярам объектов, с которыми связано обнаружение события. Использование обработчиков `on()` и `onClipEvent()` в сценарии требует обязательного прикрепления данного сценария к экземплярам объектов. В противном случае компилятор выдает сообщение об ошибке.

Прикрепление фрагментов кода к различным объектам фильма имеет очень существенный недостаток, связанный с усложнением и запутыванием структуры программного кода. Децентрализация кода, его распространение по нескольким элементам оборачиваются большими трудностями при поиске требуемого фрагмента кода и его отладке. По возможности необходимо стремиться к максимальной концентрации кода, это позволит значительно легче ориентироваться в программе, сделает программный код яснее и понятнее.

Обработчики `on()` и `onClipEvent()` обладают и еще одним серьезным недостатком — их нельзя прикрепить к кнопкам и клипам, созданным динамически, то есть программным образом с помощью методов `MovieClip.attachMovie()` и `MovieClip.createEmptyMovieClip()`.

В качестве альтернативы прикреплению обработчиков событий непосредственно к экземплярам ActionScript предоставляет возможность применения методов обработчиков событий.

Понятие метода обработчика событий

Метод обработчика события (event handler method) — это особый метод, определенный в пределах класса, объекты которого имеют возможность получать оповещение о наступлении соответствующего события. В отличие от обычных методов класса методы обработчиков событий не вызываются явным образом по требованию пользователя. Их вызов осуществляется только в момент обнаружения события, связанного с определенным обработчиком. Так, например, метод обработчика событий с именем `onEnterFrame`, определенный в классе `MovieClip`, вызывается периодически с частотой, равной скорости воспроизведения монтажной линейки документа. По умолчанию методы обработчиков событий не определены и не выполняют никаких действий. Для того чтобы в результате вызова обработчика, инициированного наступлением определенного события, выполнились какие-либо действия, необходимо присвоить свойству объекта, хранящему ссылку на соответствующий метод обработчика, функцию, определяющую эти действия. В результате этого при каждом вызове метода обработчика будет вызываться соответствующая пользовательская функция. Синтаксис применения метода обработчика событий имеет следующий вид:

```
объект.обработчик события = функция;
```

Здесь *`объект`* — это идентификатор объекта, выполняющего обработку события.

После точки указывается *`обработчик события`* — это свойство объекта *`объект`*, содержащее ссылку на метод обработчика.

И, наконец, *`функция`* — это литерал функции или идентификатор функции, которая должна выполняться при наступлении события, контролируемого данным обработчиком.

Рассмотрим простой пример использования метода обработчика, который называется onMouseDown и, как следует из его названия, позволяет обнаруживать нажатие левой кнопки мыши подобно тому, как это делает обработчик onClipEvent(). Следующий фрагмент кода, будучи размещенным в кадре, содержащем клип с именем mc, выводит в панель **Output** сообщение о каждом обнаружении события mouseDown.

```
mc.onMouseDown = function (){
        trace("Нажата левая кнопка мыши");
};
```

В данном случае мы воспользовались литералом функции. Тот же самый результат можно было бы получить, создав именованную функцию и затем присвоив ссылку на нее свойству onMouseDown. Это показано в следующем примере:

```
function detectMouseDown (){
        trace("Нажата левая кнопка мыши");
}
mc.onMouseDown = detectMouseDown;
```

Оба приведенных фрагмента эквивалентны. Обратите внимание, что во втором варианте в правой части выражения присваивания содержится ссылка на функцию, а не вызов функции, то есть оператор () здесь не используется.

Названия методов обработчиков событий кнопок и клипов начинаются с префикса on и, в основном, совпадают с названиями обрабатываемых событий, рассмотренных в *главах 10* и *11*. Классы Button и MovieClip имеют ряд одинаковых обработчиков, которые могут быть использованы как кнопками, так и клипами. В то же время клипы обладают набором собственных встроенных методов обработчиков, которые определены в классе MovieClip, но не поддерживаются классом Button. Некоторые методы обработчиков событий из представленных в табл. 11.1 могут также вызываться с помощью объекта _root. Перечень методов обработчиков событий кнопок и клипов представлен в табл. 11.1.

Таблица 11.1. *Методы обработчиков событий кнопок и клипов*

Обработчик событий	Вызывается объектом			Описание
	кнопка	клип	_root	
onPress	+	+	–	Вызывается, когда кнопка мыши нажата при нахождении курсора в области реагирования кнопки/клипа
onRelease	+	+	–	Вызывается, когда кнопка мыши нажата и затем отпущена при нахождении курсора в области реагирования кнопки/клипа
onRollOver	+	+	–	Вызывается, когда курсор входит в область реагирования кнопки/клипа

Таблица 11.1 *(продолжение)*

Обработчик событий	Вызывается объектом			Описание
	кнопка	клип	_root	
onRollOut	+	+	–	Вызывается, когда курсор выходит за пределы области реагирования кнопки/клипа при ненажатой кнопке мыши
onDragOut	+	+	–	Вызывается, когда курсор выходит за пределы области реагирования кнопки/клипа при нажатой кнопке мыши
onReleaseOutside	+	+	–	Вызывается, когда курсор выходит за пределы области реагирования кнопки/клипа при нажатой кнопке мыши, после чего кнопка мыши отпускается вне области реагирования
onDragOver	+	+	–	Вызывается, когда курсор выходит за пределы области реагирования кнопки/клипа при нажатой кнопке мыши, после чего вновь возвращается в область реагирования, кнопка мыши остается нажатой
onEnterFrame	-	+	+	Генерируется периодически с частотой, равной скорости воспроизведения, используемой в документе
onMouseMove	-	+	+	Генерируется непрерывно во время перемещения курсора мыши в пределах окна проигрывателя Flash Player независимо от смены кадров монтажной линейки
onKeyDown	+	+	–	Применяется для обнаружения нажатия любой клавиатурной кнопки
onKeyUp	+	+	–	Генерируется при отпускании нажатой клавиатурной кнопки
onMouseDown	–	+	+	Генерируется при каждом нажатии левой кнопки мыши во время нахождения курсора в пределах окна проигрывателя
onMouseUp	–	+	+	Генерируется всякий раз, когда левая кнопка мыши отпускается в момент нахождения курсора в пределах окна проигрывателя
onLoad	–	+	+	Генерируется, как только клип впервые появляется на сцене, то есть сразу после его загрузки

Таблица 11.1 *(окончание)*

Обработчик событий	Вызывается объектом			Описание
	кнопка	клип	_root	
onUnload	–	+	+	Генерируется при удалении клипа из фильма
onData	–	+	+	Генерируется при загрузке внешних данных в фильм
onSetFocus	+	+	-	Вызывается, когда происходит передача фокуса текущей(му) кнопке/клипу
onKillFocus	+	+	-	Вызывается, когда фокус переходит с текущей(го) кнопки/клипа на другой объект

Примечание

Несмотря на отмеченные выше недостатки обработчиков `on()` и `onClipEvent()`, в некоторых ситуациях их применение является наиболее удобным (если не единственным) подходом. Такая ситуация может иметь место при необходимости обработки некоторых событий кнопок и клипов. К их числу можно отнести: событие `load` (например, если появление клипа происходит перед выполнением сценария кадра с вызовом метода обработчика или при загрузке внешнего SWF-документа), событий `keyDown` и `keyUp` (в связи с возможными проблемами передачи фокуса объекту, вызывающему обработчик) и события `keyPress`, для которого вообще ненельзя определить метод обработчика.

Помимо возможностей централизации кода, применение методов обработчиков событий предоставляет следующие важные преимущества.

- Метод обработчика событий может быть легко переопределен в нужный момент. Для этого достаточно присвоить обработчику новую функцию, определяющую набор операций, которые он должен выполнить при обнаружении события.
- Методы обработчиков событий позволяют обрабатывать события объектов, созданных программным образом непосредственно в процессе воспроизведения фильма.
- Методы обработчиков событий могут быть удалены в нужный момент для того, чтобы прекратить выполняемые ими действия.

Вопросы области видимости методов обработчиков событий кнопок и клипов

Применение методов обработчиков событий позволяет контролировать события различных объектов из одного программного кода. При этом требуется четко представлять, что является областью видимости данных методов.

Метод обработчика событий определяется посредством присваивания пользовательской функции свойству объекта, содержащему ссылку на обработчик, поддерживае-

мый классом данного объекта. В связи с этим все переменные, объявленные в теле присваиваемой обработчику функции с помощью предложения `var`, являются локальными и существуют только в пределах локальной области видимости данной функции и только на протяжении времени ее выполнения. Это относится ко всем без исключения методам обработчиков событий.

Если в теле функции, присваиваемой обработчику, используется неявное объявление переменной (то есть без использования предложения `var`), то в этом случае переменная не является локальной и прикрепляется к монтажной линейке, содержащей объявление функции. Такая переменная продолжает существовать и после выполнения функции и может быть доступна другим сценариям.

Следующий пример иллюстрирует эти различия.

```
//Сценарий кадра основной монтажной линейки, содержащего клип mc
mc.onLoad = function(){
var local = 1; // Локальная переменная
notLocal = 1; // Нелокальная переменная, прикрепляется к _root
};
_root.onMouseDown = function(){ // При нажатии кнопки мыши...
trace (local); /* Выводит undefined, так как local - локальная переменная
и доступна только внутри тела функции, присваиваемой обработчику */
trace (notLocal); // Выводит 1
};
```

При использовании идентификатора `this` в блоке функции обработчика событий он возвращает ссылку на объект, для которого определяется метод этого обработчика. Это чрезвычайно важное обстоятельство будет широко применяться в дальнейшем при использовании адресации в теле метода обработчика! Так, в следующем примере `this` возвращает ссылку на клип `mc`. При наведении курсора мыши на этот клип он увеличивается в размерах, а при уходе курсора с клипа он приобретает исходный размер:

```
//Сценарий кадра, содержащего клип mc
mc.onRollOver = function(){ //При наведении на mc ......
       //Увеличение размера
       this._xscale = 300; // изменяем масштаб клипа mc
       this._yscale = 300;
};
mc.onRollOut = function(){ //При уходе курсора с mc ......
       //Возврат к исходному размеру
       this._xscale = 100; // изменяем масштаб клипа mc
       this._yscale = 100;
};
```

Если в теле функции обработчика событий используется непосредственное обращение к свойству или осуществляется вызов метода без указания объекта, то интерпре-

татор выполняет обращение к объекту, содержащему объявление функции обработчика. Так, в следующем примере при нажатии клипа с именем mc, расположенного на основной монтажной линейке, происходит остановка основной монтажной линейки и размер всего ее содержимого уменьшается в два раза. Такое поведение объясняется тем, что в теле функции не имеется явного указания на объект, к методу и свойствам которого выполняется обращение. Поэтому таким объектом является объект, содержащий объявление функции обработчика, то есть в данном случае _root.

```
//Сценарий кадра основной монтажной линейки
mc.onPress = function(){
       stop(); // Остановка основной монтажной линейки
       _xscale = 50; // Уменьшение размеров всего содержимого _root
       _yscale = 50;
};
```

Удаление методов обработчиков событий

Одно из преимуществ использования методов обработчиков событий заключается в том, что они могут быть удалены в нужный момент и, таким образом, выполнение действий, определяемых в функции обработчика, будет прекращено. Для удаления метода обработчика используется оператор delete. Рассмотрим это на простом примере. Следующий код заставляет клип с именем mc двигаться по горизонтали справа налево:

```
mc.onEnterFrame = function(){
       this._x += 10;
};
```

В определенный момент времени клип, движущийся по такому закону, должен уйти за пределы сцены и стать невидимым, поэтому можно дать пользователю возможность остановить его, щелкнув кнопкой мыши по движущемуся клипу. В следующем фрагменте кода определяется обработчик клипа onPress, который в случае нажатия на клип удаляет его обработчик onEnterFrame, тем самым заставляя клип остановиться:

```
mc.onPress = function(){
       delete this.onEnterFrame;
};
```

В листинге 11.2 представлен сценарий, развивающий описанный механизм. Его действие заключается в том, что при наведении курсора мыши на клип mc он начинает вращаться вокруг своей оси. При уходе курсора с клипа он продолжает поворачиваться до тех пор, пока угол его поворота вновь не достигнет 0. После этого вращение прекратится. При рассмотрении этого примера следует вспомнить, что идентификатор this, используемый в теле функции обработчика событий, возвращает ссылку на объект, определяющий данный обработчик.

Листинг 11.2. Применение методов обработчиков событий для программного контроля анимации вращения

```
mc.onRollOver = function() {
        // При наведении курсора мыши...
        this.onEnterFrame = function() {
                // Выполняем вращение клипа с шагом 30 градусов
                this._rotation += 30;
        };
};
mc.onRollOut = function() {
        // При уходе курсора мыши...
        this.onEnterFrame = function() {
                /* Выполняем вращение до тех пор, пока угол поворота не
                станет равным 0 (свойство _rotation всегда возвращает
                значение в диапазоне от -180 до 180, поэтому нам не
                приходится беспокоиться об исключении полных оборотов)*/
                this._rotation += 30;
                if (this._rotation == 0) {
                        delete this.onEnterFrame;
                }
        };
};
```

Применение методов обработчиков событий с использованием адресации

С помощью метода обработчика событий можно контролировать события объектов, расположенных в различных частях структуры документа из одного-единственного сценария. Для того чтобы определить метод обработчика событий объекта, расположенного не на той же самой монтажной линейке, где содержится сценарий, в котором выполняется обработка, необходимо использовать адресацию. В общем виде обработка события объекта, расположенного на любой монтажной линейке, имеет следующий синтаксис:

путь.обработчик события = функция;

Здесь *путь* — это путь от местоположения сценария к целевому объекту, событие которого необходимо контролировать. При указании пути к целевому объекту можно использовать абсолютную или относительную адресацию. Адресация может осуществляться с помощью точечного синтаксиса или с помощью оператора квадратных скобок `[]`. Правила адресации были подробно изложены в *разд. "Адресация переменных" главы 4.*

Особо следует отметить возможность динамически генерировать имя обработчика событий, используя оператор `[]`. Следующий пример иллюстрирует данную возможность:

```
var eventName:String = "Press"; //Имя события в виде строки
mc["on"+eventName] = function(){
       trace (eventName); // При нажатии на клип выводит Press
};
```

Теперь для того чтобы обработать другое событие, достаточно присвоить его название в виде строки переменной `eventName`:

```
var eventName:String = "Release"; //Имя события в виде строки
mc["on"+eventName] = function(){
trace (eventName); // При отпускании кнопки мыши на клипе выводит Release
};
```

Трансляция событий

Ряд встроенных классов ActionScript содержит обработчики, позволяющие объектам этого класса обнаруживать или перехватывать различные события, наступление которых может быть связано с действиями пользователя или с процессами, происходящими в фильме. Так, например, в классе `MovieClip` имеется обработчик `onEnterFrame`, в классе `Button` имеется обработчик `onRollOver`, в классе `Stage` — обработчик `onResize`, в классе `TextField` — обработчик `onChanged` и т. д. Некоторые события могут обрабатываться несколькими объектами, например, и в классе `MovieClip`, и в классе `Button` имеется обработчик `onPress`, позволяющий контролировать нажатие кнопки. Однако большинство событий может быть обнаружено только объектами определенного класса. Так, например, событие `enterFrame` генерируется с частотой, равной скорости воспроизведения фильма, и обрабатывается объектами `MovieClip`; событие `onMouseWheel` генерируется при вращении колеса прокрутки мыши объектом `Mouse` и т. д.

ActionScript предоставляет возможность передачи сообщения о событии, обнаруженном одним объектом, называемым *источником* (broadcaster), другому объекту, называемому *приемник* или слушатель (listener). В результате этого при обнаружении события объектом-источником (broadcaster) объект-приемник (listener) получает от него соответствующее уведомление и может выполнить определенные действия. Для того чтобы объект-приемник получил возможность принятия сообщений о событиях, генерируемых объектом-источником, его необходимо "зарегистрировать" на получение соответствующих сообщений от соответствующего объекта. Для того чтобы в ответ на получение уведомления объект-приемник мог выполнить определенный блок кода, его необходимо снабдить обработчиком событий. В принципе данная модель очень напоминает применение методов обработчиков событий за исключением двух существенных отличий.

- Объект, определяющий метод обработчика событий, сам не генерирует событие, он только получает уведомление о его наступлении от другого объекта (объекта-источника).

- Для того чтобы произвольный объект получил возможность реагировать на события, генерируемые объектом-источником, его необходимо зарегистрировать в этом объекте с помощью встроенного метода объекта-источника, который называется `addListener()`.

Данный подход применяется и во внутренней реализации функциональности встроенных объектов. Так, все экземпляры клипов зарегистрированы для получения уведомлений о событиях от объекта `MovieClip`. Кроме того, клипы также автоматически регистрируются для получения уведомлений о событиях от объектов `Mouse` и `Key`, что позволяет им обрабатывать такие события, как, например, перемещение курсора мыши (`onMouseMove`) или нажатие клавиатурной клавиши (`onKeyDown`).

Передача и обработка событий

Для реализации модели трансляции события выполняются следующие действия.

1. Вначале должен быть создан объект-приемник (listener), который будет принимать сообщение об обнаружении события от объекта-источника (broadcaster). В качестве объекта-приемника может выступать любой объект ActionScript, например, `Object`, `Array`, `MovieClip` и т. д.
2. Далее с помощью метода объекта-источника `addListener()` объект-приемник регистрируется для получения уведомления о наступлении событий, генерируемых объектом-источником.
3. После этого объект-приемник определяет метод обработчика событий, который инициирует выполнение указанного блока кода при получении сообщения о наступлении интересующего события от объекта-источника. Для этой цели используется синтаксис, идентичный использованию методов обработчиков событий.

Ниже приведен общий синтаксис реализации трансляции события объектом-источником и обработки данного события объектом-приемником.

1. Создание объекта-приемника. Если приемник — это экземпляр класса `MovieClip`, `Button` или `TextField`, применение конструктора является необязательным, поскольку экземпляры этих объектов создаются в рабочей среде

```
var приемник:тип = new конструктор ();
```

2. Регистрация объекта-приемника на получение уведомления о наступлении событий от объекта-источника

```
источник.addListener(приемник);
```

3. Обработка события *`EventName`* объектом-приемником

```
приемник.onEventName = function(){
блок кода
};
```

Здесь *`приемник`* — имя объекта-приемника (listener), *`источник`* — имя объекта-источника, `onEventName` — обработчик событий, инициирующий выполнение блока кода в теле присваиваемой функции при получении сообщения о наступлении события *`EventName`* от объекта-источника.

Название обработчика событий объекта-приемника формируется из префикса "on" и названия обрабатываемого события так же, как и при работе с методами обработчиков событий, например, onPress, onRelease, onKeyDown. Не все объекты, способные выполнять обработку событий, могут передавать сообщение об их наступлении другим объектам. В табл. 11.2 приведен перечень объектов, которые могут выступать источниками событий для других объектов (приемников), а также обработчики событий, позволяющие объектам-приемникам осуществлять контроль событий, генерируемых объектами-источниками.

Таблица 11.2. *Объекты-источники и обработчики событий*

Объект-источник	Обработчик события	Описание
Mouse	onMouseDown	Нажатие левой кнопки мыши
	onMouseMove	Перемещение курсора мыши
	onMouseUp	Отпускание кнопки мыши
	onMouseWheel	Вращение колеса мыши
Key	onKeyDown	Нажатие клавиатурной клавиши
	onKeyUp	Отпускание клавиатурной клавиши
Stage	onResize	Изменение размеров окна проигрывателя
TextField	TextField.onChanged	Изменение содержимого текстового блока
	TextField.onScroller	Прокрутка содержимого текстового блока
Selection	onSetFocus	Изменение фокуса ввода
MovieClipLoader	onLoadComplete	Окончание загрузки внешнего файла
	onLoadError	Невозможность выполнения или сбой в загрузке внешнего файла
	onLoadInit	Выполнение сценариев первого кадра загружаемого внешнего файла
	onLoadProgress	Запись порции загруженной информации на диск
	onLoadStart	Начало загрузки внешнего файла
	onLoadComplete	Окончание загрузки внешнего файла

Рассмотрим простой пример, в котором с помощью конструктора Object() создается объект obj, получающий уведомление о нажатии левой кнопки мыши от объекта

Mouse. При обнаружении данного события obj выводит соответствующее сообщение в панель **Output**.

```
// Создаем экземпляр объекта общего назначения на базе класса Object
var obj:Object = new Object();
// Регистрируем obj для получения уведомления о событиях от объекта Mouse
Mouse.addListener(obj);
// Определяем обработчик события mouseDown для объекта obj
obj.onMouseDown = function() {
        trace("Нажата левая кнопка мыши");
};
```

Теперь при каждом нажатии левой кнопки мыши в панели **Output** будет выводиться сообщение "Нажата левая кнопка мыши".

Поскольку объект obj уже зарегистрирован для получения уведомления о событиях от объекта Mouse, он может выполнять обработку любого из этих событий. Для того чтобы контролировать не только нажатие, но и отпускание кнопки мыши, можно добавить следующий код:

```
obj.onMouseUp = function() {
        trace("Левая кнопка мыши отпущена");
};
```

Следует отметить, что один и тот же объект-приемник может получать сообщения о событиях от различных объектов-источников, равно как один и тот же объект-источник может уведомлять о своих событиях несколько различных объектов-приемников. Зарегистрируем созданный выше объект obj для получения сообщений от объекта Key.

```
Key.addListener(obj);
```

Теперь объект obj имеет возможность реагировать на наступление событий, генерируемых объектом Key. Следующий фрагмент кода при нажатии клавиатурной клавиши заставляет объект obj выводить соответствующее сообщение в панель **Output**.

```
obj.onKeyDown = function() {
        trace("Нажата клавиша клавиатуры");
};
```

Таким образом, объект-приемник получает сообщения о наступлении событий от двух различных объектов Mouse и Key. Рассмотрим ситуацию, когда один и тот же объект-источник транслирует события нескольким приемникам. Для этого создадим объект Array, то есть массив (объект, созданный на базе класса Array), и сделаем так, чтобы при каждом нажатии левой кнопки мыши в этот массив добавлялся новый элемент, значение которого будет на единицу больше значения предыдущего элемента массива. Значение самого первого элемента положим равным нулю.

```
var ar:Array = new Array(); // Создаем объект-приемник, массив
Mouse.addListener(ar); /* Регистрируем массив для получения сообщений о
событиях от объекта Mouse */
```

```
// Добавляем в массив новый элемент при каждом нажатии левой кнопки мыши
ar.onMouseDown = function(){
       if(i==undefined) i = 0;
       this.push(i++);
};
```

Прекращение трансляции событий

Если передача уведомления о наступлении событий больше не требуется, ее можно прекратить с помощью метода объекта-источника `removeListener()`. Для этого используется следующий синтаксис:

```
источник.removeListener(приемник);
```

В результате применения метода `removeListener()` объект-приемник перестает получать уведомления о событиях объекта-источника.

В качестве примера применения данного метода несколько модифицируем предыдущий фрагмент кода, ограничив длину массива `ar` так, чтобы максимальное число его элементов не превысило бы значения 10. Для этого введем условное предложение, проверяющее значение переменной `i`. Как только значение этой переменной превысит 9, прекратим трансляцию событий для объекта `ar`.

```
ar.onMouseDown = function() {
       if (i == undefined) {
              i = 0;
       }
       if (i > 9) {
              Mouse.removeListener(this);
       }
       else {
              this.push(i);
              i++;
       }
};
```

Создание источников событий. Объект *AsBroadcaster*

В предыдущих разделах мы рассмотрели вопросы, связанные с механизмами обработки событий. Мы выяснили, что имеется ряд встроенных объектов, которые обладают возможностью обнаруживать наступление определенных событий и в результате этого обнаружения выполнять указанный блок кода. Кроме того, мы выяснили, что некоторые из этих объектов также обладают возможностью уведомлять другие

объекты об обнаружении определенных событий. В данном разделе мы рассмотрим, каким образом действует внутренний механизм передачи сообщения о наступлении события и как с помощью этого механизма можно заставить произвольный объект генерировать собственные события.

Внутренняя функциональность встроенных объектов ActionScript, связанная с их возможностью передавать сообщения о наступлении определенных событий, реализуется с помощью встроенного глобального объекта `AsBroadcaster`. Поскольку данный объект используется для внутренней реализации возможностей встроенных объектов, его описание отсутствует в официальной документации по Flash MX 2004 (как и в документации по предыдущей версии). Тем не менее его возможности могут быть чрезвычайно полезны при использовании их в собственных целях разработчика. Объект `AsBroadcaster` обладает четырьмя методами:

- `ASBroadcaster.initialize()`
- `ASBroadcaster.addListener()`
- `ASBroadcaster.removeListener()`
- `ASBroadcaster.broadcastMessage()`

Инициализация источника и трансляция событий

Создание объекта, служащего источником событий для других объектов, называется инициализацией источника событий. Метод `AsBroadcaster.initialize()` используется для того, чтобы снабдить объект возможностью транслировать события другим объектам. Его синтаксис имеет следующий вид:

```
AsBroadcaster.initialize(источник)
```

Здесь `источник` — это ссылка на объект, который должен транслировать свои события другим объектам.

В результате применения данного метода объект `источник`, переданный ему в качестве аргумента, приобретает три новых скрытых метода и одно новое скрытое свойство. Их скрытость выражается в том, что они не могут быть непосредственно обнаружены с помощью цикла `for...in`. Для обнаружения этих методов и свойства у объекта-источника нужно использовать функцию `ASSetPropFlags()` *(см. разд. "Цикл for.....in" главы 7)*. Перечислим эти методы и свойство и укажем их назначение:

- `_listeners` — скрытое свойство, представляющее собой массив, содержащий ссылки на все объекты-приемники, которые зарегистрированы для получения сообщений о событиях от объекта-источника. Всякий раз при регистрации очередного объекта-приемника для получения сообщений о событиях от объекта-источника, ссылка на него добавляется в массив `_listeners`;
- `addListener()` — данный метод уже рассматривался выше. Он применяется для регистрации объекта-приемника на получение уведомлений о событиях от объекта-источника;
- `removeListener()` — данный метод также был рассмотрен в предыдущих разделах. Он используется для прекращения трансляции сообщений о событиях объектом-источником;

- broadcastMessage() — данный метод используется для трансляции объектом-источником конкретного события всем объектам-приемникам, ссылки на которые хранятся в массиве _listeners объекта-источника, то есть всем объектам, зарегистрированным для получения сообщений.

Таким образом, в результате инициализации с помощью метода AsBroadcaster.initialize() любой объект получает возможность регистрировать другие объекты для приема сообщений о его событиях, отменять эту регистрацию и транслировать "слушающим" его объектам-приемникам (listeners) свои события.

Синтаксис методов addListener() и removeListener() был описан ранее. Рассмотрим синтаксис метода broadcastMessage():

источник.broadcastMessage(*обработчик события*, *параметр1*, ..., *параметрN*)

Здесь *источник* — это объект, транслирующий свои события другим объектам.

Параметр *обработчик события* — это строка, содержащая названия метода обработчика события, с помощью которого объект-приемник будет инициировать выполнение определенных действий при получении уведомления о соответствующем событии от объекта *источник*. Так, например, при нажатии клавиши клавиатуры встроенный объект Key вызывает метод broadcastMessage() с параметром onKeyDown. Метод broadcastMessage() проходит все элементы массива _listeners, для каждого из них вызывая обработчик, заданный параметром *обработчик события*. Таким образом осуществляется обработка всех объектов-приемников и вызов определенных в них методов обработчиков событий, инициирующих выполнение требуемых действий.

Остальные параметры метода broadcastMessage() — это произвольный набор значений, передаваемых методу обработчика событий объекта-приемника в качестве параметров.

Рассмотрим простой пример, иллюстрирующий принцип использования модели создания источника событий. В рамках данного примера создается объект myBroadcaster, который будет транслировать событие, контролируемое методом обработчика событий с названием onEventOccur. Во время трансляции события методу onEventOccur объекта-приемника в качестве параметра будет передана строка "Событие обнаружено". Приведем последовательность выполняемых действий.

1. Создание объекта myBroadcaster.
2. Инициализация объекта-источника myBroadcaster с помощью метода ASBroadcaster.initialize().
3. Создание объекта-приемника.
4. Регистрация объекта-приемника для получения уведомления о наступлении события от объекта myBroadcaster.
5. Определение метода обработчика события onEventOccur объектом-приемником.
6. Трансляция события объектом-источником с помощью метода broadcastMessage().

```
/* Создание объекта, который будет использоваться в качестве источника
событий */
var myBroadcaster:Object = new Object();
//Инициализация объекта myBroadcaster в качестве источника событий
AsBroadcaster.initialize(myBroadcaster);
//Создание объекта-приемника
var myListener:Object = new Object();
/* Добавление нового приемника в массив приемников (_listeners)
объекта-передатчика */
myBroadcaster.addListener(myListener);
/* Обработка пользовательского события приемником. Метод обработчика
события принимает строковый параметр mes, значение которого будет
передаваться при трансляции события методом broadcastMessage() */
myListener.onEventOccur=function(mes:String){
       trace(mes);
};
/* Передача сообщения о наступлении пользовательского события и передача
фактического параметра */
myBroadcaster.broadcastMessage("onEventOccur", "Событие обнаружено");
```

В результате выполнения этого сценария в панель **Output** выводится сообщение `"Событие обнаружено"`.

Применение объекта *AsBroadcaster*

Возможность снабдить любой объект способностью транслировать события может быть использована в ряде случаев. Так, например, встроенный класс `MovieClip` не может быть использован в качестве источника событий для объектов, являющихся экземплярами других классов. Тем не менее с помощью объекта `AsBroadcaster` можно реализовать трансляцию событий клипа другим объектам. Рассмотрим это на примере события `enterFrame`, трансляция которого могла бы в некоторых ситуациях принести определенную пользу.

В следующем сценарии реализуется трансляция события `enterFrame` объекту-приемнику общего назначения (то есть экземпляру класса `Object`), который выполняет обработку данного события и периодически выводит в панель **Output** значение переменной `i`, которое увеличивается на единицу с частотой, равной скорости воспроизведения фильма.

```
/* Инициализируем встроенный объект MovieClip в качестве источника
событий */
AsBroadcaster.initialize(MovieClip);
// Создаем объект-приемник objListener
var objListener:Object = new Object();
```

```
/* Регистрируем объект-приемник для получения уведомления о событиях от
объекта MovieClip */
MovieClip.addListener(objListener);
/* Обработка события, уведомление о котором получено от
объекта-источника объектом-приемником objListener */
objListener.onEnterFrame = function() {
        if (i == undefined) {
                i = 0; // Инициализируем i, если она еще не определена
        }
        trace(i++);
};
/* Создаем новый пустой клип с именем clipEventSource и размещаем его на
ближайшем свободном уровне стека */
this.createEmptyMovieClip("clipEventSource", this.getNextHighestDepth());
/* Транслируем событие всем зарегистрированным объектам-приемникам
(которое может быть обработано ими с помощью определенных в этих объектах
методов обработчиков onEnterFrame()) с частотой, равной скорости
воспроизведения, используемой в документе */
clipEventSource.onEnterFrame = function() {
        MovieClip.broadcastMessage("onEnterFrame");
};
```

Поскольку в данном примере требуется осуществление трансляции события, которое генерируется периодически и непрерывно, значит, метод `broadcastMessage()` также должен вызываться с частотой, равной скорости воспроизведения фильма. Проще всего это можно реализовать, используя экземпляр клипа, который вызывает метод `broadcastMessage()` с помощью своего метода обработчика события `onEnterFrame()`. В приведенном примере этот клип создается динамически с использованием метода `MovieClip.createEmptyMovieClip()`, создающего клип без графического содержимого (этот метод будет подробно рассмотрен в *главе 12*).

Рассмотрим еще один пример, иллюстрирующий более конкретное и практически применимое использование возможностей класса `AsBroadcaster` для контроля события "`onDoubleClick`", которое генерируется при двойном щелчке мыши. Это событие не обрабатывается классом `Mouse`, тем не менее его наличие часто могло бы оказаться весьма полезным. Листинг 11.3 содержит сценарий, позволяющий реализовать контроль двойного щелчка мыши. При каждом двойном щелчке мыши в панель **Output** будет выводиться сообщение об обнаружении двойного щелчка, его номер, а также интервал между щелчками, указанный в миллисекундах.

Листинг 11.3. Контроль двойного щелчка мыши

```
var counter:Number = 0;//Счетчик двойных щелчков
var delay:Number = 300;/*Допустимая задержка между щелчками
в миллисекундах */
```

```
//Создаем объект-передатчик
var myBroadcaster:Object = new Object;//Создание пустого объекта
AsBroadcaster.initialize(myBroadcaster);//Инициализация созданного
объекта в качестве передатчика событий
//Создаем объект-приемник
var myListener:Object = new Object();
//Добавление нового приемника в массив приемников объекта-передатчика
myBroadcaster.addListener(myListener);
Mouse.addListener(myListener);
//Передача сообщения о наступлении пользовательского события и передача
//параметра
myListener.onMouseDown = function() {
       clickInterval = getTimer() - clickTime; /* Временной интервал
       между текущим и предыдущим щелчками */
       clickTime = getTimer(); // Время щелчка
/*Если задержка между щелчками меньше заданной в delay., транслируем
сообщение о наступлении события "onDoubleClick" и передаем аргумент
методу обработчика. Аргумент - это строка сообщения об обнаружении
двойного щелчка и его порядковый номер. При первом щелчке выражение
(getTimer() - clickTime) разрешается в NaN, так как clickTime еще не
определена, поэтому проверочное выражение разрешается в false */
       if (clickInterval < delay) {
              myBroadcaster.broadcastMessage("onDoubleClick", "Двойной↵
щелчок #" + (++counter) + "\rзадержка между щелчками: " + clickInterval);
/*После трансляции события модифицируем clickTime так, чтобы при
следующем щелчке clickInterval оказывался бы больше, чем значение delay
и событие "onDoubleClick" не транслировалось. Если этого не сделать, то
в случае, когда щелчки производятся последовательно с частотой, меньшей
delay, событие "onDoubleClick" будет транслироваться при каждом нажатии
мыши, а не при двойном щелчке.*/
              clickTime = -delay;
       }
};
//Обработка пользовательского события "onDoubleClick" объектом-приемником
myListener.onDoubleClick = function(mes:String) {
       trace(mes); // Выводим сообщение в панель Output
       mc._x += 10; /* Смещаем клип mc вправо на 10 пикселов при каждом
       двойном щелчке мыши */
};
```

Приведем некоторые дополнительные пояснения к данному сценарию. Двойной щелчок мыши представляет собой два последовательных щелчка, временной интервал между выполнением которых не превышает разрешенного разработчиком (или выбранного пользователем) значения. Кроме того, при выполнении ряда последова-

тельных щелчков важно, чтобы регистрировался только двойной щелчок. Так, если пользователь три раза нажал левую кнопку мыши с одинаковым (и допустимым) временным интервалом, то двойной щелчок должен регистрироваться только при втором щелчке и ни в коем случае не при третьем.

В приведенном сценарии переменная `delay` задает максимальный временной интервал (в миллисекундах) между щелчками, при котором они могут считаться двойными.

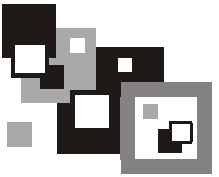
Для оценки реального интервала между выполняемыми пользователем щелчками применяется глобальная функция `getTimer()`. Функция `getTimer()` не принимает параметров и возвращает количество миллисекунд, прошедших с момента начала воспроизведения SWF-документа; таким образом, возвращаемое ею значение постоянно увеличивается по мере воспроизведения фильма. Когда пользователь щелкает кнопкой мыши, мы сохраняем значение, возвращаемое функцией `getTimer()`, в переменной `clickTime`. Эта переменная служит точкой отсчета, которую при следующем нажатии кнопки мыши можно сравнить с новым значением, возвращаемым функцией `getTimer()`. В переменной `clickInterval` при каждом нажатии кнопки мыши сохраняется временной интервал между временем текущего и предыдущего нажатия кнопки мыши. Обратите внимание на следующие строки:

```
clickInterval = getTimer() - clickTime; /* Временной интервал между
текущим и предыдущим щелчками */
clickTime = getTimer(); // Время текущего щелчка
```

В первом предложении мы вычисляем интервал между щелчками. При этом переменная `clickTime` хранит время выполнения предыдущего щелчка. В следующем предложении мы модифицируем ее значение, сохраняя в ней время текущего щелчка), с тем, чтобы при последующем щелчке можно было бы снова вычислить интервал между щелчками. Обратите внимание, что при самом первом нажатии кнопки мыши `clickInterval` будет иметь значение `undefined`, так как переменная `clickTime` еще не определена, а проверочное выражение условного предложения будет возвращать значение `false`. Поэтому при самом первом щелчке событие `"onDoubleClick"` транслироваться не будет.

Для того чтобы щелчок, следующий сразу за двойным щелчком, не приводил к трансляции события `"onDoubleClick"`, после обнаружения двойного щелчка (если проверочное выражение условного предложения дает `true`) мы присваиваем переменной `clickTime` значение `-delay` для того, чтобы при следующем щелчке значение переменной `clickInterval` оказалось больше, чем значение `delay`. Формально это означает, что между двойным щелчком и следующим за ним одинарным щелчком имеется интервал, превышающий допустимый интервал для двойных щелчков (`delay`). Поэтому щелчок, следующий за двойным щелчком, не вызывает трансляции события `"onDoubleClick"`.

Глава 12



Класс *MovieClip*. Работа с клипами

Оптимисты изобретают самолет,
пессимисты — парашют.

Закон двух подвохов Фортена

MovieClip — это один из базовых объектов рабочей среды Flash, представляющий собой ее наиболее мощный и гибкий инструмент. Характерной особенностью экземпляра символа типа MovieClip или попросту клипа является то, что он, как и кнопка, создается в рабочей среде. При создании клипа в рабочей среде Flash автоматически создает экземпляр встроенного класса `MovieClip`, обладающий полным набором свойств и методов, определенных в пределах данного класса. Кроме создания экземпляра клипа, в рабочей среде существует ряд возможностей, предоставляемых классом `MovieClip`, которые позволяют создавать экземпляр клипа программным образом, то есть динамически.

Экземпляр клипа, как и экземпляр кнопки, имеет уникальный идентификатор, называемый именем экземпляра (instance name). Для того чтобы задать имя экземпляру клипа, созданному в рабочей среде, необходимо выделить его на сцене и ввести имя, отвечающее требованиям к идентификаторам, в поле **Instance Name** панели **Properties**. Если имя экземпляра не будет задано разработчиком, оно будет сгенерировано автоматически.

Целый ряд свойств и методов класса `MovieClip` является общим для некоторых других классов. Так, набор свойств класса `Button` является подмножеством свойств, содержащихся в классе `MovieClip`. В данной главе представлено подробное описание возможностей класса `MovieClip`. Описание некоторых свойств и методов этого класса, имеющих отношение к понятиям, рассматриваемым в дальнейших главах, будет приведено позднее. Описание средств динамического рисования, предоставляемых классом `MovieClip`, ввиду обширности темы решено вынести в отдельную главу *(см. главу 13)*.

Встроенные свойства класса *MovieClip*

Свойства класса `MovieClip` предоставляют широкие возможности контроля состояния и разнообразных параметров клипов. Доступ к свойству осуществляется непо-

средственно из экземпляра. Для обращения к свойству клипа применяется точка или оператор []. При этом используется одна из следующих форм синтаксиса:

```
клип.свойство
клип["свойство"]
```

Здесь клип — это имя экземпляра клипа, к свойству которого происходит обращение, свойство — это имя свойства клипа.

Координаты. Свойства _x и _y

Свойства _x и _y содержат координаты клипа по горизонтали и вертикали соответственно и используются для контроля его местоположения и для позиционирования. Свойства _x и _y доступны как для чтения, так и для записи. Каждое из этих свойств содержит значение типа Number, измеряемое в пикселах.

Координаты объекта, хранящиеся в свойствах x и y, — это координаты точки регистрации символа, которые отсчитываются относительно начала координат монтажной линейки, содержащей экземпляр, осуществляющий обращение к этим свойствам. Поскольку местоположение точки регистрации относительно графического содержимого символа может быть абсолютно произвольным, необходимо иметь в виду, что с координатами объекта ассоциируются именно координаты его точки регистрации (перекрестия), а не, скажем, его геометрический центр. Поэтому при написании сценария необходимо четко представлять, где находится точка регистрации клипа.

Координаты клипа всегда отсчитываются относительно начала координат его родительского объекта. Если экземпляр клипа (кнопки или текстового блока) находится на основной монтажной линейке (_root), его координаты отсчитываются относительно левого верхнего угла рабочей области. Если экземпляр клипа находится на монтажной линейке другого клипа, то его координаты отсчитываются относительно точки регистрации родительского клипа.

Примечание

Следует помнить, что во Flash положительная полуось абсцисс (*X*) направлена слева направо, а положительная полуось ординат (*Y*) направлена сверху вниз.

Следующий сценарий кадра центрирует клип mc, точка регистрации которого находится в его центре относительно границ рабочей области, размеры которой составляют 550×400 пикселов:

```
mc._x = 275;
mc._y = 200;
```

Используя свойства _x и _y, можно выполнять перемещение клипов, то есть создавать *программную анимацию* движения. Для реализации перемещения объекта необходимо задать закон движения, определив соответствие между координатами *X* и *Y* и осуществлять последовательное изменение значения координаты-аргумента. Для этих целей часто используется обработчик onEnterFrame. Следующий сценарий реа-

лизует перемещение клипа mc слева направо по синусоидальной траектории, амплитуда которой составляет 100 пикселов. Траектория движения задается функцией:

$$y = A/2 \cdot \sin(x),$$

где A — это амплитуда синусоиды (то есть расстояние между ее максимумом и минимумом).

При достижении правой границы рабочей области (ширина которой составляет 550 пикселов) клип вновь начинает движение с левого края. Поскольку метод Math.sin() в качестве параметра принимает значение в радианах, мы выполняем преобразование из градусов в радианы:

```
mc.onEnterFrame = function() {
        this._x += 5;
        this._y = 50 * (Math.sin(this._x / 180 * Math.PI) + 1);
        if (this._x >= 550) {
                this._x = 0;
        }
};
```

В связи с тем, что координаты клипа (или кнопки) регистрируются относительно начала координат родительского объекта (объекта, на монтажной линейке которого он находится), при трансформации родительского объекта выполняется трансформация его системы координат. Так, если клип mc1 расположен на монтажной линейке клипа mc0, который повернут на определенный угол, то система координат клипа mc1 также поворачивается на данный угол.

Примечание

Если координаты объекта имеют дробные значения, это может привести к тому, что границы этого объекта будут выглядеть несколько размытыми и нерезкими. Кроме того, при выполнении вычислений за счет накопления погрешности результат может оказаться неточным. В связи с этим необходимо по возможности стремиться к использованию целочисленных значений координат объектов.

Свойства _x и _y также можно применять для управления местоположением всего содержимого основной монтажной линейки. Для этого при обращении к данным свойствам используется ссылка на основную монтажную линейку _root или они используются в сценарии основной монтажной линейки непосредственно без указания объекта. Так, следующий фрагмент кода сдвигает все содержимое сцены на 100 пикселов вправо и вниз:

```
_root._x += 100;
_root._y += 100;
```

Для получения того же результата в кадре основной монтажной линейки можно было написать следующее:

```
_x += 100;
_y += 100;
```

Размеры. Свойства _width, _height, _xscale и _yscale

Для программного контроля размеров клипа (а также кнопки и текстового блока) используются четыре свойства. Два из них определяют размеры объекта в абсолютных единицах, то есть в пикселах, и два — в относительных, то есть в процентах.

Абсолютные размеры

Свойства `_width` и `_height` используются для получения или задания размеров клипа, измеряемых в пикселах, по горизонтали и по вертикали соответственно. Данные свойства принимают неотрицательные числовые значения. Размеры клипа оцениваются как размеры ограничивающей рамки, включающей все графические объекты, расположенные в клипе (рис. 12.1). Местоположение точки регистрации не влияет на значения этих свойств.

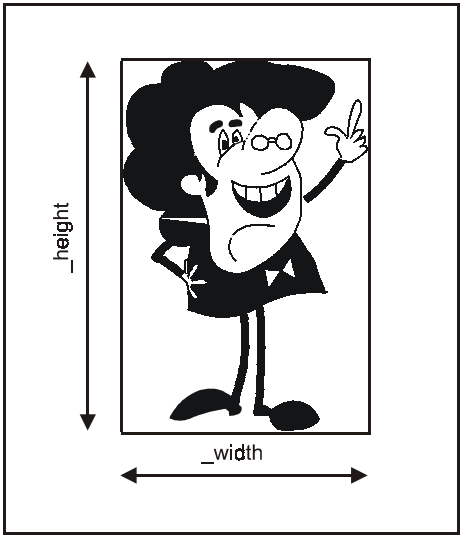


Рис. 12.1. Оценка размеров клипа

Следует иметь в виду, что масштабирование объекта выполняется относительно его точки регистрации. Местоположение центра трансформации (transformation center), заданное в рабочей среде, не оказывает никакого влияния на результат масштабирования, осуществляемого программным образом.

Некоторые сложности с контролем размеров объектов могут возникнуть в ситуации, когда клип содержит объект, состоящий из векторной формы (заливки) и контура (обводки). В этом случае абсолютные значения размеров клипа, которые отображаются на панели **Properties** в рабочей среде, и значения свойств `_width` и `_height`

могут различаться. В подобной ситуации необходимо руководствоваться следующими соображениями.

- Если обводка имеет стиль **hairline**, размеры объекта совпадают с размерами его заливки. Это происходит вследствие того, что формально у линии **hairline** отсутствует толщина.
- При использовании остальных стилей обводок размеры объекта, которые указываются в рабочей среде, не совпадают с размерами, определяемыми свойствами `_width` и `_height`. В рабочей среде размеры объекта соответствуют размерам его заливки. При программной обработке размеров ширина или высота объекта определяется как сумма ширины или высоты его заливки плюс толщина обводки. Это связано с тем, что если толщина обводки не является нулевой (стиль **hairline)**, то половина обводки лежит на заливке, а другая половина выходит за пределы заливки (рис. 12.2).

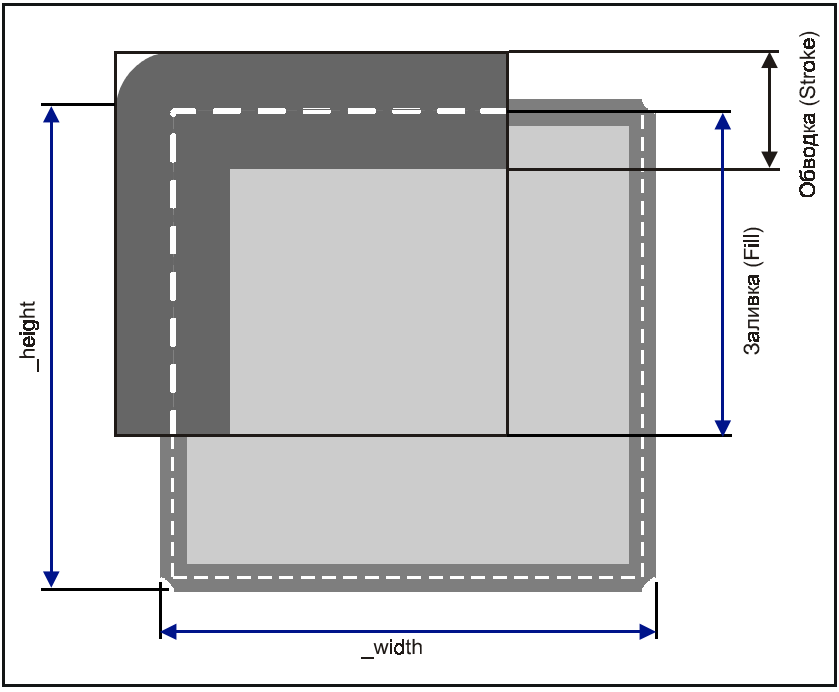


Рис. 12.2. Оценка размеров объекта, состоящего из заливки и обводки

В качестве иллюстрации сказанного рассмотрим следующий пример. На сцене имеется два клипа с именами `mc1` и `mc10`. Каждый из них представляет собой прямоугольник размером 100×50 пикселов. В клипе `mc1` прямоугольник имеет обводку толщиной 1 пункт, а в клипе `mc10` используется обводка толщиной 10 пунктов. Несмотря на то, что видимые размеры клипов отличаются (из-за разной толщины обводки), в панелях **Properties** и **Info** их размеры отображаются равными100×50 пиксе-

лов, то есть толщина обводки вообще не принимается в расчет. В этом легко убедиться, если нарисовать на сцене горизонтальную (или вертикальную) линию и увидеть, что ее высота (или ширина), отображаемая в панели **Properties**, имеет значение 0.

Тем не менее свойства `_width` и `_height` содержат информацию о реальном (то есть видимом) размере объекта. Для того чтобы в этом убедиться, поместим в кадр, где находятся клипы `mc1` и `mc10`, такой код:

```
trace("mc1._width="+mc1._width);
trace("mc1._height="+mc1._height);
trace("mc10._width="+mc10._width);
trace("mc10._height="+mc10._height);
```

В панель **Output** будет выведено следующее:

```
mc1._width=101 //Ширина заливки плюс толщина обводки
mc1._height=51 //Высота заливки плюс толщина обводки
mc10._width=110 //Ширина заливки плюс толщина обводки
mc10._height=60 //Высота заливки плюс толщина обводки
```

Использование обводки с разной толщиной в нескольких клипах может приводить к тому, что, несмотря на равенство значений их свойств `_width` и `_height`, видимые размеры этих клипов будут отличаться. Для того чтобы избежать подобной проблемы, можно воспользоваться следующими советами.

- Используйте обводки со стилем **Hairline**.
- При необходимости использования обводки с ненулевой толщиной (или несплошной обводки) ее нужно преобразовать в заливку. Для этого используется команда главного меню **Modify>Shape>Convert Lines to Fills...**.

В листинге 12.1 представлена функция `matchWidthAndHeight()`, позволяющая привести размеры двух переданных ей в качестве параметров клипов (кнопок или текстовых блоков) к одинаковым значениям. В данном примере за базовые размеры берутся большие ширина и высота.

Листинг 12.1. Приведение размеров двух объектов

```
function matchWidthAndHeight(clip1:Object, clip2:Object):Void{
       var maxWidth:Number = Math.max(clip1._width, clip2._width);
       var maxHeight:Number = Math.max(clip1._height, clip2._height);
       clip1._width = clip2._width = maxWidth;
       clip1._height = clip2._height = maxHeight;
}
```

Примечание

Если размеры объекта имеют дробные значения, это может привести к тому, что его границы будут выглядеть размытыми. Кроме того, за счет накопления погрешности при выполнении вычислений результат может оказаться неточным. В связи с этим необходимо по возможности стремиться к использованию целочисленных значений размеров.

Относительные размеры

В некоторых ситуациях бывает значительно удобнее задавать размеры объекта, используя не абсолютные, а относительные значения, которые рассчитываются на основе исходных размеров объекта. Для этих целей применяются свойства _xscale и _yscale, которые содержат, соответственно, ширину и высоту объекта, определяемую в процентном соотношении к его исходной ширине и высоте.

Исходные ширина и высота клипа определяются в среде редактирования символа как ширина и высота ограничивающей рамки, заключающей все его графическое наполнение. Исходные ширина и высота клипа также могут быть определены, как ширина и высота его нетрансформированного экземпляра.

Свойства _xscale и _yscale имеют значения типа Number и могут использоваться как для чтения, так и для записи. Необходимо иметь в виду, что масштабирование выполняется относительно точки регистрации объекта. В связи с этим, если требуется, чтобы изменение размеров происходило равномерно во всех направлениях, необходимо, чтобы точка регистрации находилась в центре клипа. Если требуется, чтобы при масштабировании граница клипа оставалась на месте, необходимо, чтобы точка регистрации лежала на этой границе.

Следует отметить, что трансформация экземпляра в рабочей среде с помощью инструмента **Free Transform** или панели **Transform** влияет на значения свойств _xscale и _yscale. Так, если экземпляр клипа в рабочей среде пропорционально уменьшен в два раза, значения его свойств _xscale и _yscale будут составлять 50%. Для того чтобы масштабировать объект без изменения значений свойств _xscale и _yscale, необходимо выполнять масштабирование в среде редактирования символа (то есть трансформировать не экземпляр, а сам символ в библиотеке).

Следующий пример иллюстрирует применение свойств _xscale и _yscale. При нажатии на клип mc его размер пропорционально увеличится в два раза.

```
mc.onPress = function(){
       this._xscale = 200;
       this._yscale = 200;
};
```

Можно дополнить этот код так, чтобы при отпускании кнопки размеры клипа снова вернулись к исходным значениям. Для этого допишем следующий фрагмент:

```
mc.onRelease = function(){
       this._xscale = 100;
       this._yscale = 100;
};
```

В листинге 12.2 представлен сценарий, заставляющий клип mc при наведении курсора мыши постепенно увеличивать размер до тех пор, пока он не достигнет величины, в два с половиной раза превышающей исходный размер. При уходе курсора мыши с клипа его размер постепенно уменьшается, пока не достигнет исходной величины. Как только масштаб клипа достигнет предельной величины (то есть мак-

симального или минимального значения), удаляем метод обработчика `onEnterFrame`, для того чтобы не занимать ресурсы проигрывателя.

Листинг 12.2. Плавное увеличение клипа при наведении курсора мыши

```
mc.onRollOver = function() {
       this.onEnterFrame = function() {
              if (this._xscale < 250) {
                     this._xscale += 20;
                     this._yscale = this._xscale;
              }
              else delete this.onEnterFrame;
       };
};
mc.onRollOut = function() {
       this.onEnterFrame = function() {
              if (this._xscale > 100) {
                     this._xscale -= 20;
                     this._yscale = this._xscale;
              }
              else delete this.onEnterFrame;
       };
};
```

Свойства `_xscale` и `_yscale` могут быть использованы для выполнения отражения объекта относительно горизонтальной или вертикальной оси, проходящей через его точку регистрации. Для этого им нужно задать отрицательные значения. Значение −100 позволит выполнить точное зеркальное отражение. Отрицательные значения, отличные от ста, будут выполнять отражение одновременно с масштабированием. В листинге 12.3 представлена функция `reflect()`, выполняющая отражение объекта, ссылка на который передается ей в качестве первого параметра. Два остальных параметра определяют направление отражения и являются необязательными. Второй параметр влияет на отражение по горизонтали, а третий — на отражение по вертикали. Если значение параметра `true` или параметр отсутствует, предполагается, что отражение в соответствующем направлении выполняется. Если значение параметра `false`, отражения не происходит.

Листинг 12.3. Выполнение зеркального отражения объекта по горизонтали и/или по вертикали

```
function reflect (clip:Object, x:Boolean, y:Boolean):Void{
       if(x || x == undefined) clip._xscale = -100;
       if(y || y == undefined) clip._yscale = -100;
}
```

Поворот. Свойство *_rotation*

Свойство _rotation позволяет контролировать угол поворота клипа, кнопки или текстового блока. Данное свойство используется как для чтения, так и для записи и имеет значение типа Number, которое указывает угол поворота объекта, заданный в градусах. Положительные значения угла поворота соответствуют повороту по часовой стрелке, отрицательные значения соответствуют повороту против часовой стрелки. Значения, возвращаемые свойством _rotation, всегда лежат в диапазоне от −180 до 180. Если данному свойству присваивается значение, выходящее из этого диапазона, оно автоматически преобразуется так, чтобы попадать в диапазон [−180; 180]. Такое преобразование выполняется за счет того, что из заданного значения вначале исключаются полные обороты (на 360°). Если остаток оказывается положительным и имеет значение меньше 180°, он прибавляется к нулю. Если остаток положительный и имеет значение больше 180°, то угол определяется как отрицательная разность числа 360 и остатка (то есть −(360 − остаток)). Отрицательные значения конвертируются аналогичным образом.

Следует иметь в виду, что поворот объекта выполняется относительно точки его регистрации. Ниже приводится фрагмент кода, заставляющий клип mc непрерывно вращаться по часовой стрелке.

```
mc.onEnterFrame = function() {
        mc._rotation += 10;
};
```

Если к экземпляру в рабочей среде применяется трансформация поворота, то это оказывает влияние на значение его свойства _rotation. Так, если экземпляр в рабочей среде повернут на 45° по часовой стрелке, то значение его свойства _rotation будет составлять 45.

Прозрачность. Свойство *_alpha*

Данное свойство позволяет контролировать степень прозрачности клипа (кнопки или текстового блока). Свойство _alpha используется как для чтения, так и для записи, и имеет значение типа Number, диапазон его значений лежит в границах от 0 до 100. Значение 100 соответствует полной непрозрачности объекта. Значение 0 делает объект полностью прозрачным. Промежуточные значения определяют частичную прозрачность.

Установки прозрачности, выполненные с помощью цветового эффекта экземпляров **alpha** в рабочей среде, оказывают влияние на значение свойства _alpha. Следует отметить, что вследствие особенностей реализации механизма вычисления данного свойства его значения могут представлять собой дробное число. При использовании свойства _alpha в составе выражений это может привести к накоплению погрешности и, как следствие, к непредвиденным результатам.

С помощью свойства _alpha можно реализовать множество интересных эффектов, связанных с изменением степени прозрачности объекта. Так, например, можно имитировать эффект фокусирования, при котором нерезкое изображение постепенно приобретает деталировку. Для этого потребуется два одинаковых растровых изо-

бражения, к одному из которых применен эффект размытия (для этих целей можно, например, воспользоваться фильтром Gaussian Blur программы Photoshop). Размытое изображение необходимо конвертировать в клип. Клип с размытым изображением следует поместить поверх резкого растрового изображения. После этого остается только реализовать такое поведение, что при наведении на клип с нерезким изображением он начинает постепенно приобретать прозрачность, пока не станет полностью прозрачным, и зритель не увидит расположенное под ним четкое изображение. При уходе курсора с клипа нерезкое изображение начинает постепенно становиться непрозрачным, пока полностью не перекроет расположенный под ним резкий экземпляр. В листинге 12.4 представлен код, позволяющий реализовать такое поведение. Здесь `picBlur` — это клип, содержащий нерезкое изображение.

Листинг 12.4. Анимация прозрачности

```
picBlur.onRollOver = function() {
       this.onEnterFrame = function() {
              this._alpha -= 5;
              if (this._alpha <= 0) {
                     delete this.onEnterFrame;
              }
       };
};
picBlur.onRollOut = function() {
       this.onEnterFrame = function() {
              this._alpha += 5;
              if (this._alpha >= 100) {
                     delete this.onEnterFrame;
              }
       };
};
```

Имя. Свойство *_name*

Свойство `_name` хранит имя экземпляра клипа (кнопки или текстового блока) в виде строки. Его можно использовать как для чтения, так и для записи. Присваивая свойству `_name` строковое значение, можно переименовать объект.

Свойство `_name` часто применяется с целью идентификации конкретного объекта, в названии которого содержится числительное. Так, например, если на сцене расположено пятьдесят экземпляров клипов и с каждым из них необходимо выполнить определенный набор операций, для того чтобы автоматизировать эту процедуру, можно при создании клипов снабдить их имена порядковыми числительными, на-

пример, mc0, mc0, ..., mc49. В этом случае для идентификации клипа всегда можно выделить из его имени числительное, например, следующим образом:

```
this._name.substr(индекс)
```

Здесь *индекс* — это индекс вхождения числительного в строку имени объекта. Вот как будет выглядеть данное предложение для клипа mc35:

```
this._name.substr(2) // Возвращает строку "35"
```

Такой подход позволяет обрабатывать множество объектов в цикле, обрабатывая каждый определенным образом, исходя из порядкового номера. Практический пример, иллюстрирующий подобное применение свойства _name, приведен в листинге 12.5.

Отображение. Свойство _visible

Данное свойство позволяет отключить отображение клипа (кнопки или текстового блока). Значением свойства _visible может быть булево значение true или false (при использовании числовых значений они автоматически преобразуются в булев тип). Если свойство _visible объекта установлено в false, то объект перестает быть виден, однако он не удаляется со сцены и, соответственно, все его переменные и функции продолжают существовать. Если на монтажной линейке такого клипа имеется анимация, то ею можно управлять так же, как и анимацией видимого клипа (за исключением того, что содержимое клипа не видно).

Во многих случаях вместо отключения видимости клипа можно просто сделать его полностью прозрачным с помощью свойства _alpha. Это приведет к тому, что клип, фактически оставаясь на сцене, становится невидимым для зрителя. Тем не менее у свойства _visible имеется одна характерная особенность. Она заключается в том, что клип, свойство _visible которого имеет значение false, перестает выполнять функции кнопки, даже если его свойство enabled установлено в true. Кроме того, следует иметь в виду, что обработка свойства _alpha требует больших ресурсов процессора, чем обработка свойства _visible, поэтому, по возможности, следует отдавать приоритет последнему.

В качестве примера использования свойства _visible рассмотрим довольно часто встречающуюся на практике ситуацию. Имеется интерфейс, состоящий из ряда кнопок и области окна. При наведении курсора на каждую кнопку в области окна должна появляться картинка, соответствующая данной кнопке. При уходе курсора с кнопки картинка в области окна исчезает. Один из способов реализации этой задачи заключается в создании ряда клипов, содержащих картинки, соответствующие кнопкам (скорее всего, клипы будут иметь одинаковый размер, чтобы соответствовать размерам окна). В рабочей среде все клипы располагаются в окне поверх друг друга в произвольной последовательности. В результате выполнения сценария видимость клипов в окне отключается, и конкретный клип отображается в окне только в том случае, если курсор наводится на соответствующую ему кнопку. При уходе курсора с кнопки отображение клипа снова отключается. В листинге 12.5 приведен сценарий, реализующий подобную функциональность. Экземпляры клипов имеют имена pic1, pic2, ..., pic5, а экземпляры кнопок названы, соответственно, but1, but2, ..., but5.

В примере, находящемся на сопровождающем книгу компакт-диске, имеется пять цветных кнопок, при наведении на каждую из которых в окне появляется цвет, соответствующий наведенной кнопке. Аналогичным образом можно использовать векторные или растровые рисунки, текст, а также анимированные клипы.

Листинг 12.5. Реализация интерфейса со сменой изображений при наведении на кнопки

```
var num:Number = 5; // Количество кнопок/картинок
for (i = 0; i <= num; i++) {
// Отключаем видимость всех картинок
        this["pic" + i]._visible = false;
/* при наведении на кнопку включаем видимость картинки, имя которой
оканчивается на ту же цифру, что и имя кнопки, на которую наведен курсор
мыши */
        this["but" + i].onRollOver = function() {
        this._parent["pic" + this._name.substr(-1)]._visible = true;
        };
// Аналогично отключаем видимость при уходе с кнопки...
        this["but" + i].onRollOut = function() {
        this._parent["pic" + this._name.substr(-1)]._visible = false;
        };
// Выполняется при уходе с кнопки с нажатой кнопкой мыши...
        this["but" + i].onDragOut = this["but" + i].onRollOut;
}
```

Координаты мыши. Свойства *_xmouse, _ymouse*

Свойства `_xmouse` и `_ymouse` содержат текущие координаты курсора мыши по горизонтали и по вертикали соответственно. Значения данных свойств имеют тип `Number`, и свойства могут быть использованы только для чтения (то есть им нельзя присвоить значение). Координаты мыши, которые хранятся в свойствах `_xmouse` и `_ymouse`, регистрируются в системе координат объекта, осуществляющего обращение к этим свойствам. Данное обстоятельство необходимо принимать в расчет, поскольку основная монтажная линейка (`_root`) и каждый экземпляр клипа обладают своей собственной системой координат. Соответственно значения свойств `_xmouse` и `_ymouse` могут различаться в зависимости от того, в системе координат какого объекта они получены. Так, для того чтобы получить значения координат курсора мыши, измеренные в системе координат клипа `mc`, необходимо использовать следующую запись (предполагается, что сценарий расположен в кадре, содержащем клип `mc`):

```
var x:Number = mc._xmouse;
var y:Number = mc._ymouse;
```

Если требуется получить координаты курсора мыши в системе координат объекта, на монтажной линейке которого находится клип mc, синтаксис обращения к свойствам _xmouse и _ymouse будет следующим (предполагается, что сценарий расположен в кадре, содержащем клип mc):

```
var x:Number = this._xmouse;
var y:Number = this._ymouse;
```

Если координаты мыши необходимо получить в системе координат объекта, на монтажной линейке которого находится клип mc, а сценарий находится на монтажной линейке mc, то синтаксис обращения к свойствам будет следующим:

```
var x:Number = this._parent._xmouse;
var y:Number = this._parent._ymouse;
```

Для получения координат мыши в системе координат основной монтажной линейки свойства _xmouse и _ymouse могут быть вызваны непосредственно с помощью _root:

```
var x:Number = _root._xmouse;
var y:Number = _root._ymouse;
```

Примечание

Отметим, что при выходе курсора мыши за пределы окна проигрывателя свойства _xmouse и _ymouse хранят значение координат точки, в которой курсор был зарегистрирован в последний раз перед выходом за пределы окна. Если курсор сразу находился за его пределами, то значение этих свойств устанавливается в 0. Тот факт, что значения свойств, хранящих координаты мыши, не отражают выход курсора за пределы окна проигрывателя, не позволяет создать совершенный механизм, отслеживающий это событие. При этом, правда, следует иметь в виду, что если курсор мыши покинул область окна проигрывателя с нажатой левой кнопкой мыши, то его координаты продолжают фиксироваться в свойствах _xmouse и _ymouse до тех пор, пока фильм удерживает фокус, или до тех пор, пока кнопка мыши не будет отпущена. Так, например, при просмотре фильма в браузере при уходе курсора с нажатой левой кнопкой мыши за пределы окна проигрывателя координаты курсора будут продолжать отслеживаться только в том случае, если кнопка мыши была нажата на кнопке или внутри выделяемого текстового блока.

Часто на практике свойства _xmouse и _ymouse применяются в ситуациях, когда состояние определенных элементов фильма зависит от местоположения курсора. Один из таких примеров пользовательский курсор уже был представлен в листинге 11.1. Теперь рассмотрим еще один полезный пример, позволяющий реализовать функциональность типа *Drag and Drop*. Задача заключается в том, что при нажатии на клип он должен следовать за курсором (*Drag*) до тех пор, пока левая кнопка мыши не будет отпущена. При отпускании кнопки мыши объект остается на том месте, куда он был помещен (*Drop*). Для того чтобы подобную процедуру можно было проделать с любым количеством объектов, создадим функцию под названием dragAndDrop(), принимающую в качестве параметра ссылку на клип, который при нажатии требуется перетаскивать. При нажатии на клип курсор мыши, конечно же, не всегда будет находиться над его точкой регистрации. В связи с этим необходимо определить, должен ли клип при перетаскивании позиционироваться так, чтобы местопо-

ложение курсора мыши совпадало с местоположением его точки регистрации, или же курсор мыши будет расположен (по отношению к клипу) в той точке, где он находился в момент нажатия на клип. Для указания требуемого режима введем параметр `lockCenter`, принимающий булево значение `true` или `false`. Если параметр `lockCenter` принимает значение `true`, то при перетаскивании курсор мыши находится в точке регистрации перетаскиваемого клипа, то есть в его начале координат. Если параметр `lockCenter` принимает значение `false`, то при перетаскивании курсор по отношению к клипу находится в той же точке, где он находился в момент нажатия на него. Если при вызове функции `dragAndDrop()` этот параметр не указан, его значение принимается равным `false`.

Функция `dragAndDrop()` приведена в листинге 12.6.

Листинг 12.6. Реализация функциональности типа *Drag and Drop*

```
function dragAndDrop(clip:MovieClip, lockCenter:Boolean):Void {
clip.onPress = function() { // При нажатии...
        // Делаем клип, по которому щелкнули курсором самым верхним в
        стопке
        this.swapDepths(this._parent.getNextHighestDepth());
        // Сохраняем расстояние от курсора в момент нажатия до точки
        регистрации клипа
        var xDist = this._x - this._parent._xmouse;
        var yDist = this._y - this._parent._ymouse;
        this.onMouseMove = function() { // При перемещении курсора...
        // Перетаскивание
                if(lockCenter){/* Если при перетаскивании курсор должен
                находиться в точке регистрации клипа... */
                this._x = this._parent._xmouse;
                this._y = this._parent._ymouse;
                }
                else { /* Если при перетаскивании курсор должен находиться
                в точке, где он был в момент нажатия на клип или, если
                параметр lockCenter не указан...*/
                this._x = this._parent._xmouse+xDist;
                this._y = this._parent._ymouse+yDist;
                }
                updateAfterEvent(); //Обновление экрана
        };
};
clip.onRelease = function() { // При отпускании кнопки мыши...
        delete this.onMouseMove; // Прекращаем перетаскивание
};
}
```

Для того чтобы при нажатии левой кнопки мыши в момент нахождения курсора над клипом данный клип оказывался над всеми остальными клипами содержащей его монтажной линейки, в теле функции используется следующее предложение:

```
this.swapDepths(this._parent.getNextHighestDepth());
```

Подробное описание применяемых здесь методов `swapDepths()` и `getNextHighestDepth()` будет приведено далее в настоящей главе.

Применим нашу функцию `dragAndDrop()` для перетаскивания за точку регистрации любого из семи клипов с именами `mc1`, `mc2`, ..., `mc7`.

```
var clipNum:Number = 7 // Число клипов
for(i = 1; i <= clipNum; i++){
dragAndDrop(this["mc"+i], true);
}
```

Обратите внимание, поскольку при вызове функции `dragAndDrop()` в качестве параметра ей на каждой итерации цикла необходимо передать новое имя клипа, для этого используется оператор `[]`. Он позволяет сгенерировать требуемое имя путем конкатенации строки `"mc"`, представляющей собой постоянную часть имени, и строкового представления счетчика цикла, принимающего уникальное значение на каждой итерации.

Кадры. Свойства *_totalframes*, *_currentframe* и *_framesloaded*

Данные свойства позволяют получить информацию о количестве кадров монтажной линейки, о номере текущего кадра и о количестве загруженных кадров. К любому из этих свойств можно обратиться как из экземпляра клипа, так и с помощью идентификатора `_root`.

Свойство *_totalframes*

Свойство `_totalframes` содержит значение, определяющее количество кадров на монтажной линейке объекта, выполняющего обращение к данному свойству. Свойство `_totalframes` принимает целочисленные значения и используется только для чтения.

Для того чтобы определить количество кадров на монтажной линейке клипа `mc`, используется следующая запись:

```
mc._totalframes
```

Для того чтобы определить количество кадров основной монтажной линейки, используется следующий синтаксис:

```
_root._totalframes
```

Свойство _currentframe

Свойство _currentframe содержит номер текущего кадра монтажной линейки объекта, выполняющего обращение к данному свойству. Как и предыдущее, данное свойство принимает только целочисленные значения типа Number и используется только для чтения. Диапазон принимаемых им значений лежит в границах от 1 до _totalframes.

Свойство _framesloaded

Данное свойство содержит количество загруженных кадров основной монтажной линейки или клипа. Свойство_framesloaded принимает целочисленные значения и используется только для чтения. Диапазон принимаемых им значений лежит в границах от 0 до _totalframes.

Следует отметить, что при вызове данного свойства из клипа оно всегда возвращает значение, соответствующее значению свойства _totalframes для этого клипа. Это происходит из-за того, что клипы полностью загружаются до начала своего воспроизведения.

Свойство _framesloaded можно использовать для создания предварительного загрузчика (preloader), который не позволяет начать воспроизведение фильма до тех пор, пока все его кадры не будут загружены. Очевидно, что для контроля состояния загрузки основного фильма свойство вызывается непосредственно для _root.

Примечание

Необходимо отметить, что если фильм состоит из небольшого числа кадров, то информация о состоянии загрузки кадров может быть неточна. Кроме того, если скорость соединения достаточно высока, то за время экспонирования одного кадра может произойти загрузка сразу нескольких следующих за ним кадров. В связи с этим при использовании свойства _framesloaded в методе обработчика onEnterFrame изменение его значения будет происходить с частотой меньшей, чем скорость загрузки кадров.

В качестве примера создадим предварительный загрузчик, отслеживающий загрузку фильма по кадрам. По мере загрузки очередного кадра наш предзагрузчик будет выводить соответствующую информацию в панель **Output**. Воспроизведение фильма будет запущено только после того, как все его кадры будут загружены (то есть после того, как значение _root._framesloaded станет равно значению _root._totalframes). Единственная трудность в создании такого предзагрузчика заключается в том, что сообщение о загрузке очередного кадра должно выводиться только один раз вне зависимости от того, в течение какого времени он будет загружаться. Сценарий, реализующий поставленную задачу, представлен в листинге 12.7.

Листинг 12.7. Покадровый предварительный загрузчик

```
/* Данный сценарий должен быть помещен в первый кадр основной монтажной
линейки */
this.stop(); // Останавливаем анимацию основной монтажной линейки
_root.onEnterFrame = function() {
```

```
        if (lastFrameLoaded == undefined) {
        lastFrameLoaded = 0; // Инициализация переменной lastFrameLoaded
        }
// Если загружены еще не все кадры фильма...
        if (this._framesloaded != this._totalframes) {
                // Если загрузился очередной новый кадр...
                if (lastFrameLoaded - this._framesloaded < 0) {
                trace("Загрузился " + this._framesloaded + "-й кадр.");
                // Обновляем значение lastFrameLoaded
                lastFrameLoaded = this._framesloaded;
                }
        }
// Если загрузка фильма окончена ...
        else {
        trace("Загрузился последний " + this._framesloaded + "-й кадр.");
        this.play(); // Запускаем анимацию основной монтажной линейки
        delete lastFrameLoaded; /* Удаляем переменную lastFrameLoaded,
        так как она больше не нужнва */
        delete _root.onEnterFrame; /* Удаляем метод обработчика событий
        для прекращения работы предзагрузчика */
        }
};
```

Примечание

Для оценки работы предзагрузчика необходимо в среде тестирования выполнить команду главного меню **View>Simulate Download**.

Дадим некоторые пояснения к приведенному сценарию. Для того чтобы сообщение о загрузке очередного кадра выводилось только один раз, мы создаем переменную `lastFrameLoaded`, в которой будем сохранять номер последнего загруженного кадра, полученный с помощью свойства `_framesloaded`. В связи с тем, что при каждом выполнении функции обработчика значение переменной должно соответствовать номеру последнего загруженного кадра, мы не можем объявить ее как локальную переменную, поскольку в этом случае при каждом выполнении функции переменная будет инициализироваться вновь, и ее значение не будет постепенно увеличиваться по мере того, как увеличивается количество загруженных кадров. Поэтому переменная объявляется неявно (как нелокальная) и инициализируется только единожды при самом первом выполнении метода обработчика.

Вывод сообщения о загрузке нового кадра осуществляется только в том случае, если выполняется условие `lastFrameLoaded - this._framesloaded < 0`. Данное условие выполняется, если значение свойства `_framesloaded` оказывается больше значения переменной `lastFrameLoaded`, то есть в момент окончания загрузки очередного кадра.

После вывода сообщения о загрузке кадра переменной `lastFrameLoaded` присваивается значение свойства `_framesloaded`, таким образом, она хранит номер последнего загруженного кадра, который при следующем вызове метода обработчика сравнивается со значением свойства `_framesloaded` для проверки того, загрузился очередной кадр или нет.

Поскольку переменная `lastFrameLoaded` не является локальной, после того как все кадры фильма загружены, ее можно удалить, чтобы освободить место в памяти.

После окончания загрузки удаляем метод обработчика событий `onEnterFrame()` для того, чтобы прекратить периодическое выполнение сценария предварительного загрузчика.

Свойства клипов-кнопок: *useHandCursor*, *enabled*, *trackAsMenu*, *hitArea*

Как было показано в *главе 11*, клипы могут обладать всей функциональностью кнопок. Для того чтобы обеспечить большую гибкость в их управлении класс `MovieClip` включает ряд встроенных свойств, описание которых представлено в этом разделе.

Свойство *useHandCursor*

Данное свойство определяет, как будет выглядеть курсор при наведении на клип-кнопку (или на экземпляр кнопки). Свойство `useHandCursor` принимает булевы значения `true` и `false` и может использоваться как для чтения, так и для записи. Если свойство `useHandCursor` клипа-кнопки установлено в значение `true`, то курсор мыши при наведении на такой клип будет принимать вид указателя в виде руки. Этот режим по умолчанию используется обычными кнопками и клипами-кнопками. Если же значение свойства `useHandCursor` будет установлено в `false`, то при наведении на клип (или кнопку) курсор будет иметь вид стандартного указателя в виде стрелки (то есть его внешний вид не изменится). Рисунок 12.3 иллюстрирует результат действия данного свойства. Значение данного свойства может быть переопределено в нужный момент, при этом выполненные изменения моментально вступят в силу.

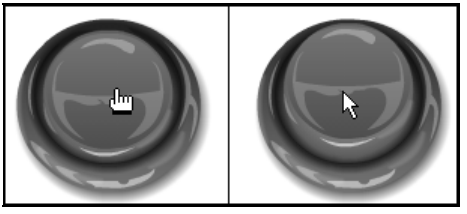


Рис. 12.3. Управление внешним видом курсора с помощью свойства `useHandCursor`

Выполнение следующего предложения приведет к тому, что при наведении на клип-кнопку с именем `mc` курсор мыши не изменит своего внешнего вида:

```
mc.useHandCursor = false;
```

Свойство *enabled*

Данное свойство используется для активизации или отключения работы клипа-кнопки или обычной кнопки. Свойство `enabled` может принимать булевы значения `true` или `false` и может использоваться как для чтения, так и для записи. Данное свойство оказывает влияние на работу клипа только в том случае, если он используется в качестве кнопки, то есть если он использует обработчик `on()` или любой из методов обработчиков событий, свойственных кнопке.

Если значение свойства `enabled` установлено в `false`, то клип-кнопка (или обычная кнопка) перестает реагировать на воздействие курсора, состояния `up`, `over` и `down` не отрабатываются. Кроме того, прекращаются вызовы методов обработчиков событий данного клипа, связанных с его функциональностью в качестве кнопки (`onPress()`, `onRelease()`, `onRollOver()` и т. д.). Выполнение блока кода обработчика `on()`, прикрепленного к клипу, также прекращается. Тем не менее продолжают выполняться вызовы методов обработчиков событий клипа (`onEnterFrame()`, `onMouseMove()`, и т. д.) и обработчик `onClipEvent()` также продолжает работу. Если на монтажной линейке клипа имеется анимация, то она будет выполняться вне зависимости от значения свойства `enabled`. Таким образом, данное свойство управляет только функциональностью клипа, связанной с его выполнением роли кнопки.

По умолчанию свойство `enabled` принимает значение `true`. Значение данного свойства может быть переопределено в нужный момент, при этом выполненные изменения моментально вступят в силу.

В следующем предложении выполняется отключение возможности выполнения роли кнопки для клипа с именем `mc`.

```
mc.enabled = false;
```

Свойство *hitArea*

Данное свойство служит для обеспечения возможности применения одного экземпляра клипа в качестве области реагирования (активной области) другого экземпляра клипа-кнопки. В качестве значения свойства `hitArea` используется ссылка на клип, выполняющий роль области реагирования.

Описание применения данного свойство было приведено *в разд. "Создание клипа-кнопки" главы 11.*

Свойство *trackAsMenu*

Свойство `trackAsMenu` применяется для определения типа поведения клипа-кнопки (или обычной кнопки). Данное свойство может принимать булевы значения `true` или `false` и используется как для чтения, так и для записи. Установка свойству `trackAsMenu` клипа (или кнопки) значения `true` активизирует для данного экземпляра режим **Track as Menu Item**, позволяющий использовать его в качестве элемента меню. В результате активации данного режима имеют место следующие особенности использования клипов-кнопок.

- Если в момент наведения курсора мыши на клип левая кнопка мыши оказывается нажатой, то при отпускании кнопки мыши над активной областью клипа генерируется событие `release`.

- Если курсор мыши выходит из активной области клипа и при этом левая кнопка мыши остается нажатой, то при отпускании кнопки мыши вне активной области клипа событие `releaseOutside` не генерируется.

В табл. 10.2 приводится подробное описание различий в обработке событий кнопки в зависимости от типа ее поведения. Все описанные в данной таблице правила также относятся и к клипам-кнопкам.

Основной особенностью режима **Track as Menu Item** является возможность обработки событий клипа-кнопки (или кнопки), вхождение курсора в активную область которого происходит при нажатой кнопке мыши. Эта особенность может быть полезной при создании выпадающих списков и меню, предполагающих перенос курсора мыши с одной опции на другую с нажатой кнопкой мыши. Применение данного подхода демонстрируется в листинге 12.8.

Если значение свойства `trackAsMenu` установлено в `false`, то клип-кнопка функционирует в режиме **Track as Button**, используемом по умолчанию.

Значение данного свойства может быть переопределено в нужный момент, при этом выполненные изменения моментально вступят в силу.

Создание выпадающего списка

В качестве примера, иллюстрирующего применение многих описанных выше свойств, рассмотрим процесс создания выпадающего списка, представляющего собой довольно часто используемый элемент интерфейса.

Для реализации выпадающего списка может быть использовано несколько различных подходов в зависимости от требуемой функциональности и уровня интерактивности. Рассмотрим создание выпадающего списка, обладающего следующим поведением:

- список раскрывается при нажатии левой кнопки мыши на его заголовке;
- доступ к элементам списка может быть получен при переносе на них курсора с заголовка как с нажатой кнопкой мыши, так и после ее отпускания на заголовке;
- при наведении курсора на элемент списка он должен быть подсвечен;
- выбор элемента списка приводит к выполнению определенного действия (в нашем случае вывод в панель **Output** имени клипа, представляющего выбранный элемент) и отображению названия выбранного элемента в заголовке списка;
- выбор элемента осуществляется при отпускании кнопки мыши в момент нахождения курсора на элементе;
- элементы списка скрываются при отпускании кнопки мыши на элементе или при щелчке кнопкой мыши в любой точке экрана вне списка.

Внешний вид выпадающего списка представлен на рис. 12.4 (подписи элементов), исходный документ fla читатель сможет найти на сопровождающем книгу компакт-диске в файле 12_8.fla, расположенном в папке Глава 12.

В основе выпадающего списка лежат два символа типа муви-клип.

- Символ, представляющий заголовок списка. Данный символ будет использован в качестве клипа-кнопки. На монтажной линейке этого символа кроме графики

должен находиться динамический (Dynamic Text) текстовый блок с именем (Instance Name) `txt`.

- Символ, представляющий элемент списка. Данный символ будет использован в качестве клипа-кнопки. Для того чтобы при наведении элемент списка подсвечивался, на монтажной линейке данного клипа создадим три кадра с метками `_up`, `_over`, `_down` и разместим в них соответствующее графическое наполнение. На монтажной линейке этого символа кроме графики должен находиться динамический (Dynamic Text) текстовый блок с именем (Instance Name) `txt`. Его нужно поместить на отдельный слой монтажной линейки символа элемента списка. В этот текстовый блок будет выводиться название опции, входящей в состав списка.

На сцене необходимо разместить один экземпляр символа, представляющего заголовок списка, и присвоить ему имя `shell`. Затем необходимо разместить столько экземпляров символа, представляющего элемент списка, сколько элементов будет отображаться в списке. Экземплярам данного символа присвоим имена `option0`, `option1`, ..., `option(N-1)`, где `N` — это количество элементов в списке (рис. 12.4).

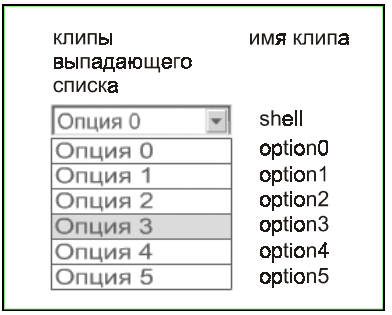


Рис. 12.4. Элементы выпадающего списка

Примечание

Несмотря на то, что используемые символы содержат текстовые блоки, имеющие одинаковое имя, конфликтов имен происходить не будет, поскольку блоки, будучи расположены на монтажных линейках символов, существуют в разных пространствах имен. Текстовый блок в каждом экземпляре находится в пределах его локальной области видимости (как если бы это была обычная переменная, объявленная на монтажной линейке символа). Применение одного и того же имени текстовых блоков позволяет реализовать все элементы списка как экземпляры одного и того же символа. Это существенно упрощает процесс создания списка.

Сценарий, обеспечивающий функционирование списка, расположен в кадре фильма, содержащем его заголовок и элементы. Рассмотрим процесс разработки этого сценария. Исходный код представлен в листинге 12.8.

Инициализация

Введем переменную optionNum, хранящую количество элементов в списке:

```
var optionNum:Number = 6;
```

Для того чтобы обеспечить возможность после нажатия кнопки мыши на заголовке списка, не отпуская ее, перевести курсор на элементы списка для выбора нужной опции, включаем для клипа-заголовка режим **Track as Menu Item**:

```
shell.trackAsMenu = true;
```

В цикле выполним общие для всех клипов-элементов установки:

```
for (i = 0; i < optionNum; i++) {
```

Поскольку в качестве элементов используются клипы-кнопки, для того чтобы исключить самопроизвольную анимацию их монтажных линеек (включающих три кадра), в цикле останавливаем их воспроизведение:

```
        this["option" + i].stop();
```

Поскольку клипы-элементы списка мы размещаем на сцене непосредственно в рабочей среде, необходимо, чтобы в начальный момент они были скрыты, поэтому устанавливаем свойство _visible для каждого их них в false:

```
        this["option" + i]._visible = false;
```

Для того чтобы обеспечить возможность перехода с одного элемента на другой при нажатой кнопке мыши, включаем режим **Track as Menu Item** для всех клипов-элементов:

```
        this["option" + i].trackAsMenu = true;
```

Выводим в текстовые блоки клипов-элементов однотипные названия, состоящие из строки "Опция " и порядкового номера элемента:

```
        this["option" + i].txt.text = "Опция " + i;
}
```

Выводим в блок заголовка списка текст, содержащий значение, используемое по умолчанию (нулевая опция):

```
shell.txt.text = "Опция 0";
```

Функция, отключающая отображение элементов списка

Создадим функцию hideOptions(), которая будет отключать отображение элементов списка:

```
function hideOptions():Void {
        for (i = 0; i < optionNum; i++) {
                // Отключение отображения элементов списка
                this["option" + i]._visible = false;
        }
}
```

Функция, обеспечивающая отображение элементов и реализацию их функциональности

Функцию, выполняющую эти задачи, назовем showOptions().

```
function showOptions():Void {
        for (i = 0; i < optionNum; i++) {
                this["option" + i]._visible = true;
                this["option" + i].onRelease = function() {
                        trace("Выбран клип с именем " + this._name);
                        this._parent.shell.txt.text = this.txt.text;
                        hideOptions();
                };
        }
}
```

Данная функция включает отображение всех элементов меню. Затем для каждого клипа-элемента определяется метод обработчика события onRelease, применение которого приведет к тому, что при отпускании кнопки мыши над любым элементом списка будут выполняться следующие действия.

1. Вывод имени выбранного клипа-элемента в панель **Output**.
2. Вывод содержимого текстового блока клипа-элемента в текстовый блок клипа-заголовка списка.
3. Отключение отображения элементов списка.

Включение отображения элементов списка

Для того чтобы при нажатии кнопки мыши в момент нахождения курсора на заголовке включить отображение элементов, используем следующий код:

```
shell.onPress = function() {
        showOptions();
};
```

Управление элементами списка при отсутствии выбора

Поскольку возможна ситуация, когда пользователь не выполнит выбор элемента, необходимо сделать так, чтобы при развернутом списке щелчок по любой точке экрана, не принадлежащей списку, приводил к сокрытию его элементов. Кроме того, необходимо, чтобы при повторном наведении курсора извне на элементы списка можно было, щелкнув по элементу, реализовать обычную схему выбора.

Для этих целей используем обработчик события onMouseDown, контролирующий нажатие кнопки мыши. Следует отметить, что блок кода обработчика onMouseDown выполняется до блока кода обработчика onPress. Все действия, выполняемые методом данного обработчика, должны происходить только в том случае, если список развернут, то есть его клипы-элементы видны. В противном случае при отключении

видимости элементов списка они будут продолжать функционировать. В связи с этим используем условное предложение, проверяющее выполнение условия видимости элементов. Поскольку видимость всех клипов-элементов включается или отключается одновременно, достаточно проверить, включено ли отображение одного из них (например, нулевого).

```
this.onMouseDown = function() {
if(option0._visible){
```

Если список развернут, проверяем, находится ли курсор мыши в момент нажатия левой кнопки мыши над элементами списка. Для этого проверяем условие попадания координат курсора в область, занимаемую клипами-элементами. Поскольку горизонтальные и вертикальные размеры клипов-элементов совпадают, и они расположены вплотную друг над другом, а их точки регистрации находятся в левом верхнем углу, можно легко составить условие попадания курсора в занимаемую ими область (см. рис. 12.4). Горизонтальные границы этой области определяются на основании положения и ширины нулевого элемента, а вертикальные границы — на основании положения нулевого и последнего элементов, а также высоты любого элемента (в данном случае последнего). Поскольку список может содержать произвольное количество элементов, имя последнего элемента генерируется динамически на основании значения переменной `optionNum`. Заголовок условного предложения будет иметь следующий вид:

```
if (this._xmouse > option0._x && this._xmouse < (option0._x +↵
option0._width) && this._ymouse > option0._y && this._ymouse <↵
(this["option" + (optionNum-1)]._y + this["option" +↵
(optionNum - 1)]._height)) {
```

Если кнопка мыши нажата над элементом, вызываем функцию `showOptions()`, которая реализует всю функциональность элемента.

```
showOptions();
}
```

Если курсор в момент нажатия кнопки мыши находится вне области, занимаемой элементами меню, скрываем эти элементы, вызывая функцию `hideOptions()`.

```
else {
hideOptions();
}
}
};
```

Листинг 12.8. Выпадающий список

```
// ____________________ИНИЦИАЛИЗАЦИЯ____________________
var optionNum:Number = 6; // Количество опций в списке
shell.trackAsMenu = true; // Поведение клипа заголовка
for (i = 0; i < optionNum; i++) {
```

```
            /* Остановка анимации клипов, представляющих элементы
            списка */
            this["option" + i].stop();
            /* Отключение отображения элементов списка в начальный
            момент */
            this["option" + i]._visible = false;
            // Поведение элементов списка
            this["option" + i].trackAsMenu = true;
            // Названия элементов списка
            this["option" + i].txt.text = "Опция " + i;
}
shell.txt.text = "Опция 0"; // Текст в заголовке списка по умолчанию
//_______ФУНКЦИЯ, ОТКЛЮЧАЮЩАЯ ОТОБРАЖЕНИЕ ЭЛЕМЕНТОВ СПИСКА_____________
function hideOptions():Void {
      for (i = 0; i < optionNum; i++) {
            // Отключение отображения элементов списка
            this["option" + i]._visible = false;
      }
}
/* ФУНКЦИЯ, ВКЛЮЧАЮЩАЯ ОТОБРАЖЕНИЕ ЭЛЕМЕНТОВ СПИСКА И ОПРЕДЕЛЯЮЩАЯ
ДЕЙСТВИЯ, ВЫПОЛНЯЕМЫЕ ПРИ ОТПУСКАНИИ КНОПКИ МЫШИ НА ЭЛЕМЕНТЕ СПИСКА */
function showOptions():Void {
      for (i = 0; i < optionNum; i++) {
            this["option" + i]._visible = true;
      /* Определение действий, выполняемых при отпускании кнопки мыши
      над элементом списка */
            this["option" + i].onRelease = function() {
                  trace("Выбран клип с именем " + this._name);
                  // Вывод выбранной опции в заголовок списка
                  this._parent.shell.txt.text = this.txt.text;
                  // Отключение отображения элементов списка
                  hideOptions();
            };
      }
}
// ВКЛЮЧЕНИЕ ОТОБРАЖЕНИЯ ЭЛЕМЕНТОВ СПИСКА ПРИ НАЖАТИИ НА КЛИП ЗАГОЛОВКА
shell.onPress = function() {
      showOptions();
};
```

```
/* ОТКЛЮЧЕНИЕ ОТОБРАЖЕНИЯ ЭЛЕМЕНТОВ СПИСКА ПРИ НАЖАТИИ КНОПКИ МЫШИ ВНЕ
ОБЛАСТИ, ЗАНИМАЕМОЙ ЕГО ЭЛЕМЕНТАМИ. ОБЕСПЕЧЕНИЕ ФУНКЦИОНАЛЬНОСТИ ПРИ
ЩЕЛЧКЕ ПО ЭЛЕМЕНТУ СПИСКА ПОСЛЕ УХОДА КУРСОРА ЗА ПРЕДЕЛЫ СПИСКА И
ПОВТОРНОГО НАВЕДЕНИЯ НА ЭЛЕМЕНТ СПИСКА */
this.onMouseDown = function() {
if(option0._visible){
        /* Проверка попадания курсора в пределы области, занимаемой
        элементами списка в момент нажатия кнопки мыши. Осуществляется
        контроль координаты левого верхнего и правого нижнего угла области
        (точка регистрации в клипах элементов списка расположена в левом
        верхнем углу */
        if (this._xmouse > option0._x && this._xmouse < (option0._x +
        option0._width) && this._ymouse > option0._y && this._ymouse <
        (this["option" + (optionNum-1)]._y + this["option" +
        (optionNum - 1)]._height)) {
        /* При нажатии кнопки мыши в момент нахождения курсора над
        элементом списка, реализуем функциональность данного элемента.
        Это необходимо в ситуации, когда курсор был вначале уведен за
        пределы списка, а потом вновь помещен над ним. Для того чтобы при
        щелчке по элементу выполнились необходимые действия, вызываем
        определяющую их функцию showOptions() */
                showOptions();
                }
         /* Если курсор в момент нажатия кнопки мыши находится вне
        области, занимаемой элементами меню, скрываем эти элементы */
        else {
        hideOptions();
        }
}
};
```

Прочие свойства

В этом разделе описываются оставшиеся свойства класса `MovieClip` различного назначения.

Свойства управления качеством отображения графики

При отображении графики на экране Flash использует технологию сглаживания (Antialiasing). В результате применения сглаживания точки, расположенные на границах форм, смешиваются с точками фона с применением линейной интерполяции, в результате чего устраняется ступенчатость границ объектов, и графика отображается более аккуратно и четко. Сглаживание может применяться не только к векторным объектам и тексту, но и к растровым изображениям, интегрированным в Flash-фильм. Сглаживание растровых изображений устраняет (размазывает) границы между пикселами.

Применение сглаживания может существенно отразиться на быстродействии. Выполнение нескольких сложных анимационных процессов с применением сглаживания (особенно с использованием растровой графики или градиентных заливок) при недостаточной мощности процессора может привести к заметному снижению скорости прорисовки кадров фильма. В связи с этим ActionScript предоставляет несколько встроенных глобальных свойств, позволяющих определить степень сглаживания, используемую в документе.

Свойство *_quality*

Данное свойство позволяет задать один из четырех возможных уровней сглаживания графики для всего SWF-документа. Свойство `_quality` является глобальным, поэтому для обращения к нему не требуется создания экземпляра. В качестве значения данного свойства можно использовать одну из четырех строковых констант:

- "`LOW`" — низкое качество прорисовки. Сглаживание полностью отключено для всех графических объектов. Использование этого режима приводит к тому, что границы векторных объектов и букв имеют зубчатые очертания. Растровая графика не сглаживается. Применение данного режима обеспечивает максимальное быстродействие;
- "`MEDIUM`" — среднее качество прорисовки. Графика сглаживается с применением сетки размером 2×2 пиксела. Растровые изображения не сглаживаются;
- "`HIGH`" — высокое качество прорисовки. Графика сглаживается с применением сетки размером 4×4 пиксела. Растровые изображения сглаживаются, только если они неподвижны (статичны). Этот режим используется по умолчанию;
- "`BEST`" — наиболее высокая степень сглаживания. Для сглаживания используется сетка размером 4×4 пиксела. Сглаживание применяется ко всем объектам. Применение данного режима может привести к существенному снижению быстродействия.

Данное свойство может быть использовано как для чтения, так и для записи. В листинге 12.9 приводится сценарий, последовательно переключающий режимы качества при каждом нажатии на кнопку с именем `qualitySwitch`.

Листинг 12.9. Управление качеством прорисовки графики

```
qualitySwitch.onPress = function() {
       if (qualityOption == undefined || qualityOption == 3) {
              qualityOption = 0;
       }
       else {
              qualityOption++;
       }
       switch (qualityOption) {
       case 0 :
```

```
                _quality = "LOW";
                break;
        case 1 :
                _quality = "MEDIUM";
                break;
        case 2 :
                _quality = "HIGH";
                break;
        case 3 :
                _quality = "BEST";
                break;
        }
        trace(_quality);
};
```

Для того чтобы каждое нажатие на кнопку приводило к изменению качества прорисовки, мы вводим числовую переменную с именем `qualityOption`. При первом нажатии кнопки данная переменная принимает значение 0, при каждом последующем нажатии ее значение увеличивается на единицу, пока не достигнет значения 3. После этого значение `qualityOption` обнуляется. В зависимости от значения переменной свойство `_quality` устанавливается в одно из допустимых строковых значений. Для этого используется конструкция переключателя `switch`. При каждом нажатии кнопки `qualitySwitch` значение свойства `_quality` выводится в панель **Output**.

Свойство *_highquality*

Данное свойство используется для выполнения той же задачи, что и свойство `_quality`, однако считается устаревшим (deprecated). Свойство `_highquality`, так же как и свойство `_quality`, позволяет задать степень сглаживания, используемого при воспроизведении фильма, однако имеет числовой тип. Данное свойство принимает одно из следующих числовых значений:

- 0 — полное отключение сглаживания;
- 1 — сглаживание включено, однако растровые изображения при наличии анимации не сглаживаются;
- 2 — сглаживание применяется ко всей графике.

В связи с тем, что данное свойство считается устаревшим, рекомендуется вместо него использовать свойство `_quality`.

Свойство *_url*

Свойство `_url` хранит строку, содержащую URL (Uniform Resource Locator), унифицированный указатель информационного ресурса, с которого был загружен клип, осуществляющий обращение к данному свойству.

Если клип загружен с сервера, то URL имеет следующий формат:

```
протокол://сервер/домен1/.../доменN/имя файла.swf
```

Если клип загружен с локального диска, то URL имеет следующий формат:

```
file:///логический диск|/каталог1/.../каталогN/имя файла.swf
```

Свойство `_url` может быть использовано только для чтения. Следующая строка выводит в панель **Output** URL, с которого был загружен текущий SWF-документ:

```
trace(_root._url);
```

Свойство *_droptarget*

Данное свойство используется при перетаскивании объектов с помощью метода (или глобальной функции) `startDrag()`. Свойство `_droptarget` возвращает строку, содержащую абсолютный путь к целевому объекту, над которым находился перетаскиваемый объект в момент прекращения перетаскивания. Данное свойство возвращает путь к целевому объекту, если ограничивающая рамка перетаскиваемого объекта пересекается с ограничивающей рамкой целевого объекта и если курсор мыши при этом находится над графическим содержимым целевого объекта. В противном случае `_droptarget` возвращает пустую строку.

Следует отметить, что путь хранится в свойстве `_droptarget` в виде строки, причем в указании пути используется устаревшая нотация с наклонной чертой (/). Синтаксис указания пути с применением наклонной черты использовался в ActionScript вплоть до выхода пятой версии Flash, где ему на смену пришел точечный синтаксис. В ActionScript 2.0 синтаксис с наклонной чертой не поддерживается. В связи с этим значение, хранимое в свойстве `_droptarget`, необходимо преобразовывать в ссылку, использующую точечный синтаксис. Для этих целей используется глобальная функция `eval()`, принимающая в качестве параметра строку и преобразующая ее в идентификатор. Если функции `eval()` передается в качестве параметра строка пути с использованием наклонной черты, то данная функция преобразует ее в ссылку с применением точечного синтаксиса.

Для пояснения сказанного рассмотрим условный пример. Пусть на основной монтажной линейке имеется два клипа с именами `dragable` и `base`. Клип перемещается вслед за курсором мыши с помощью метода `startDrag()`. Следующие строки организуют вывод абсолютного пути к клипу `base`, над которым находится клип в момент вызова его метода `stopDrag()`, то есть в момент прекращения перетаскивания:

```
trace(dragable._droptarget); // Выводит /base
trace(eval(dragable._droptarget)); // Выводит _level0.base
```

А теперь проверим различие в типе возвращаемого результата:

```
trace(typeof dragable._droptarget); // Выводит string
trace(typeof eval(dragable._droptarget)); /* Выводит movieclip, так как
функция eval() возвращает ссылку на объект, а не просто строку */
```

Примечание

Свойство `_droptarget` может быть использовано только при перетаскивании объектов с помощью `startDrag()`.

Практический пример сценария, иллюстрирующий применение свойства `_droptarget`, будет приведен в листинге 12.20.

Свойство *_target*

Свойство `_target` содержит строку, представляющую абсолютный путь к объекту, вызвавшему данное свойство. Путь указывается с помощью устаревшей нотации с использованием наклонной черты вместо точечного синтаксиса. Поскольку синтаксис с использованием наклонной черты в ActionScript 2.0 не поддерживается, значение, возвращаемое данным свойством, необходимо преобразовать в ссылку, использующую точечный синтаксис. Для этого так же, как и в случае использования свойства `_droptarget`, применяется глобальная функция `eval()`.

Данное свойство используется только для чтения.

Пусть на основной монтажной линейке находится клип `mc0`, который содержит клип `mc1`, который, в свою очередь, содержит клип `mc2`. Следующий код, будучи размещен на монтажной линейке клипа `mc2`, выводит в панель **Output** путь к этому клипу и строковое представление ссылки на него:

```
trace(this._target); // Выводит путь: /mc0/mc1/mc2
trace(eval(this._target)); // Выводит ссылку: _level0.mc0.mc1.mc2
```

Для получения строки, содержащей путь к клипу, указанный с использованием точечного синтаксиса, можно также воспользоваться глобальной функцией `targetPath()`. В качестве параметра данная функция принимает ссылку на клип (или основную монтажную линейку) и возвращает абсолютный путь к указанному объекту в виде строки. Следующее предложение, размещенное на основной монтажной линейке, выводит в панель **Output** путь к клипу `mc2`:

```
trace(targetPath(mc0.mc1.mc2)); // Выводит _level0.mc0.mc1.mc2
```

Свойство *_lockroot*

Данное свойство используется для того, чтобы определить, на какой объект будет ссылаться идентификатор `_root` в случае его применения в сценариях SWF-документа, загружаемого в другой SWF-документ. Свойство `_lockroot` может принимать булево значение `true` или `false`. Если в качестве значения `_lockroot` используется `false`, то все ссылки, осуществляемые в загружаемом документе с помощью `_root`, будут разрешаться относительно основной монтажной линейки родительского документа (то есть документа, в который осуществлялась загрузка). Этот режим используется по умолчанию. Если же значение свойства `_lockroot` установлено в `true`, то идентификатор `_root`, используемый в сценариях загружаемого клипа, будет ссылаться на его основную монтажную линейку, а не на основную линейку родительского документа.

Применение данного свойства позволяет избежать возможных конфликтов, связанных с использованием абсолютной адресации в динамически загружаемых документах. Более подробно применение свойства `_lockroot` будет описано *в разд. "Использование абсолютных ссылок во внешних документах" данной главы.*

Свойства управления порядком переноса фокуса

ActionScript предоставляет ряд возможностей, позволяющих пользователю в конечном фильме переносить фокус в окне проигрывателя с одного объекта на другой. Автоматический перенос фокуса выполняется при нажатии на клавишу <Tab>. В ряде встроенных классов (`Button`, `MovieClip`, `TextField`.) определены одинаковые свойства, указывающие, может ли фокус переноситься на конкретный экземпляр, и определяющие порядок и визуальное отображение перехода фокуса от одного объекта к другому. Вопросы, связанные с контролем порядка переноса фокуса, будут рассмотрены в *главе 14*. Здесь же ограничимся кратким перечислением свойств, доступных для любого экземпляра клипа (табл. 12.1).

Таблица 12.1. *Свойства управления порядком переноса фокуса*

Свойство	Назначение
`focusEnabled`	Определяет, может ли экземпляр клипа получать фокус при отсутствии у него обработчиков событий кнопки
`_focusrect`	Определяет, будет ли экземпляр, которому передан фокус, подсвечен желтой рамкой
`tabEnabled`	Определяет, включен ли экземпляр клипа в автоматический порядок переноса фокуса с помощью клавиши <Tab>
`tabChildren`	Определяет, включены ли вложенные экземпляры клипа в автоматический порядок переноса фокуса с помощью клавиши <Tab>
`tabIndex`	Содержит индекс, определяющий позицию клипа в используемом порядке переноса фокуса с помощью клавиши <Tab>

Свойство *menu*

Свойство `menu` применяется для сопоставления экземпляру клипа объекта `ContextMenu`, используемого для управления контекстным меню проигрывателя Flash Player. Контекстное меню, содержащее набор заданных с помощью объекта `ContextMenu` пунктов, будет выводиться при щелчке правой кнопкой мыши на клипе, определяющем данное свойство. Управление контекстным меню проигрывателя рассматривается в *главе 20*.

Адресация с помощью предложения *with*

Довольно часто приходится сталкиваться с ситуацией, требующей многократного обращения к одному и тому же объекту с целью выполнения с ним разнообразных действий или получения предоставляемой данным объектом информации. Так, на-

пример, для того чтобы сместить, масштабировать и повернуть клип, необходимо использовать пять предложений, в каждом из которых выполняется обращение к свойству одного и того же объекта:

```
mc._x = 5;
mc._y = 10;
mc._xscale = 150;
mc._yscale = 150;
mc._rotation = 45;
```

Такая запись, являясь понятной и логичной, тем не менее, может стать довольно утомительной и громоздкой, особенно, если объект, к которому выполняется обращение, находится не в непосредственной области видимости сценария, содержащего обращение к нему. В такой ситуации можно воспользоваться предложением `with()`, позволяющим единожды указать целевой объект и выполнить ряд действий в контексте данного объекта.

Предложение `with()` имеет следующий синтаксис:

```
with(объект){
блок кода;
...
}
```

Здесь *объект* — это ссылка на объект, в области видимости которого будут действовать все инструкции, используемые в *блоке кода*, заключенном в фигурные скобки.

При разрешении ссылок, используемых в блоке предложения `with()`, в первую очередь анализируется область видимости объекта, указанного в круглых скобках, затем область видимости клипа (или объекта `_root`), содержащего предложение `with()`, и, наконец, глобальная область объекта `_global`.

Приведенный выше фрагмент кода можно переписать с применением предложения `with()`:

```
with(mc){
        _x = 5;
        _y = 10;
        _xscale = 150;
        _yscale = 150;
        _rotation = 45;
}
```

Следует отметить, что при присваивании значения переменной внутри предложения `with()` необходимо, чтобы эта переменная была либо объявлена вне предложения `with()`, либо в выражении присваивания указывать полный путь к адресуемой переменной. Если в блоке предложения `with()` осуществляется присваивание переменной, не определенной в области видимости объекта, указанного в круглых скобках, то данная переменная создается автоматически и прикрепляется к монтажной линейке, содержащей предложение `with()`.

Эти особенности иллюстрируются следующим примером. Пусть на основной монтажной линейке находится клип mc. В кадр основной монтажной линейки поместим следующий код:

```
mc.v1 = "Flash"; // Переменная прикреплена к линейке клипа mc
with (mc){
trace(v1); // Выводит Flash, так как v1 определена на линейке клипа mc
v1 = "ActionScript"; // Изменяем значение переменной в клипе mc
v2 = "Macromedia"; /* Создаем новую переменную на основной монтажной
линейке, так как в области видимости предложения with() переменная v2
не обнаружена */
}
```

Следует иметь в виду, что выполнение инструкций в блоке предложения `with()` может происходить несколько медленнее, чем при использовании явной адресации. В связи с этим для обеспечения более высокого быстродействия лучше избегать использования предложения `with()`.

Управление воспроизведением

Кадры монтажной линейки по умолчанию воспроизводятся слева направо последовательно друг за другом. Также последовательно выполняется переход с одной сцены на другую согласно порядку следования сцен, заданному в панели **Scene**. Основная монтажная линейка, как и монтажные линейки анимированных муви-клипов, воспроизводятся циклически, то есть воспроизводящая головка, дойдя до последнего кадра, автоматически возвращается в первый кадр, и воспроизведение возобновляется. Очень часто для решения разнообразных практических задач возникает необходимость в управлении порядком воспроизведения кадров. Например, при создании анимационного ролика может потребоваться снабдить его контроллером, позволяющим пользователю в любой момент останавливать и запускать воспроизведение, осуществлять перемотку и т. д. ActionScript включает набор средств, позволяющих осуществлять управление порядком воспроизведения кадров и выполнять переходы между сценами. Такие действия могут быть реализованы с помощью методов класса `MovieClip` или с помощью глобальных функций, названия которых совпадают с названиями данных методов. Единственное различие в использовании методов или одноименных глобальных функций заключается в том, что методы всегда вызываются экземпляром клипа, то есть объектом, а для вызова глобальной функции создание объекта не требуется. Соответственно, метод оказывает воздействие на монтажную линейку вызвавшего его объекта, а воздействие глобальной функции определяется областью видимости сценария, в котором она вызывается.

Остановка и запуск воспроизведения

Метод *stop()*

Метод `stop()` останавливает воспроизведение монтажной линейки вызвавшего его объекта. Данный метод не принимает параметров и имеет следующий синтаксис:

```
объект.stop()
```

Здесь *объект* — это экземпляр клипа, монтажная линейка которого должна быть остановлена. Метод `stop()` может быть вызван с помощью идентификатора `_root`.

Следующая строка кода останавливает воспроизведение основной монтажной линейки фильма:

```
_root.stop();
```

При вызове `stop()` в качестве глобальной функции указания объекта не требуется.

Применение метода `stop()` или одноименной глобальной функции может потребоваться в одной из следующих ситуаций:

- в первом кадре монтажной линейки для предотвращения автоматического начала ее воспроизведения;
- в последнем кадре для предотвращения циклического воспроизведения основной монтажной линейки или линейки муви-клипа;
- в последнем кадре сцены (Scene) с целью предотвращения самопроизвольного перехода с одной сцены на другую;
- в теле метода обработчика событий объекта для остановки воспроизведения в результате выполнения определенных действий пользователем или в результате протекания процессов, связанных с ходом загрузки и воспроизведения фильма.

Следующий код при нажатии на клип-кнопку с именем `mc` останавливает воспроизведение его монтажной линейки.

```
mc.onPress = function(){
       this.stop(); // Остановка анимации монтажной линейки клипа mc
};
```

Применение глобальной функции `stop()` вместо одноименного метода клипа в данном сценарии приведет к тому, что при нажатии на клип-кнопку `mc` будет остановлена анимация не самого клипа, а монтажной линейки, на которой он расположен (то есть линейки, на которой расположен приведенный сценарий). Это связано с тем, что областью видимости функции, присваиваемой обработчику события `onPress`, является монтажная линейка, содержащая ее объявление:

```
mc.onPress = function(){
       stop(); /* Остановка анимации монтажной линейки, на которой
находится клип mc */
};
```

Метод *play()*

Метод `play()` запускает воспроизведение монтажной линейки вызвавшего его объекта. Данный метод не принимает параметров и имеет следующий синтаксис:

```
объект.play()
```

Здесь *объект* — это экземпляр клипа, монтажная линейка которого должна быть остановлена, или `_root`.

При вызове `play()` в качестве глобальной функции указания объекта не требуется.

Поскольку по умолчанию воспроизведение монтажной линейки начинается автоматически, применение метода `play()` или одноименной глобальной функции, как правило, имеет место в ситуации, требующей запуска анимации, воспроизведение которой было остановлено программным образом.

Следующий код осуществляет попеременные остановку и запуск анимации основной монтажной линейки при каждом нажатии клипа-кнопки с именем `mc`.

```
mc.onPress = function() {
        if (i) {
                _root.play();
                i = false;
        }
        else {
                _root.stop();
                i = true;
        }
};
```

Переход между кадрами

В отсутствие сценариев все кадры монтажной линейки воспроизводятся последовательно друг за другом в соответствии со своей нумерацией, то есть *линейно*. Такое поведение, однако, может быть легко изменено с помощью встроенных программных средств, позволяющих произвольно управлять положением воспроизводящей головки, реализуя *нелинейное* воспроизведение. Используя встроенные возможности управления воспроизведением, можно осуществлять переход между кадрами и даже, как будет показано в конце данного раздела, заставлять воспроизводящую головку перемещаться в обратном направлении, от последнего кадра к первому.

Функция и метод *gotoAndStop()*

Как и в случае управления воспроизведением, большинство действий, связанных с переходом между кадрами, может быть реализовано с помощью глобальных функций или одноименных методов класса `MovieClip`. Воздействие глобальной функции распространяется на монтажную линейку, содержащую вызов этой функции, в то время как при использовании методов их действие распространяется на монтажную линейку объекта, вызвавшего метод.

Глобальная функция *gotoAndStop()*

Глобальная функция `gotoAndStop()` осуществляет перевод воспроизводящей головки в кадр, заданный параметрами данной функции, с последующей остановкой воспроизведения на этом кадре.

Глобальная функция `gotoAndStop()` имеет следующий синтаксис:

`gotoAndStop(`*сцена*`, `*кадр*`)`

Параметр *сцена* — это строковый литерал, представляющий собой имя сцены (Scene), к кадру которой осуществляется переход. Если документ включает несколько сцен, то с помощью данной функции можно легко переходить с кадра одной сцены в кадр другой, используя имена сцен. Следует отметить, что в качестве этого параметра может быть использован только литерал строки, то есть набор символов, заключенных в кавычки. Применение идентификатора переменной, содержащей строковое значение, вместо строкового литерала приведет к возникновению ошибки. Данный параметр является необязательным и может быть опущен. В этом случае в результате выполнения функции произойдет переход к кадру текущей сцены, то есть сцены, содержащей вызов данной функции. Следует отметить, что переход к требуемой сцене может быть выполнен, только если функция `gotoAndStop()` вызывается с основной монтажной линейки (`_root`).

Параметр *кадр* указывает кадр, в который требуется переместить воспроизводящую головку. Кадр может быть указан с помощью своего порядкового номера или с помощью метки (frame label).

Если обращение к кадру выполняется по номеру, то параметр *кадр* задается целым десятичным числом. Следует отметить, что в отличие от параметра *сцена* параметр *кадр* может быть задан не только числовым литералом, но и идентификатором переменной, содержащей числовое значение. Напомним, что нумерация кадров начинается с единицы, а не с нуля. Если значение параметра *кадр*, передаваемое функции `gotoAndStop()`, оказывается меньше единицы или больше количества кадров, расположенных на монтажной линейке, то оно автоматически устанавливается равным 1 или `_totalframes` соответственно. Если значение параметра оказывается дробным, то в расчет принимается только целая часть числа.

Если при указании кадра требуется использование его метки, то параметр *кадр* задается как строковый литерал или с помощью идентификатора, содержащего строку. Напомним, что для создания метки кадра необходимо выделить ключевой кадр на монтажной линейке и задать метку в поле **Frame Label** на панели **Properties**. В списке **Label type**, расположенном под полем **Frame Label**, необходимо выбрать тип метки: **Name**.

Примечание

Преимущество использования меток перед использованием номеров при адресации кадров состоит в том, что при перемещении кадра на монтажной линейке его порядковый номер изменяется, а метка перемещается вместе с кадром. В связи с этим, если после создания сценариев, управляющих перемещением воспроизводящей головки, возникает необходимость в перемещении кадров (например, для того чтобы продлить длительность трансформации), то применение ссылок с использованием меток не потребует изменения программного кода.

Следующая строка кода отправляет воспроизводящую головку в третий кадр сцены с названием `Intro` и останавливает воспроизведение в этом кадре.

```
gotoAndStop("Intro", 3);
```

Метод *gotoAndStop()*

Для выполнения перехода к заданному кадру с последующей остановкой в нем наряду с глобальной функцией `gotoAndStop()` можно воспользоваться методом `gotoAndStop()`, определенным в классе `MovieClip`. Данный метод имеет следующий синтаксис:

```
объект.gotoAndStop(кадр)
```

Здесь *объект* — это объект, на монтажной линейке которого осуществляются перевод воспроизводящей головки и остановка воспроизведения.

Параметр *кадр* полностью аналогичен соответствующему параметру глобальной функции `gotoAndStop()`, описанному ранее.

Следует отметить, что при вызове метода `gotoAndStop()` отсутствует возможность указания сцены, к кадру которой требуется выполнить переход.

Рассмотрим пример. В листинге 12.10 приведена функция `skipFramesPlay()`, позволяющая увеличить скорость воспроизведения монтажной линейки основного фильма или клипа за счет пропускания заданного количества кадров. В результате выполнения данной функции воспроизводящая головка переходит с предыдущего кадра не в следующий, как это происходит по умолчанию, а в кадр, отстоящий от него на указанное количество кадров. В результате зритель видит не каждый кадр исходного фильма, а каждый второй, каждый третий и т. д. Это приводит к тому, что зритель в течение времени экспонирования двух кадров видит фазы анимации, временной интервал между экспонированием которых при обычном способе воспроизведения длится в два, в три и т. д. раза дольше.

Представленная функция `skipFramesPlay()` принимает два параметра: параметр `obj` — это объект, на монтажную линейку которого будет распространяться действие функции, параметр `skipFrames` — это количество пропускаемых кадров. Принцип работы нашей функции заключается в том, что она заставляет воспроизводящую головку монтажной линейки объекта `obj` переходить на кадр, отстоящий от текущего кадра (номер которого содержится в свойстве `obj_.currentframe`) на значение, заданное параметром `skipFrames`. Использование обработчика события `onEnterFrame` позволяет осуществлять такой переход с частотой, равной скорости воспроизведения фильма, что позволяет получить полный контроль над воспроизводящей головкой.

Листинг 12.10. Изменение скорости воспроизведения за счет пропускания кадров

```
function skipFramesPlay(obj:Object, skipFrames:Number) {
        obj.onEnterFrame = function() {
                this.gotoAndStop(this._currentframe + skipFrames);
                // Если текущий кадр - последний, переходим в начало
                if (this._currentframe == this._totalframes) {
                        this.gotoAndStop(1);
                }
        };
}
```

Ниже демонстрируется вариант применения функции `skipFramesPlay()`. При каждом последующем нажатии на клип-кнопку `frameRateBut` происходит увеличение количества кадров, пропускаемых при выполнении анимации монтажной линейки символа `background`, что приводит к эффекту увеличения скорости. После того как интервал между кадрами устанавливается равным десяти, его значение снова устанавливается равным единице (обычный режим воспроизведения).

```
frameRateBut.onPress = function() {
        /* Если переменная skipFrames, используемая для хранения
        количества пропускаемых кадров, не определена (при первом
        выполнении функции), присваиваем ей значение 1 */
        if (skipFrames == undefined) {
                skipFrames = 1;
        }
        /* Обнуляем значение skipFrames после достижения максимального
        значения */
        if (skipFrames == 10) {
                skipFrames = 0;
        }
        // При каждом нажатии инкрементируем значение skipFrames
        skipFrames ++;
/* Заставляем воспроизводящую головку отображать кадры монтажной линейки
клипа background с интервалом, заданным переменной skipFrames */
        skipFramesPlay(background, skipFrames);
};
        /* Поскольку при использовании сложной анимации быстродействие
        может значительно снизиться, установим лишь частичное сглаживание
        графики */
_quality = "MEDIUM";
```

Функция и метод *gotoAndPlay()*

Данная функция (метод) по всем особенностям применения абсолютно идентична рассмотренной выше функции (методу) `gotoAndStop()`. Различие отражено в самом названии. Функция (метод) `gotoAndPlay()` осуществляет переход к заданному кадру фильма и продолжает воспроизведение с этого кадра, в то время как функция `gotoAndStop()` останавливает воспроизведение после перехода.

Глобальная функция `gotoAndPlay()` имеет следующий синтаксис:

`gotoAndPlay(`*сцена, кадр*`)`

Метод объекта `MovieClip` имеет следующий синтаксис:

объект`.gotoAndPlay (`*кадр*`)`

Описание параметров и различия в использовании глобальной функции или метода `gotoAndPlay()` аналогичны соответствующему описанию, приведенному для функции (метода) `gotoAndStop()`.

С помощью функции `gotoAndStop()` можно осуществлять зацикливание воспроизведения произвольного участка кадров монтажной линейки. Так, например, следующая строка кода, будучи помещенной в последний кадр сцены, зацикливает воспроизведение монтажной линейки данной сцены между третьим и последним кадрами:

```
gotoAndStop(3);
```

Функция и метод *nextFrame()*

Глобальная функция `nextFrame()` переводит воспроизводящую головку на следующий по отношению к текущему кадр монтажной линейки, содержащей вызов данной функции, и останавливает ее воспроизведение. Данная функция не принимает параметров. Применение функции `nextFrame()` эквивалентно использованию следующей записи:

```
gotoAndStop(_currentframe + 1);
```

Следует отметить, что если текущий кадр основной монтажной линейки является последним кадром сцены, вызов функции `nextFrame()` переводит воспроизводящую головку в первый кадр следующей сцены.

При вызове `nextFrame()` в качестве метода объекта `MovieClip` используется следующий синтаксис:

```
объект.nextFrame()
```

Здесь `объект` — это объект, к монтажной линейке которого применяется метод `nextFrame()`.

Следующий сценарий перемещает воспроизводящую головку клипа `mc` на кадр вперед и останавливает воспроизведение при каждом нажатии на кнопку с именем `fwardBut`.

```
fwardBut.onPress = function(){
mc.nextFrame();
};
```

Функция и метод *prevFrame()*

Глобальная функция `prevFrame()` переводит воспроизводящую головку на предыдущий по отношению к текущему кадр монтажной линейки, содержащей вызов данной функции, и останавливает ее воспроизведение. Данная функция не принимает параметров. Применение функции `prevFrame()` эквивалентно использованию следующей записи:

```
gotoAndStop(_currentframe - 1);
```

Следует отметить, что если текущий кадр основной монтажной линейки является первым кадром сцены, вызов функции `prevFrame()` переводит воспроизводящую головку в последний кадр предыдущей сцены.

При вызове `prevFrame()` в качестве метода объекта `MovieClip` используется следующий синтаксис:

```
объект. prevFrame()
```

Здесь `объект` — это объект, к монтажной линейке которого применяется метод `prevFrame()`.

Следующий сценарий перемещает воспроизводящую головку клипа `mc` на кадр назад и останавливает воспроизведение при каждом нажатии на кнопку с именем `bwardBut`.

```
bwardBut.onPress = function(){
mc.prevFrame();
};
```

Функции *nextScene()* и *prevScene()*

Глобальные функции `nextScene()` и `prevScene()` напоминают функции `nextFrame()` и `prevFrame()`, однако переводят воспроизводящую головку не в соседний кадр, а на соседнюю сцену.

Функция `nextScene()` не принимает параметров и перемещает воспроизводящую головку в первый кадр сцены, следующей за сценой, содержащей вызов данной функции. После перехода воспроизведение останавливается.

Функция `prevScene()` не принимает параметров и перемещает воспроизводящую головку в первый кадр сцены, предшествующей сцене, содержащей вызов данной функции. После перехода воспроизведение останавливается.

В связи с тем, что данные функции управляют переходом между сценами, они могут вызываться только с основной монтажной линейки. По этой причине в классе `MovieClip` отсутствуют одноименные методы.

Управление воспроизведением монтажной линейки

Для того чтобы дать пользователю возможность управлять воспроизведением монтажной линейки фильма, можно поместить в него кнопки, снабдив их соответствующими сценариями. В качестве примера можно рассмотреть организацию управления воспроизведением основной монтажной линейки, содержащей анимацию. Для этого предлагается снабдить фильм следующими элементами управления:

- кнопка запуска анимации ();
- кнопка паузы ();
- кнопка перехода к началу фильма с последующей остановкой ();
- кнопки покадровой перемотки вперед () и назад ().

Для краткости предположим, что фильм состоит только из одной сцены. Чтобы реализовать данный пример, необходимо создать отдельный слой, содержащий кнопки управления фильмом. Следует иметь в виду, что панель с кнопками, управляющими воспроизведением, должна присутствовать на монтажной линейке в течение всего времени воспроизведения фильма. В качестве кнопок управления можно использовать кнопки стандартной библиотеки **Window>Other Panels>Common Libraries> Buttons**, расположенные в папке Playback. В листинге 12.11 приводятся сценарии используемых кнопок. Требуемое действие происходит при нажатии соответствующей кнопки или при использовании клавиатурной клавиши, указанной событием `keyPress`.

Листинг 12.11. Управление воспроизведением основной монтажной линейки с помощью кнопок

```
//Сценарий кнопки запуска анимации
on(press, keyPress "<Enter>"){
       _root.play();
}
// сценарий кнопки паузы
on(press, keyPress "<Space>"){
       _root.stop();
}
// сценарий кнопки покадровой перемотки вперед
on(press, keyPress "<Right>"){
       _root.nextFrame();
}
// сценарий кнопки покадровой перемотки назад
on(press, keyPress "<Left>"){
       _root.prevFrame();
}
// сценарий кнопки перехода к первому кадру с последующей остановкой
on(press, keyPress "<Backspace>"){
       _root.gotoAndStop(1);
}
```

Примечание

Для того чтобы элиминировать воздействие горячих клавиш среды тестирования, их нужно отключить с помощью команды главного меню среды тестирования **Control> Disable Keyboard Shortcuts**.

Организация навигации внутри фильма

Навигацией называется перемещение между различными частями проекта, осуществляемое пользователем. Любой проект, как правило, состоит из нескольких тематических разделов, освещающих различные его стороны. Так, например, электронная презентация портфолио может включать следующие разделы: "Об авторе", "Живопись", "Графика", "Дизайн" и т. д. Flash предоставляет различные варианты организации структуры проекта и реализации навигации по его разделам. Один из таких вариантов заключается в размещении содержимого разделов в соответствующих кадрах монтажной линейки или сценах документа. Функции управления воспроизведением позволяют реализовать переход между разделами, перемещая воспроизводящую головку к указанному кадру. Применение данного подхода позволяет поместить весь проект целиком в один конечный файл SWF.

Один из подходов к организации структуры проекта заключается в размещении его частей в различных кадрах или сценах одного и того же документа. Каждая часть, являясь самостоятельным тематическим разделом, должна быть снабжена панелью навигации, позволяющей переместиться к другой части проекта, размещенной в указанном кадре заданной сцены. В общем случае каждый раздел проекта может занимать на монтажной линейке один кадр или целый диапазон кадров, что зависит от его внутренней организации.

Каждый тематический раздел проекта может быть оформлен как муви-клип, содержащий на своей монтажной линейке все составные элементы фильма в виде вложенных символов, других графических объектов, звука и сценариев. В этом случае экземпляру данного символа достаточно отвести один кадр основной монтажной линейки документа. При таком подходе кадры, содержащие различные разделы, могут находиться в пределах одной сцены или каждый из них может быть помещен на свою отдельную сцену. В первом случае кадры имеет смысл снабдить метками, указывающими на содержание кадра и позволяющими обратиться к данному кадру вне зависимости от его порядкового номера.

Отдельные разделы проекта могут занимать целый диапазон кадров основной монтажной линейки. В этом случае целесообразно размещать такие разделы на разных сценах. Это позволит лучше организовать структуру фильма и легче ориентироваться в документе на стадии его разработки.

Кроме того, возможен вариант, представляющий собой комбинацию описанных случаев. Некоторые разделы проекта могут состоять из одного-единственного кадра, другие могут включать множество кадров основной монтажной линейки. В этой ситуации также имеет смысл прибегнуть к организации структуры документа с помощью сцен.

Размещение различных частей проекта в одном документе, как правило, приводит к получению конечного файла достаточно большого размера и, соответственно, к увеличению времени его передачи по сети. В связи с этим данный подход может быть реализован при разработке мультимедийной презентации, предназначенной для распространения на компакт-дисках и иных электронных носителях, либо при создании Web-узла в расчете на аудиторию, обладающую возможностью высокоскоростного доступа к Интернету. Этот подход, однако, обладает некоторыми преимуществами. Во-первых, он прост в реализации и, во-вторых, после того как файл проекта загру-

жен по сети, переходы между разделами осуществляются мгновенно, и зрителю не приходится дожидаться загрузки каждой части в отдельности.

Размещая несколько тематических разделов проекта в одном Flash-документе, следует иметь в виду следующие правила.

- Каждый раздел должен быть снабжен панелью навигации, позволяющей пользователю переместиться к другой части проекта. Технически данный переход реализуется с помощью кнопок, содержащих сценарии с использованием функций выбора кадра `gotoAndStop()` или `gotoAndPlay()`.
- Необходимо предотвратить самопроизвольный переход между кадрами фильма и/или сценами документа, содержащими разные тематические разделы проекта. Для этого в последний кадр каждого раздела необходимо поместить функцию `stop()`.
- Если один из разделов содержит анимацию, расположенную на основной монтажной линейке фильма, то для того чтобы зациклить ее воспроизведение, необходимо в последний кадр этой анимации поместить функцию `gotoAndPlay()`, отправляющую воспроизводящую головку в начальный кадр диапазона анимации.
- При переходе к кадру, соответствующему началу требуемого раздела, нужно использовать функцию `gotoAndPlay()` для того, чтобы после перехода начать воспроизведение, или функцию `gotoAndStop()` для того, чтобы после перехода воспроизведение было остановлено.

Программный реверс анимации

Кадры монтажной линейки воспроизводятся последовательно слева направо согласно своей нумерации. Используя возможности ActionScript, можно запускать воспроизведение в обратном направлении, то есть осуществлять реверсирование анимации.

В листинге 12.12 приведена функция, `reversePlay()`, запускающая монтажную линейку заданного объекта в обратном направлении. Данная функция принимает два параметра. Первый параметр `obj` — это ссылка на объект (клип или `_root`), воспроизведение монтажной линейки которого должно выполняться в обратном направлении. Второй параметр `loop` — это флаг, указывающий, должно ли воспроизведение быть зацикленным (`true`) или нет (`false`). Если параметр `loop` принимает значение `true`, воспроизводящая головка, дойдя из последнего кадра в первый, будет вновь возвращаться к последнему кадру и продолжать воспроизведение в обратном направлении. Если параметр `loop` принимает значение `false`, то воспроизводящая головка, дойдя из последнего кадра в первый, будет в нем остановлена. Параметр `loop` является необязательным. По умолчанию его значение принимается равным `true`.

Принцип действия данной функции заключается в том, что она вызывает метод `prevFrame()` объекта `obj` с частотой, равной скорости воспроизведения, используемой в документе. Для того чтобы при запуске функции `reversePlay()` воспроизводящая головка автоматически перемещалась в последний кадр, вводится переменная-флаг `oncePlayed`, которая принимает значение `true`, после того как воспроизводящая головка оказывается в последнем кадре. В дальнейшем переход в последний кадр будет выполняться только в том случае, если воспроизводящая го-

ловка достигнет первого кадра и если параметр `loop` при этом будет принимать значение `true` или будет опущен.

Листинг 12.12. Программный реверс анимации

```
function reversePlay(obj:Object, loop:Boolean):Void {
//obj - объект, анимацию которого нужно запустить в обратном направлении
//loop - зациклено ли воспроизведение
obj.onEnterFrame = function() {
// При запуске функции reversePlay()...
if (!oncePlayed) {
       this.gotoAndStop(this._totalframes); // Переходим в последний
                                            // кадр
       oncePlayed = true; /* Переключаем флаг, чтобы автоматический
       переход в последний кадр больше не выполнялся */
}
/* После того, как воспроизводящая головка была перемещена в последний
кадр ... */
else {
       // Если мы уже в первом кадре...
       if (this._currentframe == 1) {
              /* Если воспроизведение зациклено или второй параметр
              (loop) не указан... */
              if ((loop || loop == undefined)) {
              this.gotoAndStop(this._totalframes); /* Переходим в
              последний кадр */
              }
              // Если воспроизведение не зациклено...
              else {
              stop(); // Останавливаем воспроизведение
              }
       }
       // Если мы еще не в первом кадре...
       else {
       this.prevFrame(); // Переходим на кадр назад
       }
}
};
}
```

Рассмотрим пример практического применения функции reversePlay(). В примере, помещенном на сопровождающий книгу компакт-диск, анимация содержится на основной монтажной линейке, а также на монтажных линейках символов man и dog, расположенных на основной монтажной линейке. Требуется организовать возможность запуска анимации данных объектов в прямом и обратном направлениях. Для этих целей в кадре основной монтажной линейки находятся две кнопки с именами reversePlayBut и forwardPlayBut, запускающие анимацию в обратном и прямом направлениях соответственно. Ниже приведены сценарии, обеспечивающие данную функциональность.

```
//Реверс анимации
reversePlayBut.onPress = function(){
reversePlay(_root, true); // Реверс анимации основной монтажной линейки
reversePlay(man, true); // Реверс анимации монтажной линейки клипа man
reversePlay(dog, true); // Реверс анимации монтажной линейки клипа dog
};
//Запуск анимации в обычном направлении
forwardPlayBut.onPress = function(){
delete _root.onEnterFrame; // Прекращаем реверс анимации _root
delete man.onEnterFrame; // Прекращаем реверс анимации клипа man
delete dog.onEnterFrame; // Прекращаем реверс анимации клипа dog
_root.play(); // Запускаем анимацию _root
man.play(); // Запускаем анимацию клипа man
dog.play(); // Запускаем анимацию клипа dog
};
```

Для запуска анимации в обычном направлении мы удаляем методы обработчика события onEnterFrame, которые были определены для каждого объекта (_root, man и dog) с помощью функции reversePlay(). Это прекращает воспроизведение их монтажных линеек в обратном порядке. Для того чтобы возобновить обычное воспроизведение, используется метод play().

Загрузка сетевого ресурса

Зачастую при размещении Flash-фильма на HTML-странице возникает потребность в выполнении непосредственно из фильма таких операций, как загрузка заданной страницы в окно браузера, выполнение функции JavaScript или отправка переменных фильма серверному сценарию.

Для решения этих задач используется глобальная функция getURL() или одноименный метод класса MovieClip. Различия в синтаксисе глобальной функции getURL() и метода MovieClip.getURL() заключаются только в том, что метод вызывается экземпляром клипа, а для вызова функции не требуется ссылка на вызывающий ее объект. Это различие проявляется только при отправке переменных серверному сценарию. Глобальная функция отправляет переменные текущей монтажной

линейки (то есть линейки, откуда она вызывается), в то время как метод позволяет отправить переменные монтажной линейки вызывающего его клипа. Глобальная функция `getURL()` имеет следующий формат:

```
getURL(URL, окно, метод)
```

Параметр *URL* — это строка, содержащая абсолютный или относительный путь к документу, который требуется загрузить для отображения в окне браузера, или серверному сценарию, который должен быть выполнен. Параметр может быть задан строковым литералом или идентификатором переменной, содержащей строку. Абсолютный путь начинается с указания протокола и логического диска (для локально расположенного файла):

```
"http://www.avalon.ru" //Абсолютный путь к Web-странице
file://C:\\WINDOWS\\Desktop\\Книжные магазины.txt//Путь к локальному файлу
```

Относительный путь задается относительно местоположения текущего документа:

```
"index/pictures/001.html"
"../design/artStudio/art.html"
```

Следует четко понимать, что при использовании относительных ссылок они разрешаются относительно местоположения HTML-страницы, содержащей файл SWF, в котором осуществляется вызов функции `getURL()`.

Примечание

С помощью параметра/атрибута `base` тегов `<object>` и `<embed>`, соответственно, можно задать базовый каталог, относительно которого будут разрешаться все относительные ссылки, используемые в SWF-файле, размещенном на данной HTML-странице.

Параметр *окно* — строка, указывающая окно браузера или фрейм, в который будет загружен запрашиваемый документ или в котором будут выведены результаты выполнения удаленного сценария, местоположение которого задается параметром *URL*. В качестве данного аргумента можно использовать пользовательское имя окна или фрейма или одно из зарезервированных значений:

- `_blank` — загрузка в новое окно браузера;
- `_self` — загрузка в текущее окно браузера или фрейм;
- `_parent` — загрузка в родительский фрейм по отношению к текущему фрейму;
- `_top` — загрузка в самый верхний в иерархии фрейм.

Этот параметр является необязательным. Если его не указывать, документ будет загружен в текущее окно браузера или фрейм (при фреймовой структуре).

Параметр *метод* — строка, задающая метод передачи переменных текущей монтажной линейки во внешний удаленный сценарий, заданный параметром *URL*. В качестве значения данного параметра можно указать один из следующих методов:

- `"GET"` — переменные отправляются в виде строки запроса, которая присоединяется к URL;

- "POST" — переменные отправляются в виде отдельного блока данных после заголовка запроса HTTP POST.

При вызове getURL() в качестве метода клипа серверному сценарию будут отправлены все переменные монтажной линейки данного клипа.

Данный параметр является необязательным.

Следующий пример содержит сценарий кнопки, нажатие которой приведет к загрузке в новое окно браузера страницы, расположенной по адресу **www.avalon.ru**.

```
on(press){
       getURL("http://www.avalon.ru","_blank");
}
```

Функцию getURL() можно использовать для вызова почтовой программы и автоматической передачи ей требуемых данных для отправки почтового сообщения. Для этого в аргументе *URL* необходимо использовать ключевое слово mailto:. Следующий сценарий кнопки при ее нажатии откроет используемую в системе пользователя почтовую программу и передаст ей адрес, по которому можно будет послать письмо (для отправки необходимо выполнить соответствующую команду почтового приложения).

```
on(press){
getURL("mailto:book_flash@mail.ru");
}
```

Несколько усложнив данную конструкцию, можно переслать почтовому приложению имя адресата так, что оно будет выведено в строке адреса:

```
on(press){
getURL("mailto:Dmitri.I.Albert< book_flash@mail.ru>");
}
```

Для того чтобы указать тему сообщения и переслать его текст, можно воспользоваться следующей записью:

```
on(press){
getURL("mailto:Dmitri.I.Albert< book_flash@mail.ru>?subject=Тема
сообщения&body=Текст сообщения");
}
```

С помощью функции getURL() также можно вызвать функции JavaScript или VBScript, используя следующий формат:

```
getURL("javascript:functionName") // вызов функции JavaScript
getURL("vbscript:functionName") // вызов функции VBScript
```

Здесь functionName — вызываемая функция. Следующий пример содержит сценарий кнопки, нажатие которой приведет к появлению системного окна с сообщением:

```
on(press){
getURL("javascript:alert('Выполнение функции alert()')");
}
```

Организация навигации между Web-страницами

В качестве альтернативы размещению всех частей проекта в одном файле SWF может быть применен подход, в соответствии с которым каждый раздел проекта помещается в отдельный SWF-файл, размещаемый на собственной HTML-странице. Каждый фильм, являющийся частью проекта, должен быть снабжен панелью навигации, позволяющей осуществить переход к другому разделу посредством загрузки HTML-страницы, содержащей соответствующий SWF-файл. В результате на сервере необходимо разместить файлы SWF и содержащие их HTML-страницы. Данный подход позволяет обеспечить более быструю загрузку частей проекта и может быть использован при разработке Web-узла.

При размещении частей проекта на нескольких HTML-страницах следует иметь в виду следующие особенности.

- При создании сценариев, обеспечивающих возможность перехода с одной HTML-страницы на другую, целесообразно использовать относительную адресацию. Относительные ссылки задаются в качестве параметра функции `getURL()` в рабочей среде Flash, но по умолчанию разрешаются относительно фактического местоположения HTML-страницы, содержащей SWF-файл, в котором производится вызов данной функции. В связи с этим на стадии разработки необходимо четко представлять структуру взаимного расположения HTML-страниц, на которых будут размещены различные SWF-документы, входящие в состав проекта. С помощью параметра/атрибута `base` тегов `<object>` и `<embed>`, соответственно, можно задать базовый каталог, относительно которого будут разрешаться все относительные ссылки, используемые в SWF-файле, размещенном на HTML-странице. Этой возможностью можно воспользоваться, если HTML-страницы, содержащие различные части проекта, находятся в разных каталогах. Если относительный путь в аргументе функции `getURL()` указан неверно, требуемая страница не сможет быть загружена.
- Для того чтобы SWF-документ мог быть отображен на HTML-странице, необходимо верно указать путь к нему в HTML-коде с помощью параметра `movie` и атрибута `src` тегов `<object>` и `<embed>` соответственно. Если путь будет указан неверно, файл не сможет быть загружен.

Динамическое создание клипов

Помимо создания экземпляров клипов в рабочей среде Flash ActionScript предоставляет широкие возможности создания экземпляров программным образом непосредственно в процессе воспроизведения фильма. Существуют три способа динамического создания экземпляров:

- дублирование имеющихся экземпляров с помощью метода или функции `duplicateMovieClip()`;
- размещение в фильме экземпляра клипа или кнопки, находящегося в библиотеке, с помощью метода или функции `attachMovie()`;
- создание нового пустого клипа с помощью метода `createEmptyMovieClip()`.

Прежде чем перейти к подробному описанию этих возможностей, необходимо рассмотреть существующий в Flash стек графических объектов.

Стеки графических объектов Flash

Читатель уже привык к тому, что при создании экземпляров клипов и кнопок в рабочей среде они размещаются в стеке слоев, визуальное представление которого находится в панели **Timeline**. Слои управляют порядком наложения объектов, поскольку содержимое вышележащих слоев перекрывает содержимое нижележащих слоев. В то же время в пределах каждого слоя в одном и том же кадре фильма может одновременно располагаться несколько графических объектов, причем объекты наложенного уровня формируют стопку, в которой каждый объект занимает определенное положение и где действует тот же принцип, что и в стеке слоев. Объект, расположенный выше в стопке, перекрывает объекты, расположенные под ним. При этом следует помнить, что объекты рабочего уровня (заливки и контуры) всегда находятся в самом нижнем уровне стопки, а объекты наложенного уровня (экземпляры символов, группы, текстовые блоки и т. д.) располагаются на вышележащих уровнях стопки, причем каждый объект находится на своем личном уровне.

Экземпляры символов располагают своей собственной монтажной линейкой и, соответственно, своим стеком слоев, в котором действуют те же правила, что и в стеке слоев основной монтажной линейки.

При работе с графикой в рабочей среде документа порядком размещения графических объектов по уровням стека можно управлять "вручную", используя слои для организации внутри фильма и команды главного меню **Modify>Arrange** для организации элементов в пределах конкретных слоев. Однако динамическое создание экземпляров требует точного указания их местоположения в структуре документа. В связи с этим необходимо понять, как структура стеков, используемая в рабочей среде, реализуется с точки зрения ее программной обработки.

Все графическое содержимое, расположенное на основной монтажной линейке (`_root`), размещается в *стеке основной монтажной линейки*. Этот стек можно мысленно представить себе в виде набора слоев или *уровней*. Каждый уровень стека имеет свой уникальный номер, называемый *z-индексом* или *глубиной* (depth). Значения номеров уровней стека лежат в диапазоне от −16 384 до 1 048 575. Уровень с глубиной −16 384 располагается в самом низу стека, а уровень с глубиной 1 048 575 находится в самом верху. Каждый объект наложенного уровня (экземпляр символа, группа, текстовый блок, экземпляр растрового изображения, видеоролик) занимает свой персональный уровень стека. На одном уровне стека может располагаться только один объект наложенного уровня. В то же время все объекты рабочего уровня, лежащие в пределах одного слоя монтажной линейки, занимают один общий уровень стека. Объекты не обязательно должны занимать последовательно расположенные уровни стека. Так, например, один объект может находиться на уровне 0, а другой — на уровне 200, при этом уровни с промежуточными значениями глубины могут оставаться свободными.

Каким же образом стек слоев, используемый в рабочей среде документа, проецируется на стек графических объектов основной монтажной линейки?

При выполнении такого "проецирования" имеют место следующие закономерности.

- На каждый слой монтажной линейки отводится как минимум два уровня стека. Нижний из этих уровней предназначен для размещения объектов рабочего уровня (заливок и контуров), вышележащие уровни предназначены для размещения объектов наложенного уровня (экземпляров клипов, групп и т. д.).
- Самому нижнему слою монтажной линейки соответствуют нижние уровни стека. Принимая во внимание, что самый нижний уровень стека имеет глубину −16 384, отметим, что объекты рабочего уровня самого нижнего слоя попадают на уровень стека с глубиной −16 383, а самый нижний объект наложенного уровня данного слоя попадает на следующий уровень стека с глубиной −16 382.
- Если на слое нет объектов рабочего уровня, то нижний объект наложенного уровня занимает уровень стека, соответствующий данному слою, имеющий меньшую глубину. Так, в отсутствие объектов рабочего уровня самый нижний экземпляр символа нижнего слоя монтажной линейки будет помещен на уровень стека с глубиной −16 383. Если же на нижнем слое монтажной линейки будут расположены объекты рабочего уровня, то они попадут на уровень стека с глубиной −16 383, а самый нижний экземпляр символа займет следующий уровень стека с глубиной −16 382.
- Пустые слои монтажной линейки не проецируются на уровни стека.

Графические объекты могут размещаться не только на основной монтажной линейке, но и на монтажной линейке символов, в частности, клипов. В соответствии с этим каждый клип содержит свой собственный стек графических объектов, называемый *стеком клипа*. В стеке клипа действуют те же правила и закономерности, которые имеют место для стека основной монтажной линейки. Следует понимать, что поскольку клип размещается на основной монтажной линейке фильма, то и его стек, обладая полным набором собственных уровней, тем не менее, представляет собой своеобразный подстек стека основной монтажной линейки. Все уровни этого подчиненного стека располагаются в пределах единственного уровня стека основной линейки.

Все методы, позволяющие реализовать динамическое создание экземпляра, требуют указания уровня стека, на который будет помещен создаваемый экземпляр. При этом следует понимать, что объекты, расположенные на уровнях стека с большими значениями глубины, будут перекрывать (заслонять) объекты, расположенные на уровнях с меньшей глубиной.

Одной из важнейших особенностей стека является то, что каждый его уровень может быть занят только одним объектом. Если динамически созданный экземпляр помещается на занятый уровень стека, то содержимое этого уровня удаляется, чтобы освободить место для нового элемента. Этот факт необходимо хорошо помнить, поскольку очень часто возникает потребность в создании нескольких экземпляров. В этом случае каждый из них необходимо поместить на отдельный уровень стека, иначе создание следующего экземпляра будет приводить к удалению предыдущего, помещенного на тот же уровень стека.

Заполнение стека объектами, созданными в рабочей среде, начинается снизу, то есть с уровней с отрицательными значениями глубины. В связи с этим размещение динамически создаваемых экземпляров на уровнях стека, занятых содержимым, соз-

данным в рабочей среде, приведет к его удалению. Для того чтобы избежать связанных с этим возможных неприятностей, при создании экземпляров программными средствами следует размещать их на уровнях стека с положительными значениями глубины (вряд ли для размещения содержимого монтажной линейки в рабочей среде будет задействовано более 16 с лишним тысяч уровней!).

Стек документов проигрывателя

Помимо стеков графических объектов фильма и клипов существует еще один глобальный стек, называемый *стеком документов проигрывателя*. В стек документов загружаются не просто графические объекты, а целые внешние файлы (SWF или JPEG), которые размещаются на разных уровнях стека документов друг над другом, причем вышележащие документы перекрывают нижележащие. Уровни стека документов подобно слоям в рабочей среде не имеют фона, то есть являются прозрачными в тех областях, на которых отсутствуют графические объекты. При этом объекты, расположенные на вышележащих уровнях стека документов, перекрывают объекты нижележащих уровней. Это означает, что зритель может одновременно видеть содержимое нескольких документов. Как и в стеке монтажной линейки, в стеке документов на каждом уровне может находиться только один документ. Если новый документ загружается на уже занятый уровень стека документов, то расположенный там документ выгружается, и его место занимает новый. Для обращения к уровню стека документа используется идентификатор `_level`, сопровождаемый порядковым номером уровня стека документов. Нумерация уровней начинается с нуля, то есть основной документ всегда находится на нулевом уровне, к которому можно обратиться с помощью идентификатора `_level0` (именно этот идентификатор используется при выводе ссылок на объекты в панель **Output**). При загрузке внешнего документа на уровень `_level0`, текущий документ со всем своим содержимым выгружается из проигрывателя, и его место занимает новый загружаемый документ. Уровни стека документов так же, как и уровни стека графических объектов, не обязательно должны располагаться последовательно. Можно загрузить один документ на уровень `_level1`, а следующий — сразу на `_level10`.

Необходимо иметь в виду, что глобальные параметры, такие как скорость воспроизведения и цвет рабочей области, устанавливаются для всех загруженных в проигрыватель документов на основании значений этих настроек, определенных в документе, помещенном на нулевой уровень стека (`_level0`). Так, если в основном документе, загруженном на нулевой уровень стека, скорость воспроизведения монтажной линейки составляет 12 кадров в секунду, то все документы, загруженные в проигрыватель на более высокие уровни, будут воспроизводиться с той же самой скоростью.

Присоединение экземпляра. Метод *attachMovie()*

Метод `attachMovie()` позволяет динамически создавать экземпляры символов, находящихся в библиотеке документа, и размещать их в стеке основной монтажной линейки или целевого клипа. Фактически данный метод представляет собой программную реализацию процедуры, выполнение которой в рабочей среде осуществляется путем "вытаскивания" экземпляра из библиотеки на сцену.

Данный метод может быть использован для создания экземпляров символов любого типа: MovieClip, Button или Graphic. Динамически созданные экземпляры символов типа MovieClip и Graphic представляют собой объекты `MovieClip`, то есть обладают поведением клипа. Поскольку в ActionScript не существует встроенного класса для программной обработки символов типа Graphic, поведение динамически созданных экземпляров графических символов автоматически переопределяется. Динамически созданные экземпляры символа типа Button представляют собой объекты `Button`, то есть кнопки.

Настройка параметров экспорта

Для того чтобы получить возможность динамического создания экземпляров символа, созданного в рабочей среде, необходимо указать, что данный символ будет использоваться в качестве источника программно генерируемых экземпляров (то есть экспортироваться для обработки инструкциями ActionScript), и снабдить его уникальным идентификатором связи (linkage identifier). Для установки данных настроек в рабочей среде необходимо выполнить следующие действия.

1. Правой кнопкой мыши щелкнуть на символе в окне библиотеки (панель **Library**) и из появившегося контекстного меню выбрать команду **Linkage...**.
2. В диалоговом окне **Linkage Properties** (рис. 12.5) установить флажок **Export for ActionScript**.
3. В поле **Identifier** ввести идентификатор связи, состоящий из произвольного набора символов. Идентификатор связи каждого объекта библиотеки, используемого в качестве источника программно генерируемых экземпляров, должен быть уникальным.
4. Установка флажка **Export in first frame** приводит к тому, что экспортируемый для ActionScript символ будет загружен до начала загрузки содержимого первого кадра фильма. Это может вызвать нежелательную задержку перед началом воспроизведения и выполнения сценариев первого кадра фильма.

   Снятие флажка **Export in first frame** предотвращает загрузку экспортируемого символа перед первым кадром фильма, однако в этом случае необходимо, чтобы экземпляр данного символа был вручную (то есть в рабочей среде) помещен в кадр монтажной линейки, перед выполнением вызова метода `attachMovie()`. Это необходимо сделать для того, чтобы данный символ загрузился и мог быть использован в качестве источника экземпляров. Если к моменту вызова метода `attachMovie()` символ еще не успел загрузиться, создания его экземпляра не произойдет.

   Снятие флажка **Export in first frame** имеет смысл в том случае, если динамическое создание экземпляров должно выполняться не в первом кадре и если объем символа достаточно велик, что приводит к ощутимой задержке перед загрузкой первого кадра. Для того чтобы экземпляр символа, в рабочей среде помещенный на сцену, был не виден, можно воспользоваться свойством `_visible` или просто поместить его за пределы рабочей области.

   Поле **AS 2.0. Class** используется для указания пути к классу, все свойства и методы которого будут унаследованы динамически созданным клипом. Подробно эта возможность будет рассматриваться в *главе 25*.

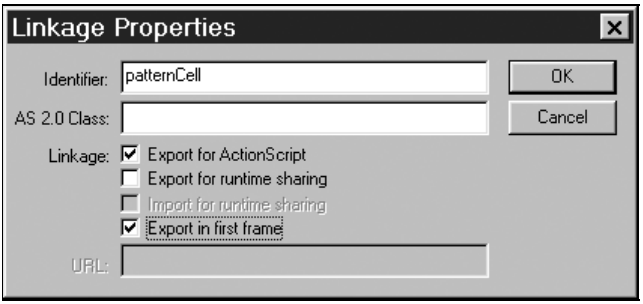


Рис. 12.5. Диалоговое окно **Linkage Properties**

Синтаксис

Метод `attachMovie()` имеет следующий синтаксис:

объект.attachMovie(*идентификатор связи*, *имя*, уровень, *объект инициализации*)

Здесь *объект* — это родительский объект, в стек которого будет помещен динамически созданный экземпляр. В качестве родительского объекта может выступать основная монтажная линейка (`_root`) или клип. Если родительским объектом выступает `_root`, то созданный экземпляр будет помещен на заданный уровень стека основной монтажной линейки. Если родительским объектом является клип, то созданный экземпляр будет помещен на заданный уровень стека монтажной линейки клипа.

Параметр *идентификатор связи* — это строка или выражение, возвращающее строку, содержащую идентификатор, заданный в поле **Identifier** диалогового окна **Linkage Properties**. Этот параметр определяет символ, служащий источником экземпляров.

Параметр *имя* — это строка или выражение, возвращающее строку, задающую имя создаваемому экземпляру. Для того чтобы при создании нескольких экземпляров каждый из них имел уникальное имя, данный параметр задается выражением. Такое выражение обычно представляет собой соединение (конкатенацию) неизменных строковых фрагментов с уникальными числовыми значениями.

Параметр *уровень* — это число или выражение, возвращающее число, определяющее уровень стека, на который будет помещен создаваемый экземпляр. Значение данного параметра представляет собой целое число, которое должно находиться в диапазоне от −16 384 до 1 048 575. Если указанный данным параметром уровень стека оказывается занят, то его содержимое выгружается (удаляется), и его место занимает создаваемый экземпляр.

Параметр *объект инициализации* — это объект, свойства которого передаются создаваемому экземпляру. Объект может быть представлен с помощью литерала или идентификатором, содержащим ссылку на объект. Данный параметр является необязательным и может быть опущен. Применение этого параметра может оказаться весьма полезным при необходимости установки значений свойств или для инициализации переменных монтажных линеек динамически создаваемых экземпляров.

Так, например, для того чтобы каждый экземпляр был наполовину прозрачным, можно передать методу attachMovie() объект вида:

```
{_alpha:50}
```

Возвращаемым значением метода attachMovie() является ссылка на созданный им экземпляр.

Следует отметить, что динамически созданный экземпляр по умолчанию позиционируется таким образом, что его точка регистрации совпадает с началом координат родительского объекта. Начало координат основной монтажной линейки находится в левом верхнем углу рабочей области, а начало координат клипа находится в его точке регистрации, обозначаемой перекрестием.

Примечание

Экземпляр клипа, созданный с помощью метода attachMovie(), так же, как и клип, созданный в рабочей среде, обладает собственным стеком, в котором могут размещаться новые динамически созданные экземпляры. Это означает, что такой клип может, в свою очередь, вызывать метод attachMovie() для динамического присоединения экземпляров символов из библиотеки документа.

Следующая строка кода создает экземпляр клипа с идентификатором связи id. Созданный экземпляр получает имя newClip и помещается на первый уровень стека _root.

```
_root.attachMovie("id", "newClip", 1);
```

Для того чтобы сместить клип newClip, можно воспользоваться его свойствами _x и _y. При этом важно четко понимать, как расположена точка регистрации newClip, поскольку именно ее координаты ассоциируются с координатами newClip. Допустим, что точка регистрации находится точно в центре клипа, а рабочая область имеет размеры 550×400 пикселов. Следующий код отцентрирует клип newClip относительно рабочей области.

```
newClip._x = 275;
newClip._y = 200;
```

В действительности позиционирование динамически созданного экземпляра может быть выполнено непосредственно при его создании. Для этих целей используем четвертый, необязательный параметр метода attachMovie(), передав этому методу объект, содержащий два свойства _x и _y:

```
root.attachMovie("id", "newClip", 1, {_x:275, _y:200});
```

Данная строка кода заменяет три строки, приведенных выше. Здесь осуществляется создание экземпляра newClip, свойствам _x и _y которого присваиваются значения, заданные с помощью свойств инициализирующего объекта.

При многократном создании экземпляров необходимо, чтобы имя каждого вновь создаваемого экземпляра и номер уровня стека, на который он помещается, задавались динамически, поскольку при загрузке очередного экземпляра на занятый уровень стека расположенный на нем экземпляр выгружается.

Следующий сценарий помещает в стек клипа but новый экземпляр символа с идентификатором "id" при каждом нажатии на клип but. Переменная i, увеличивая

свое значение на единицу при каждом нажатии клипа, выполняет роль счетчика и используется при формировании уникальных имен экземпляров и задании уникального уровня стека. Таким образом, в результате выполнения этого сценария при каждом нажатии клипа `but` на первый уровень его стека помещается экземпляр с именем `newClip1`, на второй уровень помещается экземпляр `newClip2`, на третий уровень — экземпляр `newClip3` и т. д. Для того чтобы все экземпляры были видны, а не размещались друг над другом в одних и тех же координатах, каждый последующий экземпляр смещается относительно предыдущего на 50 пикселов вправо. Это реализуется с помощью инициализирующего объекта, передаваемого методу `attachMovie()`. Значение свойства `_x`, определенного в данном объекте, ставится в зависимость от значения счетчика `i` и, таким образом, в каждый экземпляр копируется уникальное значение свойства `_x`.

```
but.onPress = function() {
       if(i==undefined) i=1
       this.attachMovie("id", "newClip" + i, i, {_x:i * 50});
       i++;
};
```

Создание регулярного узора

Рассмотрим практический пример, иллюстрирующий возможности динамического присоединения экземпляров. В листинге 12.13 приведена функция `setPattern()`, позволяющая создавать регулярное заполнение прямоугольной области экземплярами, порожденными символом, расположенным в библиотеке документа.

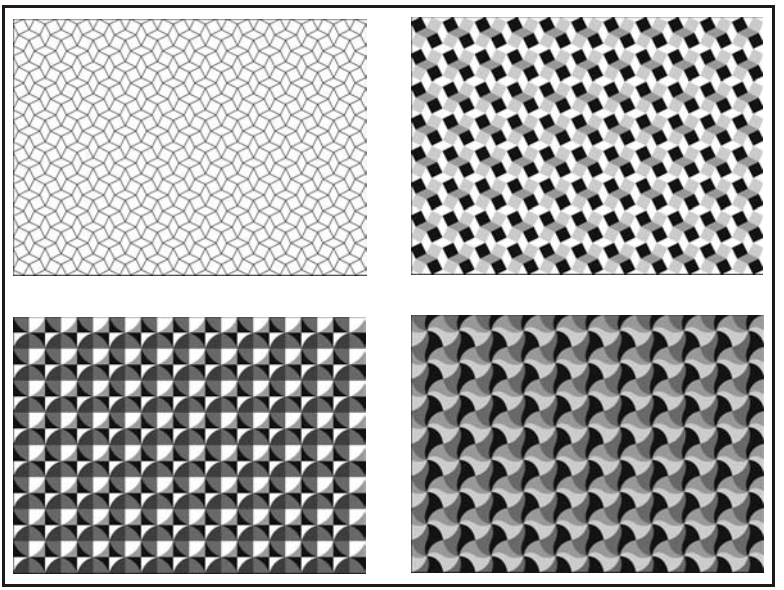


Рис. 12.6. Результаты применения функции `setPattern()` с различными символами

В результате ее применения формируется регулярный узор или "паттерн[1]" подобный представленным образцам на рис. 12.6.

Формирование узора реализуется за счет динамического создания экземпляров символа с идентификатором связи "patternCell". Создаваемые экземпляры размещаются по столбцам и строкам, формируя матрицу заданного размера. Все создаваемые экземпляры получают имена вида cell*c*_*r*, где *c* — это номер столбца, а *r* — это номер строки элемента. Функция предполагает возможность задания равных промежутков между элементами по вертикали и горизонтали, а также позволяет задать местоположение левого верхнего элемента прямоугольной области регулярного заполнения.

В объявлении функции формальные параметры намеренно помещены на отдельных строках, для того чтобы их можно было снабдить комментариями, разъясняющими назначение каждого параметра.

Листинг 12.13. Создание регулярного заполнения на основе динамического присоединения экземпляров

```
function setPattern(
       obj:Object,  //Объект, в стек которого будут помещены элементы
       linkageID:String,//Идентификатор связи
       nRow:Number,  //Количество рядов
       nCol:Number,  //Количество столбцов
       xSpace:Number, //Расстояние между элементами по X
       ySpace:Number, //Расстояние между элементами по Y
       initX:Number,  //Координата левого верхнего элемента по X
       initY:Number  //Координата левого верхнего элемента по Y
                     ):Void{
//Создание и размещение экземпляров
for (var i=1; i<=nRow; i++) { //Формирование строк
       for (var j=1; j<=nCol; j++) { //Формирование столбцов
       //Создание экземпляров
       obj.attachMovie(linkageID, "cell"+i+"_"+j, i*nCol+j)
       //Позиционирование экземпляров
       obj["cell"+i+"_"+j]._x = (j-1)*(obj.cell1_1._width+xSpace)+initX;
       obj["cell"+i+"_"+j]._y = (i-1)*(obj.cell1_1._height+ySpace)+initY;
              }
       }
}
```

[1] От английского слова *pattern* — узор, шаблон, трафарет. — *Прим. автора.*

Координаты каждого объекта рассчитываются относительно начала координат объекта, указанного параметром obj, исходя из количества созданных ранее элементов. Значения текущего количества элементов в ряду и столбце предоставляются переменными j и i соответственно, исполняющими роль счетчиков циклов. Умножая количество элементов на сумму ширины элемента и промежутка между элементами, получаем местоположение нового элемента. При необходимости смещения области заполнения к полученному значению прибавляем величину отступа, заданного параметрами initX и initY. Поскольку первый элемент должен находиться в начале координат (при отсутствии отступа, заданного параметрами initX и initY), ширина объекта умножается на значение счетчика за вычетом единицы.

Следует отметить, что поскольку все экземпляры имеют одинаковую ширину, то при вычислении координат очередного экземпляра может быть использована ширина любого из них. В настоящем примере "эталоном" ширины служит клип cell1_1, поскольку он создается первым.

Вызов функции setPattern() со следующими параметрами полностью заполнит рабочую область размером 550×400 пикселов экземплярами символа с идентификатором связи "patternCell", размер которого составляет 50×50 пикселов.

```
setPattern(_root, "patternCell", 8, 11, 0, 0, 0, 0);
```

Примечание

Необходимо отметить, что клип, в который был загружен внешний документ (SWF или JPEG), уже не может быть использован для вызова метода attachMovie(). Это означает, что после загрузки в клип внешнего документа в стеке данного клипа не могут быть размещены экземпляры символов из библиотеки его документа.

Дублирование экземпляра. Метод *duplicateMovieClip()*

Метод duplicateMovieClip() позволяет программным образом создавать новые экземпляры клипов, представляющие собой дубликаты клипа, вызвавшего данный метод. Принципиальное отличие данного метода от рассмотренного выше метода attachMovie() заключается в том, что метод attachMovie() создает экземпляры на основе символа, хранящегося в библиотеке, и не требует наличия экземпляра этого символа на сцене. Метод duplicateMovieClip() создает новый экземпляр как копию уже существующего экземпляра клипа, присутствие которого на сцене является необходимым. Еще одно важное отличие заключается в том, что метод duplicateMovieClip() позволяет создавать только экземпляры клипов (объекты MovieClip) и не может применяться по отношению к кнопкам (объектам Button), в то время как метод attachMovie() позволяет создавать экземпляры обоих типов.

Синтаксис

Метод duplicateMovieClip() имеет следующий синтаксис:

клип.duplicateMovieClip(*имя*, уровень, *объект инициализации*)

Здесь *клип* — это дублируемый или порождающий экземпляр клипа. В отличие от метода attachMovie(), который может вызываться объектом _root, метод duplicateMovieClip() вызывается исключительно экземпляром клипа.

Дубликат, созданный в результате применения метода duplicateMovieClip(), всегда помещается в стек объекта, содержащего клип, который вызвал данный метод (то есть в стек объекта, содержащего порождающий клип). Так, дубликат клипа, расположенного на основной монтажной линейке, будет помещен в стек _root поверх всего содержимого, созданного в рабочей среде. Дубликат клипа, расположенного на монтажной линейке клипа mc, будет помещен в стек mc поверх всего содержимого его (клипа mc) монтажной линейки. И, наконец, дубликат клипа amc, помещенного в стек клипа mc с помощью метода attachMovie(), будет расположен в стеке mc, при этом он может располагаться и выше и ниже порождающего клипа (amc) в зависимости от заданного значения уровня.

Параметры метода duplicateMovieClip() идентичны соответствующим параметрам метода attachMovie(), описанным выше. Приведем здесь только краткое их описание.

- Параметр *имя* — это строка или выражение, возвращающее строку, задающую имя создаваемому дубликату клипа.
- Параметр уровень — это число или выражение, возвращающее число, определяющее уровень стека, на который будет помещен создаваемый экземпляр.
- Параметр *объект инициализации* — это объект, свойства которого передаются создаваемому экземпляру. Данный параметр является необязательным.

Возвращаемым значением метода duplicateMovieClip() является ссылка на созданный им экземпляр.

Особенности использования

Необходимо отметить, что дубликат клипа, созданный методом duplicateMovieClip(), наследует все трансформации и цветовые эффекты (brightness, tint, alpha) оригинала. Так, если экземпляр клипа повернут на 45° и имеет 50%-ную прозрачность, то и его дубликат будет повернут на тот же угол и тоже наполовину прозрачен. Местоположение точки регистрации дубликата совпадает с ее положением у оригинала.

При дублировании клипа координаты дубликата совпадают с координатами исходного экземпляра, то есть они размещаются строго друг над другом. В связи с этим, для того чтобы оба экземпляра были видны, один их них после копирования необходимо сместить или трансформировать. Эта операция может быть выполнена путем передачи методу duplicateMovieClip() инициализирующего объекта, определяющего значения соответствующих свойств (например, _x, _y, _width, _height и т. д.), или с помощью дополнительного фрагмента кода.

Важно иметь в виду, что при копировании клипов сценарии, расположенные на их монтажных линейках, копируются в дубликаты. Однако переменные и методы обработчиков событий, определенные для дублируемого клипа не на его собственной монтажной линейке, не копируются в клипы, полученные в результате дублирования. Дубликаты получают только те фрагменты кода, которые непосредственно находятся на монтажной линейке порождающего клипа (то есть сценарии кадров его

монтажной линейки), а также сценарии, прикрепленные к порождающему клипу с применением обработчиков on() и onClipEvent(). Так, если для некоторого клипа определен метод обработчика onEnterFrame, этот метод не будет автоматически передаваться его дубликатам. Однако если к этому клипу прикреплен обработчик onClipEvent(), контролирующий событие enterFrame, данный обработчик будет прикреплен и ко всем порожденным дубликатам. Соответственно, если дубликат, полученный с помощью метода duplicateMovieClip(), должен автоматически получить сценарий породившего его клипа, этот сценарий должен находиться либо на монтажной линейке порождающего клипа, либо прикрепляться к порождающему клипу с помощью обработчиков on() или onClipEvent().

Следующий сценарий при нажатии на клип mc создает его очередной дубликат, отстоящий от предыдущего дубликата на 50 пикселов. Имена дубликатов, как и номера уровней, на которые помещаются экземпляры, формируются динамически. Обратите внимание, что метод обработчика события onPress не передается дубликатам, поскольку его определение не находится ни на монтажной линейке клипа mc, ни в прикрепленном к нему сценарии.

```
mc.onPress = function(){
       if(i==undefined) i=1;
       this.duplicateMovieClip("mc_copy"+i, i);
       this._parent["mc_copy"+i]._x = this._x+i*50;
       i ++;
};
```

Поскольку дубликат клипа помещается в стек объекта, содержащего этот клип, то в приведенном коде для обращения к дубликатам применяется адресация с использованием идентификатора _parent, возвращающего ссылку на объект, содержащий mc. Здесь местоположение дубликата задается отдельной строкой кода. Ту же процедуру можно выполнить, используя объект инициализации. Следующий сценарий эквивалентен предыдущему:

```
mc.onPress = function(){
       if(i==undefined) i=1;
       this.duplicateMovieClip("mc_copy"+i, i, {_x:this._x+i*50});
i++;
};
```

Монтажная линейка дубликата всегда начинает воспроизводиться с начала (с первого кадра), вне зависимости от того, в каком кадре она находилась у порождающего клипа. Это позволяет создавать интересные анимационные эффекты. Так, если на монтажной линейке символа находится анимация цветового перехода, то многократное дублирование и последовательное размещение копий этого клипа создаст эффект движущейся цветной ленты (подобный эффект демонстрируется в примере 15_2.fla, расположенном в папке Глава 15 на сопровождающем книгу компакт-диске).

Каскадное дублирование и трансформация

Динамическое дублирование клипов может быть использовано для создания разнообразных декоративных эффектов. В качестве примера создадим функцию, выполняющую каскадное копирование клипов и применяющую к дубликатам масштабирование и поворот. При каскадном копировании дубликат, созданный на основе исходного клипа, копируется, порождая новый клип, который, в свою очередь, копируется и т. д. При этом трансформация каждого последующего дубликата осуществляется на основе значений параметров трансформации порождающего дубликата. Так, при заданном значении угла поворота 45° первый дубликат будет повернут на 45° по отношению к исходному клипу, второй — на 90°, третий — на 135° и т. д. В рабочей среде для выполнения подобных операций используется кнопка **Copy and apply transform** панели **Transform**.

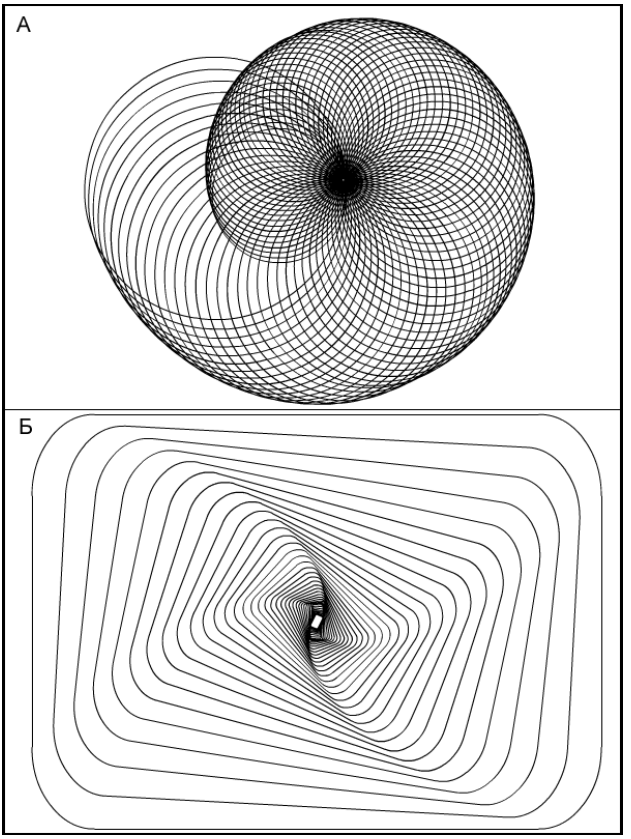


Рис. 12.7. Результаты применения функции `scaleAndRotate()` к различным фигурам: А — исходная фигура окружность, параметры вызова функции `scaleAndRotate(mc, 90, 90, 3, 40)`; Б — исходная фигура прямоугольник со скругленными углами, параметры вызова функции `scaleAndRotate(mc, 101, 101, 5, 75)`

В листинге 12.14 приведена функция `scaleAndRotate()`, выполняющая каскадное дублирование и трансформацию экземпляров с частотой, равной скорости воспроизведения, используемой в документе. Таким образом, в результате вызова данной функции копирование и трансформация клипов выполняются как программная анимация.

В качестве параметров функции передаются ссылка на дублируемый клип, параметры трансформации и количество дубликатов, которые должны быть созданы в результате ее выполнения. Результаты применения данной функции к различным фигурам представлены на рис. 12.7.

Листинг 12.14. Анимация на основе копирования и вращения

```
function scaleAndRotate(
              obj:MovieClip, // Дублируемый клип
              xScale:Number, // Масштаб по горизонтали в процентах
              yScale:Number, // Масштаб по вертикали в процентах
              rotate:Number, // Поворот в градусах
              times:Number  // Число копий клипа и трансформаций
                                  ):Void {
_quality = "MEDIUM";
obj._parent.onEnterFrame = function() {
       if (i == undefined) {//При первом запуске...
              i = 0;
              /* Создаем первый дубликат с именем copy0, помещаем его на
              нулевой уровень стека */
              obj.duplicateMovieClip("copy"+i ,i);
       }
       if (i < times) { /* Пока не выполнилось требуемое количество
       трансформаций... */
              i++;
              //Создаем дубликат предыдущего дубликата
              this["copy"+(i-1)].duplicateMovieClip("copy"+i ,i);
              //Поворачиваем дубликат
              this["copy" + i]._rotation += rotate;
              /* Масштабируем дубликат. Поскольку размеры объекта не
              могут быть отрицательными, используем абсолютное значение
              масштаба на случай передачи отрицательных параметров */
              this["copy" + i]._width *= Math.abs(xScale/100);
              this["copy" + i]._height *= Math.abs(yScale/100);
              }
```

```
        else { /* Если выполнилось требуемое число трансформаций,
        прекращаем их дальнейшее выполнение, удаляя метод обработчика */
                delete obj._parent.onEnterFrame;
                }
        };
}
```

Для автоматической обработки дубликатов необходимо, чтобы они имели унифицированные имена. Можно было бы наложить требование на имя исходного клипа, задаваемого параметром `obj`, однако такой подход ставит внутреннюю реализацию нашей функции в зависимость от внешних условий, что делает ее менее гибкой. В связи с этим мы вначале копируем исходный клип `obj`, задавая первому дубликату унифицированное имя `copy0`, с тем, чтобы при выполнении дальнейшего каскадного копирования мы могли легко формировать имена новых дубликатов, используя значения счетчика `i`. Установка качества прорисовки в значение `"MEDIUM"` связана с тем, что при задании большого количества дубликатов быстродействие может существенно снизиться.

Следующий вызов функции `scaleAndRotate()` приведет к созданию 60 дубликатов клипа `mc`, каждый из которых будет в полтора раза больше предыдущего, и угол его поворота по отношению к предыдущему дубликату будет составлять 4°.

```
scaleAndRotate(mc, 150, 150, 4, 60);
```

Примечание

Следует отметить, что при дублировании клипа, в который был загружен внешний документ (SWF или JPEG), содержимое внешнего документа в составе данного клипа не копируется. Дубликат такого клипа, полученный с помощью метода `duplicateMovieClip()`, включает только содержимое клипа, созданное в его "родном" документе.

Дублирование клипов также может быть выполнено с помощью глобальной функции `duplicateMovieClip()`, имеющей следующий синтаксис:

```
duplicateMovieClip(клип, имя, уровень)
```

Для вызова глобальной функции не требуется применение экземпляра. С помощью глобальной функции `duplicateMovieClip()` можно копировать не только клипы, но и кнопки (в отличие от одноименного метода), передавая ей в качестве первого параметра ссылку на дублируемый экземпляр.

- Параметр *клип* — это ссылка на дублируемый клип (кнопку).
- Параметр *имя* — это строка или выражение, возвращающее строку, задающую имя создаваемому дубликату клипа (кнопки).
- Параметр `уровень` — это число или выражение, возвращающее число, определяющее уровень стека, на который будет помещен создаваемый экземпляр.

При вызове `duplicateMovieClip()` в качестве глобальной функции отсутствует возможность передачи ей объекта инициализации.

Контроль уровней стека

В классе `MovieClip` содержится ряд методов, предоставляющих информацию о порядке размещения клипов в стеке и позволяющих управлять этим порядком.

Метод *getDepth()*

Данный метод позволяет получить номер уровня стека (глубину), на котором расположен вызвавший этот метод экземпляр клипа. Одноименные и идентичные в применении методы также существуют в классах `Button` и `TextField` и могут вызываться экземплярами кнопок и текстовых блоков.

Метод имеет следующий синтаксис:

```
клип.getDepth()
```

Здесь *клип* — это экземпляр клипа, глубину уровня которого требуется получить.

Данный метод не принимает параметров и возвращает целочисленное значение в диапазоне от −16 384 до 1 048 575.

Приведенная ниже функция `getAllDepths()` выводит в панель **Output** информацию о глубине уровней стека объекта `stack`, на которых расположены все экземпляры кнопок и клипов. Используемый здесь оператор `instanceof` возвращает значение `true`, если его левый операнд является экземпляром класса, указанного правым операндом.

```
function getAllDepths(stack:Object):Void {
for (i in stack) {
if (stack[i] instanceof MovieClip || stack[i] instanceof Button) {
trace(typeof stack[i]+": " + i + " --> уровень: " + stack[i].getDepth());
                }
        }
}
```

Следующая строка кода позволит определить порядок расположения всех кнопок и клипов по уровням стека основной монтажной линейки:

```
getAllDepths(_root)
```

Метод *getNextHighestDepth()*

Метод `getNextHighestDepth()` возвращает глубину ближайшего свободного уровня стека, расположенного над всеми занятыми уровнями стека объекта, вызвавшего данный метод. Другими словами, данный метод определяет, на какой ближайший уровень стека необходимо поместить экземпляр, чтобы он оказался на переднем плане, перекрывая все остальное содержимое стека.

Метод `getNextHighestDepth()` не принимает параметров и имеет следующий синтаксис:

```
клип.getNextHighestDepth()
```

Здесь *клип* — это экземпляр клипа, в стеке которого будет выполняться поиск свободного уровня. Данный метод может быть вызван объектом _root для поиска свободного уровня в стеке основной монтажной линейки.

Данный метод не принимает параметров и возвращает целочисленное неотрицательное значение в диапазоне от 0 до 1 048 575 (то есть уровни стека с отрицательными номерами в расчет не принимаются).

Метод getNextHighestDepth() удобно использовать при динамическом создании экземпляров, поскольку он избавляет от необходимости рассчитывать номера свободных уровней. Следующий сценарий при каждом нажатии на клип mc создает его дубликат. Каждый последующий дубликат смещается на 20 пикселов вправо по отношению к предыдущему. В данном случае для генерирования уникальных имен экземпляров так же, как и для указания уровня стека, на который они должны быть помещены, используется метод getNextHighestDepth(). Поскольку данный метод определяет свободный уровень стека вызвавшего его клипа, а дубликаты помещаются не в стек клипа mc, а в стек монтажной линейки, на которой расположен mc, метод getNextHighestDepth() вызывается из объекта, родительского по отношению к mc.

```
mc.onPress = function(){
this.duplicateMovieClip("mc"+this._parent.getNextHighestDepth(),
                        this._parent.getNextHighestDepth(),
                        {_x:this._parent.getNextHighestDepth()*20}
                        )
};
```

Метод *getInstanceAtDepth()*

Данный метод позволяет определить, занят ли уровень стека, указанный в качестве его параметра.

Метод getInstanceAtDepth() имеет следующий синтаксис:

клип.getInstanceAtDepth(*уровень*)

Здесь *клип* — это экземпляр клипа, уровень стека которого будет проверяться. Данный метод может быть вызван объектом _root для проверки того, является ли указанный уровень свободным, или нет.

Параметр *уровень* — это целое число или выражение, возвращающее число в диапазоне от −16 384 до 1 048 575, которое задает уровень стека.

Если уровень стека, заданный параметром *уровень*, содержит экземпляр клипа, кнопки или текстового блока, то метод getInstanceAtDepth() возвращает ссылку на этот экземпляр.

Если уровень стека, заданный параметром *уровень*, содержит иной графический объект, то метод getInstanceAtDepth() возвращает ссылку на сам объект (клип или _root), вызвавший данный метод.

Если уровень стека, заданный параметром *уровень*, не содержит графических объектов, метод getInstanceAtDepth() возвращает значение undefined.

Данный метод может быть использован при динамическом создании экземпляра для проверки того, не является ли уровень, на который он помещается, занятым.

Метод *swapDepths()*

Метод `swapDepths()` позволяет поменять уровни стека для двух клипов или поместить клип на заданный уровень.

Метод `swapDepths()` имеет следующий синтаксис:

клип.`swapDepths(`*уровень*`)`

или

клип.`swapDepths(`*клип*`)`

Здесь *клип* — это экземпляр клипа, который требуется поместить на другой уровень.

В качестве параметра метода `swapDepths()` может передаваться либо номер уровня стека (глубина), на который должен быть помещен *клип*, либо ссылка на клип, с которым *клип* будет меняться уровнями.

Если в качестве параметра передается номер уровня стека, то клип, вызвавший метод `swapDepths()`, помещается на этот уровень, а содержимое данного уровня помещается на уровень стека, на котором находился клип. Уровень может быть задан целочисленным литералом или выражением, возвращающим целое число в диапазоне от −16 384 до 1 048 575[2].

Если в качестве параметра методу `swapDepths()` передается ссылка на экземпляр клипа, то данный клип меняется уровнем стека с клипом, вызвавшим этот метод. Так, если клип `A` находится на уровне `i`, а клип `B` находится на уровне `j`, то при вызове метода `swapDepths()` из экземпляра `A` с параметром, представляющим ссылку на `B`, клип `A` будет помещен на уровень `j`, а клип `B` отправится на освободившийся уровень `i`.

Примечание

Клип, вызывающий метод `swapDepths()`, может быть помещен только на уровень стека, в котором он находится. По этой же причине он может поменяться уровнями с клипом только в том случае, если данный клип расположен в том же самом стеке.

Приведенная ниже функция перемещает переданный ей в качестве параметра клип на самый верхний свободный слой стека, то есть на передний план по отношению ко всем объектам его стека.

```
function bringToFront(obj:MovieClip) {
       obj.swapDepths(obj._parent.getNextHighestDepth());
}
```

[2] При попытке поместить клип на уровень стека, содержащий объект, который не является клипом, возникает ошибка, приводящая к закрытию приложения, в котором запущен проигрыватель Flash Player. Это обстоятельство не позволяет манипулировать размещением графических объектов, не являющихся клипами, в стеке. — *Прим. автора.*

Создание пустого клипа. Метод *createEmptyMovieClip()*

Наряду с созданием экземпляра на основе символа библиотеки и путем копирования существующего экземпляра ActionScript предоставляет возможность создания нового клипа непосредственно в процессе воспроизведения фильма. Характерной особенностью данного подхода является отсутствие необходимости создания символа в рабочей среде. Для динамического создания нового клипа используется метод `createEmptyMovieClip()`, имеющий следующий синтаксис:

клип.createEmptyMovieClip(*имя*, *уровень*)

Здесь *клип* — это клип, в стек которого будет помещен клип, созданный методом `createEmptyMovieClip()`. Данный метод может быть вызван объектом `_root`, при этом создаваемый клип будет помещен в стек основной монтажной линейки.

Параметр *имя* — это имя создаваемого экземпляра. Данный параметр может быть задан как строковый литерал или с помощью выражения, возвращающего строку.

Параметр *уровень* — это номер уровня стека (глубина) объекта, вызвавшего данный метод, на который будет помещен создаваемый клип.

Возвращаемым значением метода `createEmptyMovieClip()` является ссылка на созданный клип.

В результате применения метода `createEmptyMovieClip()` создается новый пустой клип, то есть клип, монтажная линейка которого лишена содержимого. Точка регистрации этого клипа находится в его левом верхнем углу. Назначение данного метода близко к назначению метода `attachMovie()`. Принципиальное отличие заключается в том, что создание нового пустого клипа не требует указания параметров связи, поскольку метод `createEmptyMovieClip()` не просто помещает в стек новый экземпляр элемента библиотеки, а создает новый символ. Метод `createEmptyMovieClip()` исключает возможность передачи в качестве параметра объекта инициализации, как это может быть сделано при работе с методами `attachMovie()` и `duplicateMovieClip()`. На практике пустой клип, создаваемый методом `createEmptyMovieClip()`, может быть использован в следующих целях.

- Для предоставления пространства в своем стеке экземплярам, созданным с помощью методов `attachMovie()`, `duplicateMovieClip()` или `createEmptyMovieClip()`.
- Для предоставления пространства, используемого для размещения графических объектов, созданных средствами программного рисования.
- Для размещения динамически создаваемых текстовых блоков.
- В качестве источника событий, обработка которых может инициировать выполнение определенного блока кода.

Удаление динамически созданных клипов

Для удаления клипов, созданных с помощью методов `attachMovie()`, `duplicateMovieClip()` или `createEmptyMovieClip()`, можно воспользоваться методом `removeMovieClip()`. В результате удаления клипа удаляется все его содержимое,

включая переменные и функции, объявленные на монтажной линейке данного клипа. Метод позволяет удалять только экземпляры клипов, расположенные на уровнях стека с положительными номерами. В связи с этим его нельзя непосредственно применить к экземплярам, созданным в рабочей среде. Тем не менее если созданный в рабочей среде экземпляр будет перемещен на уровень с положительным номером с помощью метода `swapDepths()`[3], его можно будет удалить. Метод `removeMovieClip()` имеет следующий синтаксис:

клип.removeMovieClip()

Здесь *клип* — это динамически созданный экземпляр, подлежащий удалению.

Данный метод не принимает параметров и не возвращает значения.

В листинге 12.15 приведен пример, реализующий простейшую модель игры, основанной на динамическом создании и удалении экземпляров. В библиотеке фильма имеется экземпляр с идентификатором связи "bug", представляющий жука. На монтажной линейке данного символа находится анимация движения жука по круговой траектории. С помощью метода `attachMovie()` в стек основной монтажной линейки помещается количество экземпляров данного клипа, соответствующее значению переменной `maxBugs`. Используя объект инициализации, передаваемый методу `attachMovie()` в качестве четвертого параметра, каждому экземпляру сообщаются случайные значения координат (в пределах рабочей области размером 550×400 пикселов) и случайный угол поворота. Задание случайного угла поворота для каждого экземпляра приводит к тому, что экземпляры движутся в различных направлениях. Для каждого экземпляра определяется метод обработчика события `onPress`, при обнаружении которого данный экземпляр удаляется с помощью метода `removeMovieClip()`. При нажатии на клип также выполняется обновление значения счетчика удаленных экземпляров (переменная `hit`) и вывод информации об оставшемся количестве экземпляров в панель **Output**.

Листинг 12.15. Модель игры на основе динамического создания и удаления экземпляров

```
var maxBugs:Number = 15; // Число жуков
for (i = 0; i < maxBugs; i++) {
// Создание и размещение экземпляров
_root.attachMovie("bug", "bug" + i, i, {
                _x:Math.random()*550, // Случайная координата по X
                _y:Math.random()*400, // Случайная координата по У
                _rotation:Math.random()*360 // Случайный угол поворота
                }
                                );
```

[3] Отметим, что метод `removeMovieClip()` позволяет удалять экземпляры, расположенные в стеке, не выше уровня с номером FFFFF. — *Прим. автора*.

```
// Удаление экземпляров при нажатии
_root["bug" + i].onPress = function() {
       this.removeMovieClip();
       if (hit == undefined) hit = 0; // Счетчик удаленных экземпляров
       hit ++;
       trace ("Осталось жуков: " + (maxBugs - hit))
};
}
```

Помимо метода `removeMovieClip()` существует одноименная глобальная функция, эквивалентная данному методу. Для вызова глобальной функции не требуется создание экземпляра клипа. Функция имеет следующий синтаксис:

`removeMovieClip(`*клип*`)`

Здесь *клип* — это экземпляр клипа, который требуется удалить.

Проверка состояния загрузки

Загрузка любого Flash-фильма из сети не происходит мгновенно, при этом, если его объем достаточно велик, то время загрузки может быть весьма продолжительным. Среда тестирования позволяет оценить реальный процесс загрузки с помощью команды **View>Simulate Download**. Для того чтобы избежать остановок воспроизведения, возникающих при загрузке, и сделать ее процесс более предсказуемым и понятным для зрителя, необходимо снабдить фильм предварительным загрузчиком. Ранее в этой главе уже было описано создание предзагрузчика с использованием свойств `_framesloaded` и `_totalframes`. Помимо этих свойств, класс `MovieClip` располагает еще двумя методами, позволяющими контролировать состояние загрузки, основываясь не на количестве загруженных кадров, а на объеме информации.

Метод *getBytesTotal()*

Метод `getBytesTotal()` возвращает общее количество байтов, занимаемое объектом (клипом или всем фильмом). Данный метод имеет следующий синтаксис:

`объект.getBytesTotal()`

Здесь *объект* — это экземпляр клипа или `_root`. При вызове данного метода с помощью экземпляра клипа он возвращает объем в байтах, занимаемый данным экземпляром. При вызове метода `getBytesTotal()` с помощью идентификатора `_root` он возвращает общее количество байтов, занимаемое всем содержимым основной монтажной линейки.

Примечание

Вызов метода `getBytesTotal()` для определения объема экземпляра имеет смысл, только если в данный экземпляр загружается внешний документ (SWF или JPEG). В этом

случае метод `getBytesTotal()` возвращает объем загруженного в клип внешнего документа. Применение метода `getBytesTotal()` к клипам, не осуществляющим загрузку внешних документов, не дает ожидаемых результатов.

Следует отметить, что метод `getBytesTotal()` возвращает объем фильма без учета используемой компрессии. В связи с этим при использовании сжатия (опция **Compress movie** диалогового окна **Publish Settings**) реальный объем SWF-документа будет меньше объема, возвращаемого методом `getBytesTotal()`.

Метод *getBytesLoaded()*

Метод `getBytesLoaded()` возвращает количество загруженных байтов клипа или основной монтажной линейки. Данный метод имеет следующий синтаксис:

```
объект.getBytesLoaded()
```

Здесь *объект* — это экземпляр клипа или `_root`. При вызове данного метода с помощью экземпляра клипа он определяет, какой объем данного клипа в байтах успел загрузиться на момент вызова этого метода. При вызове метода `getBytesLoaded()` с помощью идентификатора `_root` он возвращает объем содержимого основной монтажной линейки, загруженного на момент вызова этого метода.

Примечание

Вызов метода `getBytesLoaded()` для определения загруженного объема экземпляра имеет смысл, только если в данный экземпляр загружается внешний документ (SWF или JPEG). В этом случае метод `getBytesLoaded()` возвращает загруженный объем внешнего документа. Применение метода `getBytesLoaded()` к клипам, не осуществляющим загрузку внешних документов, не дает ожидаемых результатов.

Создание предварительного загрузчика

Принцип действия предзагрузчика заключается в том, что он не позволяет начинать воспроизведение фильма до полного окончания его загрузки. На протяжении загрузки зритель может видеть сообщение о количестве загруженных байт, которое может быть визуализировано графической полосой состояния, простой анимацией или иным образом.

Листинг 12.16 демонстрирует создание простого внутреннего графического предзагрузчика, который выводит в динамическое текстовое поле состояние загрузки в процентах и отображает процесс загрузки графически в виде изменяющейся в размерах (по горизонтали) полосы состояния. Для создания данного предзагрузчика необходимо выполнить следующие действия.

1. Создать новый ключевой кадр и сделать его первым, поместив перед всеми остальными кадрами фильма, расположенными на основной монтажной линейке. Для этих целей можно использовать отдельную сцену, поместив ее перед всеми остальными сценами фильма.
2. В первом кадре создать динамический текстовый блок и присвоить ему имя экземпляра `counter`, указав его в поле **Instance Name** панели **Properties**

3. В первом кадре создать муви-клип с именем `progressBar`, содержащий изображение полосы состояния загрузки. Точка регистрации клипа должна совпадать с левой гранью полосы состояния. Затем поверх этого клипа нужно нарисовать рамку, совпадающую с его границами (рис. 12.8).

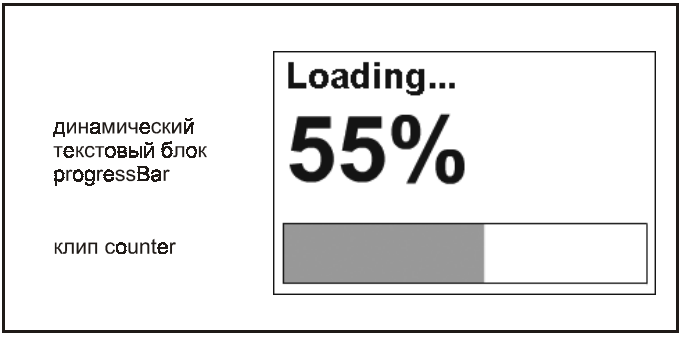


Рис. 12.8. Простой графический предзагрузчик

4. В первый кадр основной монтажной линейки поместить сценарий, приведенный в листинге 12.16.

Листинг 12.16. Графический предзагрузчик

```
//Сохранение размера фильма в байтах в переменной total
var total=_root.getBytesTotal();
_root.stop();//Остановка воспроизведения
progressBar.onEnterFrame=function(){
//Сохранение текущих значений загруженных байтов в переменной loaded
var loaded=_root.getBytesLoaded();
//Если загрузка продолжается...
if (loaded<total) {
//Сохранение текущего состояния загрузки в процентах в переменной
var loadProgress = Math.round(loaded/total*100);
/* Изменение ширины полосы состояния в соответствии с процентом
загруженных байтов */
progressBar._xscale=loadProgress;
/* Вывод состояния загрузки в процентах в динамическое текстовое поле
counter */
counter.text=loadProgress+"%";
}
```

```
/* Когда загрузка окончена - удаляем все данные, обеспечивающие
функционирование предзагрузчика, и переходим к начальному кадру фильма */
else {
       delete this.onEnterFrame;
       delete total;
       _root.gotoAndStop("start");
}
};
```

Следует отметить, что в зависимости от конкретной ситуации после окончания загрузки могут выполняться различные действия. В приведенном примере осуществляется переход в кадр с меткой "start", после которого анимация останавливается. Очевидно, такое поведение требуется, если фильм состоит из одного-единственного кадра основной монтажной линейки или если запуск фильма должен выполняться в результате определенных действий пользователя, выполняемых в начальном кадре. Если же фильм состоит более чем из одного кадра и его воспроизведение должно начинаться автоматически по окончании загрузки, то в последнем предложении будет использоваться не `gotoAndStop()`, а `gotoAndStart()`. В этом случае следует также позаботиться о том, чтобы при зацикливании воспроизведения воспроизводящая головка всякий раз после достижения конца монтажной линейки отправлялась в кадр с меткой "start", а не в первый кадр, содержащий предзагрузчик.

Примечание

Для оценки работы предзагрузчика необходимо в среде тестирования выполнить команду главного меню **View>Simulate Download**.

Приведенный пример описывал создание внутреннего (internal) предзагрузчика, то есть предзагрузчика, который был встроен в загружаемый документ. Существенным ограничением применения такого подхода может являться наличие большого объема информации, загружаемого перед содержимым первого кадра. Как известно, все динамически присоединяемые элементы библиотеки, для которых в окне **Linkage Properties** установлен флажок **Export in first frame**, загружаются еще перед загрузкой содержимого первого кадра, то есть перед тем как выполняется инициализация элементов предзагрузчика. Перед содержимым первого кадра по умолчанию также загружаются классы ActionScript 2.0, хранящиеся во внешних файлах, а также служебные данные, необходимые для инициализации встроенных классов ActionScript. Если совокупный объем таких элементов оказывается довольно большим, то зрителю придется ждать появления предзагрузчика, сидя перед пустым экраном. Кроме того, начальный процент загрузки, выведенный предзагрузчиком, может существенно отличаться от нуля. В подобной ситуации можно прибегнуть к применению внешнего (external) предзагрузчика. Внешний предзагрузчик представляет собой документ-контейнер, в который по мере необходимости загружаются внешние документы, то есть собственно фильмы. Ход загрузки внешних документов можно контролировать либо с помощью описанных в данном разделе методов, либо (более

гибко и с большей степенью контроля) с помощью нового класса `MovieClipLoader`. Описание данного класса и пример создания внешнего предзагрузчика содержатся в *главе 20*. Внешний предзагрузчик для растровых изображений, реализованный с применением методов `getBytesLoaded()` и `getBytesTotal()`, будет описан в данной главе.

Загрузка внешних Flash-документов и изображений

До настоящего момента мы имели дело с ситуацией, когда в проигрыватель Flash Player загружен и воспроизводится только один SWF-документ. Содержимое этого документа представляет собой содержимое основной монтажной линейки исходного FLA-документа, на основе которого был получен данный SWF-файл. При этом для запуска другого SWF-документа нужно либо открыть новое окно автономного проигрывателя, либо загрузить другой документ вместо текущего в то же окно.

ActionScript располагает набором средств, позволяющих осуществлять загрузку внешних SWF-документов и растровых изображений в формате JPEG в стек документов проигрывателя или непосредственно в клип, при этом исходный фильм продолжает воспроизведение в обычном режиме. Это открывает возможности одновременного просмотра нескольких документов и организации переключения между документами без необходимости размещать каждый из них на отдельной HTML-странице.

Примечание

Flash не поддерживает файлы JPEG, использующие режим Progressive, поэтому важно проследить, чтобы все динамически загружаемые изображения были сохранены без использования данного режима.

Читатель уже имеет представление о стеке документов проигрывателя *(см. разд. "Стек документов проигрывателя" данной главы)*. В настоящем разделе речь пойдет о глобальной функции `loadMovieNum()` и методе `loadMovie()`, которые позволяют осуществлять загрузку внешнего файла в формате SWF или JPEG на уровень стека документов или в клип соответственно. Для выполнения этих функций также может быть использован новый класс `MovieClipLoader`, подробное описание которого будет приведено в *главе 20*.

При загрузке внешних SWF-документов из домена, отличного от домена, в котором расположен принимающий их фильм, вступают в силу ограничения, связанные с особенностями применения политики безопасности, используемой в Flash Player 7. Эти особенности будут подробно рассмотрены в *главе 20*.

Загрузка на уровень стека проигрывателя

Для загрузки внешнего файла в формате SWF или JPEG на заданный уровень стека документов может быть использована глобальная функция `loadMovieNum()`. Данная

функция не требует создания экземпляра для своего вызова и имеет следующий синтаксис:

```
loadMovieNum(путь, уровень, метод)
```

Параметр *путь* — это строка или выражение, возвращающее строку, представляющую собой абсолютный или относительный путь к SWF- или JPEG-файлу, который требуется загрузить.

Параметр *уровень* — это уровень стека документов, на который должен быть загружен требуемый файл. Данный параметр может быть задан целочисленным неотрицательным литералом или выражением, возвращающим целое, неотрицательное число. Следует помнить, что при загрузке внешнего документа на уровень стека, уже содержащий документ, загружаемый документ заместит текущее содержимое данного уровня стека.

Параметр *метод* — это строка, указывающая метод передачи переменных во внешний сценарий. Допустимыми значениями являются "GET" и "POST". Данный параметр является необязательным и его назначение аналогично назначению соответствующего параметра функции getURL(), описанной ранее.

Функция loadMovieNum() не имеет возвращаемого значения.

Следующие строки кода загружают файл с именем myMovie.swf на 1-й уровень стека документов, а файл с именем myImage.jpg – на 5-й уровень стека (предполагается, что загружаемые файлы расположены в одном каталоге с файлом, содержащим приведенный сценарий).

```
loadMovieNum("myMovie.swf", 1);
loadMovieNum("myImage.jpg", 5);
```

Следует иметь в виду, что при загрузке внешнего документа он позиционируется таким образом, что его левый верхний угол совпадает с левым верхним углом рабочей области.

Для обращения к документу, загруженному на заданный уровень стека документов, используется идентификатор _level, сопровождаемый номером уровня.

Следующий сценарий при нажатии кнопки but смещает ранее загруженный документ myMovie.swf на 50 пикселов вправо и вниз и делает изображение myImage.jpg наполовину прозрачным:

```
but.onPress = function(){
       _level1._x = _level1._y += 50;
       level5._alpha += 50;
};
```

Идентификатор _root, используемый в документе, загруженном на определенный уровень стека документов проигрывателя, всегда возвращает ссылку на данный уровень. Так, например, если документ загружен на пятый уровень стека документов, то идентификатор _root, используемый в этом документе, возвращает ссылку на _level5. Таким образом, _root, всегда возвращает ссылку на основную монтажную

линейку документа, содержащего этот идентификатор, вне зависимости от того, на каком уровне стека документов находится данный документ.

Организация навигации с использованием динамической загрузки внешних документов

Ранее в этой главе уже было рассмотрено два подхода к реализации навигации: навигация внутри документа с помощью перехода между кадрами (сценами) и навигация между HTML-страницами, предполагающая размещение каждого SWF-документа на отдельной странице. Применение возможностей динамической загрузки внешних документов позволяет реализовать третий подход, предполагающий загрузку внешних документов в документ, исполняющий роль своеобразного контейнера. Преимуществом такого подхода является то, что для его реализации достаточно одной-единственной HTML-страницы. Переход из одного раздела в другой не будет сопровождаться сменой страниц, и зритель может терпеливо ожидать окончания загрузки очередного раздела, видя перед собой привычный интерфейс фильма-контейнера.

При использовании данного подхода в основном фильме (то есть на нулевом уровне стека документов проигрывателя) можно разместить панель навигации и типовые графические элементы интерфейса, исполняющие роль заднего плана, который должен быть виден всегда.

При подготовке документов, представляющих содержимое разделов проекта, которые будут загружены в проигрыватель, необходимо подобрать их размер в соответствии с размером области, отведенной для их отображения в основном фильме.

В листинге 12.17 демонстрируется модель реализации описываемого подхода. Приведенный здесь сценарий находится в кадре монтажной линейки документа, загруженного на нулевой уровень. Этот документ содержит неизменную панель навигации, используемую для загрузки документов разделов, а также рамку, внутрь которой будут загружаться внешние документы. Кнопки в панели навигации имеют имена, состоящие из неизменной части `but` и уникального порядкового номера: `but1`, `but2`, ..., `but5`. Названия загружаемых документов состоят из порядкового номера, точки и расширения: 1.swf, 2.swf, ..., 5.swf. Это позволит легко динамически сформировать строку пути для передачи ее в качестве параметра функции `loadMovieNum()` с тем, чтобы загрузка требуемого файла происходила только при нажатии на соответствующую ему кнопку. Все загружаемые документы помещены в папку main, расположенную в одном каталоге с основным документом. Их размер задан в соответствии с размерами основного документа.

Листинг 12.17. Модель реализации навигации посредством загрузки в стек проигрывателя внешних документов

```
function getSWF(but:Object, url:String, level:Number):Void {
        but.onPress = function() {
                loadMovieNum(url, level);
        };
```

```
}
loadMovieNum("main/1.swf", 1); // Первый раздел загружается автоматически
var maxBut:Number = 5;      //Число разделов
for (i = 1; i <= maxBut; i++) {
        getSWF(this["but" + i], "main/" + i + ".swf", 1);
}
```

Приведем некоторые пояснения. Функция `getSWF()` определяет метод обработчика события `onPress`, который осуществляет загрузку внешнего документа на указанный уровень стека документов при нажатии на заданную кнопку. В качестве параметров данная функция принимает ссылку на кнопку, нажатие которой инициирует загрузку документа, путь к этому документу, заданный относительно основного документа, и уровень стека, на который он должен быть загружен.

Переменная `maxBut` определяет количество внешних файлов и, соответственно, кнопок в панели навигации. Далее в цикле осуществляется многократный вызов функции `getSWF()` для каждой кнопки и передача ей в качестве параметров пути к файлу, соответствующему кнопке.

Особенности загрузки в клип

Помимо загрузки внешних документов на уровень стека документов, существует возможность осуществлять загрузку непосредственно в клип. Для этих целей можно использовать метод `loadMovie()` или одноименную глобальную функцию. Метод `loadMovie()` имеет следующий синтаксис:

клип.loadMovie(*путь*, *метод*)

Здесь *клип* — это экземпляр клипа, в который будет загружен внешний документ.

Параметр *путь* — это строка или выражение, возвращающее строку, представляющую собой абсолютный или относительный путь к SWF- или JPEG-файлу, который требуется загрузить.

Параметр *метод* — это строка, указывающая метод передачи переменных во внешний сценарий, указанный параметром *путь*. Допустимыми значениями являются `"GET"` и `"POST"`. Данный параметр является необязательным, и его назначение аналогично назначению соответствующего параметра функции `getURL()`, описанной ранее.

При использовании метода `loadMovie()` для загрузки внешнего SWF- или JPEG-файла необходимо иметь в виду следующие характерные особенности.

- Внешний документ, будучи загружен в клип, позиционируется таким образом, что его левый верхний угол совмещается с точкой регистрации клипа. В связи с этим необходимо четко представлять, где находится точка регистрации клипа перед выполнением загрузки в него внешнего документа. Скорее всего, точку регистрации клипа, служащего контейнером для внешнего документа, следует поместить в его левый верхний угол.

- Внешний документ, будучи загружен в клип, наследует все трансформации и цветовые эффекты, примененные к этому клипу. Так, если исходный клип был увеличен вдвое и повернут на 30°, то после загрузки в него внешнего документа его содержимое также будет увеличено и повернуто на соответствующий угол.
- При загрузке внешнего документа в клип все исходное содержимое клипа удаляется и замещается содержимым внешнего документа. Это правило относится и к графическому содержимому клипа, и к переменным, и функциям, объявленным на его монтажной линейке. После загрузки в клип внешнего документа все переменные и функции, объявленные на линейке этого клипа (или на линейках его дочерних клипов), оказываются недоступными (удаляются).
- Поскольку переменные и функции, объявленные на линейке клипа, удаляются в результате загрузки в него внешнего документа, методы обработчиков событий, определенные в этом клипе, также перестают функционировать. Тем не менее обработчики, прикрепленные непосредственно к экземпляру клипа (`on()` и `onClipEvent()`), продолжают исправно функционировать. В связи с этим обстоятельством при необходимости обработки событий клипа, осуществляющего загрузку внешнего документа, следует либо помещать его внутрь другого клипа-контейнера, либо прикреплять обработчик событий непосредственно к экземпляру. Прикрепить метод обработчика события к клипу-контейнеру, содержащему внешний документ, можно только после окончания загрузки. Обработчики событий, используемые в загруженном документе, продолжают исправно работать.
- Документ, загруженный в клип, может, в свою очередь, выполнять загрузку внешних документов в стек документов проигрывателя или в свои дочерние клипы.
- После загрузки внешнего документа в клип к этому клипу можно по-прежнему обращаться с помощью имени экземпляра и выполнять с ним различные манипуляции, используя встроенные свойства клипов. Кроме того, к этому клипу можно прикреплять переменные, функции и, в частности, методы обработчиков событий.
- После загрузки внешнего документа в клип доступ к элементам его библиотеки закрывается. Это означает, что после загрузки внешнего документа клип-контейнер не может осуществлять динамическое присоединение экземпляров из своей библиотеки с помощью метода `attachMovie()`. Тем не менее загруженный в клип-контейнер документ может выполнять динамическое присоединение экземпляров из своей собственной библиотеки. После удаления загруженного документа из стека клипа с помощью метода `unloadMovie()` или `MovieClipLoader.unloadClip()` принимавший его клип вновь получает возможность выполнять динамическое присоединение экземпляров из своей библиотеки.
- После загрузки в клип внешнего документа он уже не может динамически дублироваться с помощью метода `duplicateMovieClip()`.

Следующий фрагмент кода при щелчке по клипу `holder` загружает в него внешний документ, расположенный в том же каталоге, что и фильм, содержащий клип `holder`. После этого клип `holder` вместе с загруженным в него фильмом смещаются на 100 пикселов вправо и вверх.

```
holder.onPress = function(){
        this.loadMovie("myExternalMovie.swf");
```

```
        this._x += 100;
        this._y -= 100;
};
```

Для загрузки внешнего документа в клип помимо метода `loadMovie()` можно использовать одноименную глобальную функцию, имеющую следующий синтаксис:

`loadMovie(`*`путь`*`, `*`клип`*`, `*`метод`*`)`

Параметр *`клип`* — это строка, содержащая путь к клипу, или ссылка на клип, в который осуществляется загрузка внешнего документа. Остальные параметры аналогичны соответствующим параметрам метода `loadMovie()`.

Создание интерактивного слайд-шоу

Слайд-шоу представляет собой последовательную смену растровых изображений. Слайд-шоу может быть использовано для демонстрации каталога товаров фирмы-производителя, создания личного портфолио или электронного фотоальбома. Однако следует иметь в виду, что наличие в проекте большого числа растровых изображений может привести к существенному увеличению его объема. В связи с этим наиболее эффективным подходом к созданию слайд-шоу является динамическая загрузка изображений.

В листинге 12.18 представлен сценарий, реализующий интерактивное слайд-шоу, обладающее следующей функциональностью. Наличие кнопок "Вперед" и "Назад" позволяет последовательно просматривать изображения, загружая их из внешнего файла. Все файлы изображений находятся в папке pics, расположенной в том же каталоге, что и файл слайд-шоу. Первое изображение будет загружено автоматически. При переходе к следующему или возврату к предыдущему изображению текущее изображение постепенно становится прозрачным, после чего начинается загрузка нового изображения. В процессе загрузки на сцене появляется текстовое поле, содержащее информацию о состоянии загрузки изображения в процентах (предзагрузчик). После того как изображение полностью загрузится, текстовое поле исчезает и новое изображение постепенно "проявляется", становясь непрозрачным. До тех пор, пока новое изображение полностью не загрузится и не станет полностью непрозрачным, переход к следующему изображению невозможен.

Для реализации данной задачи необходимо выполнить следующие действия.

1. Создать две кнопки с именами *`prevPic`* и *`nextPic`*, по нажатию на которые будет выполняться переход к предыдущему или следующему изображению.
2. Создать муви-клип, исполняющий роль контейнера, в который будет осуществляться загрузка изображений. Наполнение клипа-контейнера не играет роли, поскольку загружаемые изображения заместят содержимое его монтажной линейки. Экземпляр этого клипа нужно назвать `holder`. При позиционировании `holder` на сцене следует иметь в виду, что левый верхний угол загружаемого изображения будет находиться в точке регистрации данного клипа.
3. Разместить на сцене динамическое текстовое поле, присвоив ему имя `display`. В ходе загрузки изображения в этом поле будет выводиться состояние загрузки в

процентах. Следует убедиться, что текстовое поле расположено поверх клипа `holder`.

4. Все изображения, включенные в слайд-шоу, должны быть размещены в папке pics, расположенной в том же каталоге, что и файл слайд-шоу, и названы Image1.jpg, Image2.jpg, Image3.jpg и т. д. Убедитесь в том, что данные изображения не используют режим **Progressive**! Для того чтобы все изображения занимали одинаковую область в клипе `holder`, они должны иметь одинаковый размер (для приведения размеров можно воспользоваться любым растровым редактором, например, Fireworks или Photoshop).
5. Для корректной работы слайд-шоу переменной `maxPic` в начале сценария необходимо присвоить значение, соответствующее количеству изображений (в данном примере их 12). Читатель также может варьировать значение переменной `alphaStep` для регулировки скорости изменения прозрачности изображения.

Листинг 12.18. Интерактивное слайд-шоу с динамической загрузкой изображений

```
//Начальный номер загружаемого изображения
var pic:Number = 1;
//Максимальное число загружаемых изображений
var maxPic:Number = 12;
//Шаг изменения прозрачности изображения
var alphaStep:Number = 10
/*Начальное значение прозрачности клипа, в который осуществляется
загрузка изображений*/
holder._alpha = alphaStep;
/*Переход к следующему изображению только при условии, что текущее
изображение полностью загрузилось и стало непрозрачным*/
//------КНОПКИ "ВПЕРЕД" И "НАЗАД"------
nextPic.onPress = function() {
       if (pic < maxPic && !fadeIn && !fadeOut && loadProgress == 100) {
              pic++;
              fadeOut = true;
       }
};
/*Переход к предыдущему изображению только при условии, что текущее
изображение полностью загрузилось и стало непрозрачным */
prevPic.onPress = function() {
              if (pic>1 && !fadeIn && !fadeOut && loadProgress == 100) {
              pic--;
              fadeOut = true;
       }
};
```

```
//----------------------------------
//Сценарий, выполняемый с частотой равной скорости воспроизведения фильма
_root.onEnterFrame = function() {
//-----------ПРЕДЗАГРУЗЧИК-----------
loadProgress =↳
Math.round(holder.getBytesLoaded()/holder.getBytesTotal()*100);
       if (loadProgress<100) {
/*Если изображение еще не загрузилось, выводим состояние загрузки*/
              display._visible=true;
              display.text=loadProgress+"%";
       }
/*Если изображение загрузилось, отключаем текстовое поле*/
       else display._visible=false;
//-----------КОНЕЦ ПРЕДЗАГРУЗЧИКА-----------
/*Если непрозрачность holder продолжает уменьшаться (то есть _alpha
уменьшается), но остается больше минимального значения, равного alphaStep
уменьшаем значение _alpha с шагом alphaStep */
       if (holder._alpha>=alphaStep && fadeOut) {
              holder._alpha -= alphaStep;
       }
/*Если непрозрачность holder достигла минимального значения (alphaStep),
загружаем в holder требуемое изображение, переключаем флаги*/
       if (holder._alpha<=alphaStep) {
              holder.loadMovie("pics/Image"+pic+".jpg");
              fadeOut = false;//Прекращаем уменьшать непрозрачность
              fadeIn = true;//Начинаем увеличивать непрозрачность
       }
/*Если непрозрачность holder продолжает увеличиваться (то есть _alpha
растет), но остается меньше 100, увеличиваем _alpha с шагом alphaStep
только при условии, что изображение полностью загружено */
       if (holder._alpha<=100 && fadeIn && !fadeOut) {
/*Проверка окончания загрузки изображения, если true, то уменьшаем про-
зрачность*/
              if(loadProgress>=100)holder._alpha += alphaStep;
       }
/*Если ни одно из предыдущих условий не выполняется, то изображение стало
полностью непрозрачным, поэтому прекращаем увеличивать непрозрачность*/
       else {
              fadeIn = false;//Прекращаем увеличивать непрозрачность
       }
};
```

Дополнительные пояснения

Для обеспечения изменения прозрачности клипа, содержащего загруженное изображение, в данном сценарии используются две переменные-флага: *fadeIn* и *fadeOut*. Каждая из них в определенный момент принимает логическое значение `true` или `false`. Так, переменная *fadeIn* принимает значение `true`, если непрозрачность клипа *holder* увеличивается, а переменная *fadeOut* принимает значение `true`, если непрозрачность клипа *holder* уменьшается. Во избежание путаницы следует пояснить, что увеличение непрозрачности означает увеличение значения свойства `_alpha` (при этом объект становится менее прозрачным), соответственно, уменьшение непрозрачности означает уменьшение значения свойства `_alpha` (при этом объект становится более прозрачным). Таким образом, максимальное значение непрозрачности соответствует `_alpha = 100`. В качестве минимального значения непрозрачности в данном сценарии используется значение *alphaStep*. Читатель может изменить это минимальное значение, заменив *alphaStep* в условиях проверочных предложений произвольным фиксированным значением.

Использование абсолютных ссылок во внешних документах

Применение абсолютной адресации (с использованием идентификатора `_root`) в документе, подлежащем загрузке в клип, может привести к проблемам, связанным с конфликтом имен. Действительно, если внешний документ, содержащий обращение к своей основной монтажной линейке в виде идентификатора `_root`, загружается в клип другого документа, то возникает вопрос, относительно какой линейки будут разрешаться ссылки с применением `_root`, используемые в загруженном документе?

Для того чтобы разрешить эту проблему, можно воспользоваться свойством `_lockroot`. Свойство `_lockroot` определяет, на какую монтажную линейку указывает идентификатор `_root`, используемый в документе, загруженном в клип. Данное свойство принимает значения булева типа: `true` или `false`. Это свойство может быть использовано как в самом загружаемом документе, так и клипом, в который осуществляется загрузка внешнего документа.

Если свойство `_lockroot` загружаемого документа установлено в `true`, то идентификатор `_root`, используемый в этом документе, будет всегда возвращать ссылку на его основную монтажную линейку. Для того чтобы идентификатор `_root`, используемый в сценарии SWF-документа, всегда разрешался относительно его основной монтажной линейки, нужно в кадре основной монтажной линейки этого документа поместить следующую строку кода:

```
this._lockroot = true;
```

Если по тем или иным причинам доступ к исходному документу загружаемого файла отсутствует, то можно воспользоваться свойством `_lockroot` самого клипа, в который должен быть загружен внешний документ. В этом случае в документ, содержащий клип, в который будет загружен внешний документ, нужно поместить код следующего вида:

```
клип._lockroot = true;
```

Здесь *клип* — это клип, в который будет загружен внешний документ.

По умолчанию свойство _lockroot принимает значение false. При этом идентификатор _root, используемый в загружаемом документе, возвращает ссылку на основную монтажную линейку принимающего документа, а не на основную линейку загруженного документа.

Рассмотрим принцип действия свойства _lockroot на простом примере. Имеется файл loader.swf, содержащий клип holder, в который требуется загрузить другой внешний файл с названием content.swf.

В файле content.swf на основной монтажной линейке создадим переменную movieStatus и строку кода, выводящую значение этой переменной в панель **Output**:

```
//Документ content.swf
_root.movieStatus = "Загружаемый документ";
trace(_root.movieStatus);
```

В документе loader.swf определим на основной линейке переменную с тем же именем и разместим сценарий, выводящий ее значение в панель **Output**. После этого загрузим документ content.swf в клип holder, расположенный в документе loader.swf.

```
//Документ loader.swf
_root.movieStatus = "Принимающий документ";
trace(_root.movieStatus);
holder.loadMovie("content.swf");
```

В результате загрузки и выполнения сценариев принимающего и загруженного документов в панель **Output** будет выведено следующее:

```
Принимающий документ
Принимающий документ
```

Это произошло из-за того, что после загрузки документа content.swf в клип holder идентификатор _root, используемый в сценарии документа content.swf, возвращает ссылку на монтажную линейку принимающего документа, то есть loader.swf. Для того чтобы _root после загрузки content.swf по-прежнему ссылался на его основную монтажную линейку, можно выполнить одно из следующих действий.

- Поместить в кадр основной монтажной линейки загружаемого документа content.swf следующий код:

```
//Документ content.swf
this._lockroot = true;
_root.movieStatus = "Загружаемый документ";
trace(_root.movieStatus);
```

- Поместить в кадр основной монтажной линейки принимающего документа loader.swf следующий код:

```
//Документ loader.swf
_root.movieStatus = "Принимающий документ";
trace(_root.movieStatus);
holder.loadMovie("content.swf");
holder._lockroot = true;
```

В результате выполненных изменений в панели **Output** будет выведено следующее:

```
Принимающий документ
Загружаемый документ
```

Следует иметь в виду, что для каждого клипа значение свойства `_lockroot` можно установить индивидуально.

Загрузка переменных из внешнего источника

Наряду с загрузкой внешних документов имеется возможность загрузки в Flash-фильм переменных из внешнего источника. В качестве источника переменных может быть использован текстовый документ или серверный сценарий CGI, PHP, ASP или Perl. Переменные могут быть загружены на уровень стека документов или непосредственно в клип.

Для загрузки переменных на уровень документа используется глобальная функция `loadVariablesNum()`, имеющая следующий синтаксис:

`loadVariablesNum(`*путь*`,` *уровень*`,` *метод*`)`

Параметр *путь* — это строка или выражение, возвращающее строку, представляющую собой абсолютный или относительный путь к файлу, служащему источником переменных.

Параметр *уровень* — это уровень стека документа, принимающий переменные. Данный параметр может быть задан целочисленным неотрицательным литералом или выражением, возвращающим целое неотрицательное число. Если указанный уровень уже содержит загружаемые переменные, то их текущие значения будут заменены новыми. При загрузке переменных на уровень стека документов областью видимости этих переменных является данный уровень.

Параметр *метод* — это строка, указывающая метод передачи переменных во внешний серверный сценарий. Допустимыми значениями являются `"GET"` и `"POST"`. Данный параметр является необязательным и его назначение аналогично назначению соответствующего параметра функции `getURL()`, описанной ранее. При указании этого параметра все переменные монтажной линейки, содержащей вызов данной функции, передаются сценарию на сервере.

Текст файла, содержащего загружаемые переменные, должен соответствовать стандартному формату MIME *application/x-www-urlform-encoded*, используемому сценариями CGI. Этот формат предполагает выполнение следующих требований.

- Все переменные передаются в виде пар "имя = значение", разделенных знаком & (амперсанд).
- Пробелы не допускаются. Все пробелы должны быть заменены знаком + (плюс).
- Все символы, не являющиеся цифрой или буквой кодировки ASCII, должны быть заменены шестнадцатеричными Escape-последовательностями вида `%xx`.

Для загрузки переменных в клип используется метод `loadVariables()`, имеющий следующий синтаксис:

клип`.loadVariables(`*путь*`,` *метод*`)`

Здесь *клип* — это клип, в который будут загружены переменные. Если указанный клип уже содержит загружаемые переменные, то их текущие значения будут заменены новыми. Областью видимости загруженных переменных является монтажная линейка данного клипа.

Остальные параметры метода аналогичны соответствующим параметрам глобальной функции `loadVariablesNum()`.

При загрузке переменных в клип для проверки окончания загрузки можно использовать событие `onData`, которое генерируется после того, как все переменные загрузились в клип. Рассмотрим простой пример, иллюстрирующий возможности загрузки переменных из внешнего текстового документа.

Создадим текстовый документ с именем vars.txt и поместим в него следующее содержимое:

```
greeting=Hello&name=Dmitri
```

Затем в Flash-документе создадим следующий сценарий:

```
this.createEmptyMovieClip("loader", this.getNextHighestDepth());
loader.loadVariables("vars.txt");
loader.onData = function() {
        with (this) {
                trace(greeting + " " + name + "!");
        }
};
```

В первой строке приведенного сценария выполняется создание нового пустого клипа, предназначенного для загрузки переменных из файла vars.txt. Затем следует вызов метода `loadVariables()` данного клипа, инициирующий загрузку переменных. Поскольку переменные могут быть использованы только после окончания их загрузки, блок кода, в котором осуществляется обращение к данным переменным, помещен в метод обработчика событий `onData`. Поскольку загруженные переменные находятся в области видимости клипа `loader`, для обращения к ним здесь используется предложение `with`. В результате выполнения данного сценария в панели **Output** выводится следующее:

```
Hello Dmitri!
```

Помимо метода `loadVariables()` для загрузки переменных в клип может быть использована одноименная глобальная функция. Эта функция имеет следующий синтаксис:

```
loadVariables(клип, путь, метод)
```

Параметр *клип* — это ссылка на клип, в который будут загружены переменные. Данный параметр может представлять собой строку или идентификатор. Остальные параметры аналогичны соответствующим параметрам метода `loadVariables()`.

Удаление загруженных документов и содержимого клипов

Внешние документы (SWF или JPEG), загруженные в стек документов проигрывателя или в клип, могут быть удалены или "выгружены" программным образом с помощью глобальных функций `unloadMovieNum()`, `unloadMovie()` и метода `unloadMovie()`. Для этих целей также можно воспользоваться методом `unloadClip()` класса `MovieClipLoader()`, описание которого содержится в *главе 20*.

Следует также помнить, что можно удалить внешний документ, загрузив другой документ на его место.

Удаление документов из стека проигрывателя

Для того чтобы удалить документ, загруженный на определенный уровень стека документов, можно использовать глобальную функцию `unloadMovieNum()`, имеющую следующий синтаксис:

`unloadMovieNum(`*уровень*`)`

Параметр *уровень* — это уровень стека документов, содержимое которого должно быть удалено. Данный параметр задается целочисленным литералом или выражением, возвращающим целое неотрицательное число. При удалении документа из стека проигрывателя удаляются все объявленные в нем переменные и функции. Передача функции `unloadMovieNum()` в качестве параметра числа 0 приведет к удалению из проигрывателя основного документа.

Удаление содержимого клипа

Для того чтобы удалить документ, загруженный в клип, можно воспользоваться методом `unloadMovie()`. В результате применения данного метода клип становится пустым, то есть из него удаляется все содержимое: как графическое, так и не имеющее визуального представления. Этот метод имеет следующий синтаксис:

клип`.unloadMovie()`

Здесь *клип* — это клип, содержащий загруженный документ, который требуется удалить.

Примечание

Применение метода `unloadMovie()` к клипу, созданному в рабочей среде, приведет к удалению его содержимого, однако сам клип удален не будет.

Следует отметить, что в результате применения к клипу метода `unloadMovie()` переменные и методы обработчиков событий этого клипа удаляются вместе со всем его графическим содержимым. Однако обработчики событий, прикрепленные к клипу (`on()` и `onClipEvent()`), продолжают существовать и функционировать.

Преобразование координат и контроль границ клипа

При необходимости контроля координат различных объектов источником непредвиденных (или ожидаемых) трудностей может являться то, что их координаты определены в разных координатных системах. Как известно, монтажная линейка каждого объекта рабочей среды, в частности, клипа, имеет собственную систему координат. Несмотря на то, что направления координатных осей на всех монтажных линейках одинаковы, начало отсчета координат может находиться в произвольной точке. Так, начало координат основной монтажной линейки расположено в левом верхнем углу рабочей области. Начало координат клипа находится в его точке регистрации. Координаты экземпляра, задаваемые с помощью свойств `_x` и `_y`, всегда задаются относительно начала координат родительского объекта, то есть объекта, на монтажной линейке которого находится данный экземпляр. В связи с этим координаты клипа, расположенного на основной монтажной линейке, и координаты клипа, расположенного внутри другого клипа, не могут быть непосредственно сопоставлены, так как они определены в разных системах координат. Для того чтобы получить возможность сопоставления координат, заданных в разных координатных системах, необходимо совместить их точки отсчета или выразить координаты каждого объекта в общих глобальных координатах. В качестве глобальных координат можно воспользоваться координатами основной монтажной линейки, то есть `_root`. В классе `MovieClip` определены два метода, позволяющие выражать локальные координаты объекта в координатах основной монтажной линейки и, наоборот, преобразовывать глобальные координаты в локальные координаты заданного клипа.

Преобразование в глобальные координаты

Метод `localToGlobal()` позволяет выразить координаты точки, заданные в системе координат клипа, в глобальные координаты основной монтажной линейки. Данный метод имеет следующий синтаксис:

клип.localToGlobal(*объект*)

Здесь *клип* — это клип, в системе координат которого заданы исходные локальные координаты точки.

Параметр *объект* — это ссылка на объект, являющийся экземпляром класса `Object`, который содержит координаты точки, заданные в системе координат клипа *клип*. Координаты точки хранятся в объекте *объект* в виде значений двух его свойств `x` и `y`. Эти значения могут быть заданы числовым литералом или выражением, возвращающим число.

Метод `localToGlobal()` не имеет возвращаемого значения.

Для применения метода необходимо создать объект `Object`, представляющий точку, координаты которой должны быть преобразованы из локальной системы координат клипа *клип* в глобальную систему координат основной монтажной линейки (`_root`).

Для создания такого объекта можно воспользоваться конструктором или литералом объекта:

```
//Применение конструктора
var point:Object = new Object();
//Применение литерала
var point:Object = {};
```

Любая из приведенных строк кода создает экземпляр пустого (без свойств) объекта `Object` и сохраняет ссылку на него в переменной `point`.

После создания объекта необходимо добавить в него два свойства с именами `x` и `y`. В качестве значений этих свойств необходимо задать координаты точки, измеренные относительно точки регистрации клипа *клип*.

```
point.x = 50;
point.y = -25;
```

С помощью литерала объекта можно совместить создание объекта и добавление в него свойств в одном предложении:

```
var point:Object = {x:50, y:-25};
```

При вызове метода `localToGlobal()` с помощью клипа *клип* значения координат точки, помещенные в свойства `x` и `y` объекта `point`, преобразуются из системы координат клипа в систему координат `_root`. Таким образом, данный метод как бы проецирует координаты точки, заданные относительно локальной системы координат клипа, в точку на основной монтажной линейке.

Рассмотрим простой пример. На основной монтажной линейке имеется клип с именем `holder`. Клип `holder` на своей монтажной линейке содержит вложенный клип с именем `mc`. Необходимо определить координаты вложенного клипа `mc`, измеренные в системе координат основной монтажной линейки.

Координаты клипа ассоциируются с местоположением его точки регистрации. Таким образом, для ответа на поставленный вопрос нам необходимо узнать, в какую точку `_root` проецируется точка регистрации клипа `mc`. Точка регистрации клипа всегда является точкой отсчета в его системе координат. Поэтому наша задача сводится к определению глобальных координат точки, координаты которой в системе координат `mc` равны нулю.

```
var point:Object = new Object();
point.x = 0;
point.y = 0;
holder.mc.localToGlobal(point);
```

Другим путем решения этой задачи может служить преобразование координат клипа `mc` из системы координат клипа `holder` (относительно начала координат которого регистрируются координаты `mc`) в систему координат `_root`.

```
var point:Object = new Object();
point.x = holder.mc._x;
```

```
point.y = holder.mc._y;
holder.localToGlobal(point);
```

Оба фрагмента кода приводят к тому, что свойства `x` и `y` объекта `point`, принимают значения, соответствующие координатам клипа `mc` в системе координат `_root`.

Преобразование в глобальные координаты может потребоваться при необходимости выполнения взаимосвязанных перемещений клипов, находящихся на разных монтажных линейках. В этом случае координаты этих клипов для сопоставимости могут быть преобразованы в глобальные координаты, после чего с ними можно выполнять требуемые преобразования. Для того чтобы преобразования, выполненные с глобальными координатами клипов, могли отразиться на их размещении, необходимо выполнить обратное преобразование — из глобальных координат в локальные координаты клипа.

Преобразование в локальные координаты

Метод `globalToLocal()` позволяет выразить координаты точки, заданные в координатах основной монтажной линейки, в локальные координаты клипа. Данный метод имеет сходный синтаксис с методом `localToGlobal()`:

клип`.globalToLocal()(`*объект*`)`

Здесь *клип* — это клип, в систему координат которого должны быть преобразованы исходные глобальные координаты точки.

Параметр *объект* — это ссылка на объект, который содержит координаты точки, заданные в системе координат основной монтажной линейки. Этот параметр аналогичен соответствующему параметру предыдущего метода `localToGlobal()`.

Метод `globalToLocal()` не имеет возвращаемого значения.

Принцип использования данного метода аналогичен принципу использования метода `localToGlobal()`.

Рассмотрим небольшой пример. На основной монтажной линейке имеется клип с именем `mc`. Следующий сценарий при нажатии левой кнопки мыши выводит в панель **Output** текущие координаты курсора относительно точки регистрации клипа `mc`.

```
var point:Object = new Object();
mc.onMouseDown = function(){
       point.x = _xmouse;
       point.y = _ymouse;
       mc.globalToLocal(point);
       trace("X: "+ point.x +"; Y: "+point.y);
};
```

Контроль границ клипа

Изученные ранее средства контроля координат и размеров клипа не всегда оказываются достаточными для управления положением объектов, поскольку они не дают представления о местоположении их точек регистрации. Для того чтобы получить информацию о координатах ограничивающей рамки клипа, можно воспользоваться методом `getBounds()`. Под ограничивающей рамкой клипа понимается прямоугольная область минимальной площади, внутри которой лежат все входящие в его состав графические объекты. Следует иметь в виду, что в разных кадрах монтажной линейки клипа могут быть расположены разные объекты, соответственно во время анимации размеры ограничивающей рамки могут меняться.

Метод `getBounds()` позволяет получить координаты левого верхнего и правого нижнего угла ограничивающей рамки клипа. Этот метод имеет следующий синтаксис:

клип.getBounds(*система координат*)

Здесь *клип* — это клип, координаты ограничивающей рамки которого требуется получить.

Параметр *система координат* — это объект (клип или `_root`), в системе координат которого будут указаны возвращаемые значения координат ограничивающей рамки.

Метод `getBounds()` возвращает объект типа `Object` с четырьмя свойствами:

- `xMin`, `yMin` — координаты левого верхнего угла ограничивающей рамки клипа *клип*, указанные в системе координат объекта, переданного в качестве параметра;
- `xMax`, `yMax` — координаты правого нижнего угла ограничивающей рамки клипа *клип*, указанные в системе координат объекта, переданного в качестве параметра.

Ссылка на объект, возвращаемый методом `getBounds()`, должна быть сохранена в переменной с тем, чтобы информация о координатах границ клипа могла быть использована в дальнейшем.

Ниже иллюстрируется применение метода `getBounds()` для определения координат ограничивающей рамки клипа `mc` в системе координат `_root`.

```
var bounds:Object;
bounds = mc.getBounds(_root);
trace("Левый верхний: "+bounds.xMin + "; " + bounds.yMin);
trace("Правый нижний: "+bounds.xMax + "; " + bounds.yMax);
```

Равномерное распределение клипов

В качестве иллюстрации применения методов преобразования координат и контроля границ клипов рассмотрим пример создания функции, выполняющей распределение клипов с равными промежутками между их границами. Назовем нашу функцию `spaceEvenlyAt()` и потребуем, чтобы эта функция обеспечивала следующую функциональность.

- Возможность установки равных промежутков заданного размера как по горизонтали, так и по вертикали. При этом функция должна позволять выполнять распределение только по горизонтали, только по вертикали или и по горизонтали, и

по вертикали одновременно. В последнем случае величины промежутков между клипами по горизонтали и по вертикали могут отличаться.

- При выравнивании по горизонтали клип, левая грань которого находится левее всех остальных, остается на своем месте, а все остальные клипы размещаются в порядке удаленности их левых граней от левой грани первого клипа. Положение каждого последующего клипа рассчитывается, исходя из положения предыдущего.
- При выравнивании по вертикали клип, верхняя грань которого находится выше всех остальных, остается на своем месте, а все остальные клипы размещаются в порядке удаленности их верхних граней от верхней грани первого клипа. Положение каждого последующего клипа рассчитывается, исходя из положения предыдущего.
- Возможность выполнения выравнивания для любого количества клипов вне зависимости от их размеров.
- Возможность выполнения выравнивания клипов вне зависимости от местоположения их точек регистрации.
- Возможность выполнения выравнивания вложенных клипов, расположенных на разных монтажных линейках.

Функция `spaceEvenlyAt()` будет принимать три параметра:

- параметр `objs` — это массив, состоящий из ссылок на клипы, которые нужно распределить. Элементами этого массива могут быть ссылки на клипы, расположенные на разных монтажных линейках;
- параметры `xSpace` и `ySpace` — это числа, задающие промежутки между клипами по горизонтали и по вертикали соответственно. Если выравнивание должно быть выполнено только по горизонтали или только по вертикали, то в качестве значения лишнего параметра нужно указать `null`.

Функция `spaceEvenlyAt()` приведена в листинге 12.19. Поясним основные этапы используемого алгоритма.

Поскольку функция `spaceEvenlyAt()` должна выполнять одну из двух или последовательно две различные операции (распределение по горизонтали и по вертикали), она фактически состоит из двух похожих фрагментов кода. Данное пояснение и комментарии в тексте листинга описывают процесс распределения клипов по горизонтали. Распределение по вертикали выполняется аналогично. Вначале выполняется распределение по горизонтали.

Массив клипов, переданный в качестве первого параметра, сортируется по местоположению левой (верхней) грани. Для выполнения этой сортировки внутри функции содержится объявление двух функций сравнения, которые передаются методу `Array.sort()` для реализации сортировки. В результате сортировки первым элементом в массиве (с индексом 0) становится клип, который будет оставаться на своем исходном месте.

Поскольку в общем случае размеры и местоположение клипов могут отличаться, положение каждого последующего клипа будет рассчитываться, исходя из координат предыдущего. Координаты первого клипа известны, так как он остается на месте.

Для определения координат следующего клипа необходимо знать местоположение его точки регистрации, местоположение ближайшей к нему границы предыдущего клипа и расстояние, которое должно быть установлено между клипами. Введем обозначения:

X_i — горизонтальная координата i-го клипа;

$X_{i\ (прав)}$ — горизонтальная координата правой границы i-го клипа;

$X_{i\ (лев)}$ — горизонтальная координата левой границы i-го клипа;

Δ — расстояние между клипами;

Исходя из введенных обозначений, горизонтальная координата i-го клипа вычисляется следующим образом:

$$X_i = X_{i-1} + X_{i-1\ (прав)} + \Delta + X_{i\ (лев)}.$$

При вычислении вертикальной координаты используется аналогичный подход.

Поскольку в процессе выполнения вычислений данной функцией обновления экрана не происходит (то есть объекты не двигаются с места до тех пор, пока выполнение функции не закончится), мы не можем непосредственно получить координаты границ предыдущего клипа при вычислении координат следующего. Поэтому приходится использовать рекурсивный подход.

После окончания вычислений все созданные в процессе выполнения функции и ставшие ненужными переменные удаляются с помощью оператора `delete`, чтобы освободить место в памяти. В данном случае мы не могли использовать локальные переменные, которые удаляются автоматически, так как (почти) все переменные создавались динамически.

Листинг 12.19. Установка равных промежутков между клипами с произвольными размерами и расположением точки регистрации, находящимися на разных монтажных линейках

```
function spaceEvenlyAt(objs:Array, xSpace:Number, ySpace:Number) {
// Функция сравнения для сортировки клипов по местоположению левой грани
       function compareX(a, b) {
              var boundsA = a.getBounds(_root);
              var boundsB = b.getBounds(_root);
              return boundsA.xMin - boundsB.xMin;
       }
/* Функция сравнения для сортировки клипов по местоположению верхней
грани */
       function compareY(a, b) {
              var boundsA = a.getBounds(_root);
              var boundsB = b.getBounds(_root);
              return boundsA.yMin - boundsB.yMin;
       }
```

```
//--------------РАЗМЕЩЕНИЕ КЛИПОВ ПО ГОРИЗОНТАЛИ-----------------------
if (xSpace != null) {
/* Сортируем клипы в массиве в соответствии с расположением их левых
граней */
objs.sort(compareX);
//Для каждого клипа из массива objs...
for (var i = 0; i < objs.length; i++) {
        // Сохраняем границы клипа
        this["bounds" + i] = objs[i].getBounds(_root);
        // Сохраняем исходные координаты точки регистрации клипа
        this["centerPoint" + i] = {x:0, y:0};
        /* Преобразуем координаты точки регистрации из локальной системы
        координат клипа в систему координат _root */
        objs[i].localToGlobal(this["centerPoint" + i]);
        /* Сохраняем расстояние от точки регистрации клипа до его левой
        границы */
        this["left" + i] = this["centerPoint" + i].x - this["bounds" +↓
        i].xMin;
        /* Сохраняем расстояние от точки регистрации клипа до его правой
        границы */
        this["right" + i] = -this["centerPoint" + i].x + this["bounds" +↓
        i].xMax;
        // Объект для хранения конечных координат клипа
        this["coord" + i] = new Object();
        /* Для горизонтального размещения координата Y не требуется,
        однако метод globalToLocal () требует наличия у объекта свойств
        x и y, поэтому присваиваем y значение NaN */
        this["coord" + i].y = NaN;
        // Для первого клипа в массиве...
        if (i == 0) {
                // Местоположение не меняется
                coord0.x = centerPoint0.x;
                /* Локальная переменная для хранения координаты i-го клипа,
                измеренной в системе координат _root. Значение данной
                переменной изменяется на каждой итерации цикла (поскольку
                координаты, хранящиеся в this["coord" + i] будут на каждой
                итерации преобразовываться из локальных в глобальные, а для
                вычисления положения следующего клипа нам потребуется
                значение глобальных координат предыдущего клипа, вводим
                временную переменную tempX) */
                var tempX = this["coord" + i].x;
        }
```

```
        // Для всех клипов массива, кроме первого...
        if (i > 0) {
                /* Вычисляем координаты следующего клипа (в системе
                координат _root) на основании параметров предыдущего клипа
                и заданного промежутка между клипами */
                this["coord" + i].x = tempX + this["right" + (i - 1)] +
                xSpace + this["left" + i];
                /* Обновляем значение переменной tempX, с тем, чтобы
                использовать ее на следующей итерации для вычисления
                положения следующего клипа*/
                tempX = this["coord" + i].x;
                /* Преобразуем полученные координаты i-го клипа из системы
                координат _root в систему координат объекта, на линейке
                которого он находится */
                objs[i]._parent.globalToLocal(this["coord" + i]);
                // Позиционируем i-ый клип
                objs[i]._x = this["coord" + i].x;
                // Удаляем переменные, которые уже более не нужны
                delete this["bounds" + i];
                delete this["centerPoint" + i];
                delete this["left" + (i - 1)];
                delete this["right" + (i - 1)];
                delete this["coord" + i];
        }
}
// А эти переменные становятся ненужными только теперь!
delete this["left" + (objs.length - 1)];
delete this["right" + (objs.length - 1)];
}
//---------------РАЗМЕЩЕНИЕ КЛИПОВ ПО ВЕРТИКАЛИ-------------------------
if (ySpace != null) {
objs.sort(compareY);
for (var i = 0; i < objs.length; i++) {
        this["bounds" + i] = objs[i].getBounds(_root);
        this["centerPoint" + i] = {x:0, y:0};
        objs[i].localToGlobal(this["centerPoint" + i]);
        this["up" + i] = this["centerPoint" + i].y - this["bounds" +
i].yMin;
        this["down" + i] = -this["centerPoint" + i].y + this["bounds" +
i].yMax;
        this["coord" + i] = {};
```

```
        this["coord" + i].x = NaN;
        if (i == 0) {
                coord0.y = centerPoint0.y;
                var tempY = this["coord" + i].y;
        }
        if (i > 0) {
                this["coord" + i].y = tempY + this["down" + (i - 1)] +
        ySpace + this["up" + i];
                tempY = this["coord" + i].y;
                objs[i]._parent.globalToLocal(this["coord" + i]);
                objs[i]._y = this["coord" + i].y;
                delete this["bounds" + i];
                delete this["centerPoint" + i];
                delete this["up" + (i - 1)];
                delete this["down" + (i - 1)];
                delete this["coord" + i];
        }
}
}
delete this["up" + (objs.length - 1)];
delete this["down" + (objs.length - 1)];
/* Переменные, используемые при размещении клипов по горизонтали и по
вертикали. Теперь их можно удалить! */
delete bounds0;
delete centerPoint0;
delete coord0;
}
```

Перетаскивание объектов

Достаточно часто при реализации схемы интерактивного взаимодействия перед разработчиком возникает необходимость обеспечения возможности перемещения объектов с помощью курсора мыши. ActionScript предоставляет несколько способов решения данной задачи. Один из них заключается в том, что при перемещении курсора мыши его координаты, хранящиеся в свойствах `_xmouse` и `_ymouse`, присваиваются свойствам `_x` и `_y` объекта, в результате чего объект следует за курсором. Этот подход уже был описан ранее в листинге 12.6. Другой подход к реализации перетаскивания объектов предоставляется методом `startDrag()`.

Перетаскивание с помощью метода *startDrag()*

Вызов метода `startDrag()` приводит к тому, что вызвавший его клип начинает следовать за курсором мыши, причем его перетаскивание продолжается до тех пор, пока не будет вызван метод `stopDrag()`, прекращающий перетаскивание, или пока другой клип не вызовет метод `startDrag()`. Метод `startDrag()` имеет следующий синтаксис:

клип.startDrag(*центр*, *лево*, *верх*, *право*, *низ*)

Здесь *клип* — это имя перетаскиваемого экземпляра клипа.

Параметр *центр* принимает значение `true` или `false`, определяющее местоположение точки регистрации клипа во время перетаскивания. Если параметр *центр* принимает значение `true`, то при перетаскивании курсор мыши находится в точке регистрации перетаскиваемого клипа, то есть в его начале координат. Если параметр *центр* принимает значение `false`, то при перетаскивании курсор по отношению к клипу находится в той же точке, где он находился в момент вызова метода `startDrag()`. Данный параметр является необязательным. Если при вызове метода `startDrag()` этот параметр не указан, его значение принимается равным `false`.

Параметры *лево*, *верх*, *право*, *низ* задаются числовыми литералами или выражениями, возвращающими числа, и позволяют задать прямоугольную область, в пределах которой выполняется перетаскивание. Параметры *лево*, *верх* задают координаты левого верхнего угла области по горизонтали и вертикали соответственно. Параметры *право*, *низ* задают координаты правого нижнего угла области по горизонтали и вертикали соответственно. Эти параметры являются необязательными. Если область перетаскивания не задана, то перетаскивание выполняется внутри всей области окна проигрывателя Flash Player.

Следует иметь в виду, что с помощью метода `startDrag()`можно одновременно перетаскивать только один объект. Если во время перетаскивания одного объекта метод `startDrag()` вызывается для другого объекта, то перетаскивание первого объекта прекращается.

Следующий блок кода при нажатии на клип `mc` заставляет его следовать за курсором в пределах заданной области. Во время перетаскивания курсор находится в той точке клипа, где он находился во время нажатия на него.

```
mc.onPress = function(){
       this.startDrag(false, 100, 100, 300, 300);
};
```

С помощью метода `startDrag()` можно перетаскивать только клипы или все содержимое основной монтажной линейки, если данный метод вызывается объектом `_root`. Для перетаскивания других объектов (кнопок и текстовых блоков) можно воспользоваться глобальной функцией `startDrag()`, имеющей следующий синтаксис:

startDrag(*объект*, *центр*, *лево*, *верх*, *право*, *низ*)

Параметр *объект* — это целевой объект (клип, кнопка или текстовый блок), который требуется перетаскивать. Назначение остальных параметров глобальной функ-

ции `startDrag()` аналогично назначению соответствующих параметров одноименного метода.

Следует иметь в виду, что при установке параметра *центр* глобальной функции `startDrag()` в значение `true` перетаскиваемый объект позиционируется таким образом, что координаты курсора мыши совпадают с его точкой регистрации. Точка регистрации динамического (пользовательского) текстового блока находится в его левом верхнем углу.

Прекращение перетаскивания

Для прекращения перетаскивания клипа, инициированного методом `startDrag()`, используется метод `stopDrag()`. Данный метод не принимает параметров и имеет следующий синтаксис:

клип.`stopDrag()`

Здесь *клип* — это перетаскиваемый с помощью метода `startDrag()` клип, перетаскивание которого требуется прекратить.

Следующий блок кода при отпускании кнопки мыши над перетаскиваемым клипом `mc` прекращает его перетаскивание.

```
mc.onRelease = function(){
       this.stopDrag();
};
```

Для прекращения перетаскивания объектов, инициированного методом `startDrag()`, также может быть использована глобальная функция `stopDrag()`. Эта функция не принимает параметров и просто прекращает текущий процесс перетаскивания для любого перетаскиваемого в момент ее вызова объекта.

Приведенный ранее блок кода можно заменить эквивалентным ему кодом с применением функции `stopDrag()`:

```
mc.onRelease = function(){
       stopDrag();
};
```

Следует помнить, что вызов метода или глобальной функции `startDrag()` во время перетаскивания объекта прекращает текущее перетаскивание и инициирует новое перетаскивание другого объекта.

Для получения ссылки на объект, над которым находился перетаскиваемый клип в момент прекращения перетаскивания, можно воспользоваться свойством `_droptarget`, описанным в начале *данной главы*.

В листинге 12.20 приведена функция `dropToBase()`, контролирующая попадание перетаскиваемого клипа в область, занятую другим клипом. Такая функция может иметь применение при создании разнообразных игр, тестов и т. д., предполагающих перемещение объектов в определенную область.

Функция `dropToBase()` принимает 4 параметра:

- параметр `clip` — это ссылка на клип, который при нажатии должен начинать перетаскиваться, а при отпускании прекращать следовать за курсором;
- параметры `initX`, `initY` — это начальные координаты клипа `clip`;
- параметр `base` — это клип, над которым должен находиться перетаскиваемый клип.

Если при отпускании кнопки мыши клип `clip` находится над клипом `base`, то клип `clip` центрируется относительно клипа `base`, то есть "прилипает" к нему. В противном случае клип `clip` возвращается в исходное положение, заданное параметрами `initX` и `initY`. Следует отметить, что под нахождением одного клипа над другим подразумевается следующая совокупность условий:

- ограничивающие рамки клипов пересекаются;
- в момент прекращения перетаскивания курсор мыши находится над графическим содержимым клипа `base`.

Проверку условия совмещения клипов можно реализовать с помощью свойства `_droptarget`, описанного ранее в этой главе, или вручную, используя метод `hitTest()`, который будет описан далее в этой главе. В листинге проверка совмещения клипов реализуется с помощью свойства `_droptarget`.

Листинг 12.20. Контроль попадания объекта в заданную область с применением свойства `_droptarget`

```
function dropToBase(clip:MovieClip, initX, initY, base:MovieClip) {
        // Исходное положение клипа
        clip._x = initX;
        clip._y = initY;
        // При нажатии на клип...
        clip.onPress = function() {
                // Делаем клип, по которому щелкнули курсором самым верхним
                в стопке
                this.swapDepths(this._parent.getNextHighestDepth());
                // Включаем перетаскивание
                this.startDrag();
                };
        // При отпускании кнопки мыши...
        clip.onRelease = function() {
                // Прекращаем перетаскивание
                this.stopDrag();
                /* Если клип "сброшен" в требуемую область (клип base),
                он центрируется по этой области */
                if (eval(this._droptarget) == base) {
```

```
                this._x = base._x;
                this._y = base._y;
            }
            // Если клип "сброшен" мимо цели, возвращаем его в исходное
            положение
            else {
                this._x = initX;
                this._y = initY;
            }
        };
}
```

Альтернативный подход к решению этой задачи будет продемонстрирован в листинге 12.22.

Контроль наложения клипов

Ранее в этой главе уже были рассмотрены некоторые возможности, позволяющие осуществлять контроль наложения или пересечения клипов. Приведем их обобщенный перечень.

- Свойства `_x`, `_y`, `_width`, `_height` позволяют контролировать взаимное размещение клипов, основываясь на их координатах и размерах. Ограничениями применения этих средств является невозможность учесть местоположение графического содержимого относительно точки регистрации, а также форму границ клипа.
- Свойство `_droptarget` позволяет обнаружить пересечение клипов, однако действует только в сочетании с методом `startDrag()`.
- Метод `getBounds()` позволяет проверить факт пересечения ограничивающих рамок клипов.

Наряду с перечисленными возможностями существует еще один метод, позволяющий обнаружить пересечение ограничивающих рамок клипов или попадание точки с заданными координатами в область, занимаемую клипом. Этот метод называется `hitTest()`, он имеет две разные формы синтаксиса.

Контроль пересечения ограничивающих рамок клипов

Первый вариант использования метода позволяет проверить факт пересечения ограничивающих рамок двух клипов. Подобную процедуру читатель может выполнить самостоятельно с помощью метода `getBounds()`. Синтаксис метода `hitTest()`, используемого для решения данной задачи, приведен ниже.

клип.hitTest(*целевой клип*)

Здесь *клип* — это экземпляр клипа, пересечение ограничивающей рамки которого требуется проверить.

Параметр *целевой клип* — это ссылка на клип, пересечение с ограничивающей рамкой которого требуется проверить.

Если ограничивающая рамка клипа *клип* пересекает ограничивающую рамку клипа *целевой клип*, то метод `hitTest()` возвращает значение `true`. Если ограничивающие рамки клипов не пересекаются, метод `hitTest()` возвращает значение `false`. Следует отметить, что порядок размещения клипов в стеке не влияет на результат, возвращаемый методом `hitTest()`, то есть проверяется только геометрическое пересечение границ объектов.

Следующая строка кода проверяет, пересекаются ли ограничивающие рамки клипов `mc` и `target`.

```
mc.hitTest(target);
```

В листинге 12.21 приводится сценарий, реализующий модель простой игры с мячом и битой. Мяч, представлен клипом `ball`, который движется внутри ограниченного прямоугольного поля, отскакивая от боковых стенок и от верхней грани. Внизу поля расположена бита, представленная клипом `bat`. Бита управляется с помощью стрелок клавиатуры (клавиши <Left> и <Right>) и двигается влево и вправо соответственно. Цель игры заключается в том, чтобы, перемещая биту, не давать мячу коснуться нижней границы поля. На сцене находится экземпляр динамического текстового блока с именем `scoreTxt`, предназначенный для вывода счета игры. Для хранения счета используется переменная `score`, значение которой увеличивается после каждого отскока мяча от биты.

Игра начинается при щелчке по бите. Смещение мяча реализуется путем периодической модификации значений их свойств `_x` и `_y` на величины `speedX` и `speedY` соответственно. Такой подход позволяет легко реализовать отскок от стенки. Если координата мяча выходит за пределы границ поля (в данном примере используется стандартный размер рабочей области 550×400 пикселов), мы просто меняем знак переменной `speedX` или `speedY` на противоположный, в результате чего мяч продолжает движение в противоположном направлении.

Столкновение мяча с битой контролируется с помощью метода `hitTest()`. Поскольку мяч имеет круглую форму, а бита — это прямоугольник, для обнаружения столкновения достаточно проверить факт пересечения ограничивающих рамок мяча и биты.

Управление битой осуществляется с помощью методов класса `Key`, которые будут подробно описаны в *главе 17*.

Листинг 12.21. Модель простой игры с мячом и битой

```
//При щелчке по бите...
bat.onPress = function() {
        //--------------------ИНИЦИАЛИЗАЦИЯ-------------------------
        ball._x = ball._y =0; // Начальные координаты мяча
```

```
var score:Number = 0; // Начальный счет
var radius = .5 * ball._width; // Радиус мяча
scoreTxt.text = score; // Вывод начального счета в текстовое поле
//---------------ОБЕСПЕЧЕНИЕ ДВИЖЕНИЯ МЯЧА-------------------
ball.onEnterFrame = function() {
        /* Инициализация переменных, задающих шаг смещения мяча по
        горизонтали и по вертикали */
        if (speedX == undefined && speedY == undefined) {
                speedX = speedY =15;
        }
        // Смещение мяча по горизонтали и по вертикали
        ball._x += speedX;
        ball._y += speedY;
        /* Контроль столкновения со стенками, кроме нижней (размер
        поля 550 x 400). При каждом столкновении со стенкой меняем
        направление движения мяча, изменяя знак шага горизонтальной
        или вертикальной составляющей движения */
        if (ball._x > 550 - radius) {
                ball._x = 550 - radius;
                speedX = -speedX;
        }
        if (ball._x < radius) {
                ball._x = radius;
                speedX = -speedX;
        }
        if (ball._y < radius) {
                ball._y = radius;
                speedY = -speedY;
        }
        //При достижении нижней стенки...
        if (ball._y > 400 - radius) {
                // Обнуляем счет и выводим его в текстовое поле
                scoreTxt.text = 0;
                // Помещаем мяч в исходные координаты
                ball._x = ball._y =0;
                // Прекращаем движение мяча
                delete this.onEnterFrame;
        }
        // При столкновении мяча с битой...
        if (ball.hitTest(bat)) {
```

```
                        // Меняем направление движения (отскок)
                        speedY = -speedY;
                        // Увеличиваем счет игры на единицу
                        scoreTxt.text = ++score;
                }
        };
};
// -------------ОБЕСПЕЧЕНИЕ УПРАВЛЕНИЯ БИТОЙ---------------------
bat.onEnterFrame = function() {
        // Движение биты вправо
        if (Key.isDown(Key.RIGHT)) {
                if (bat._x >= 550 - .5 * bat._width) {
                        bat._x = 550 - .5 * bat._width;
                }
                else {
                        this._x += 25;
                }
        }
        // Движение биты влево
        if (Key.isDown(Key.LEFT)) {
                if (bat._x <= .5 * bat._width) {
                        bat._x = .5 * bat._width;
                }
                else {
                        this._x -= 25;
                }
        }
};
```

Контроль попадания точки в заданную область

Второй вариант использования метода `hitTest()`позволяет проверить, попадает ли указанная точка в область, занятую клипом. Для этих целей метод `hitTest()` вызывается с использованием следующего синтаксиса:

```
клип.hitTest(x, y, форма/рамка)
```

Здесь `клип` — это экземпляр клипа, попадание точки в область которого требуется проверить.

Параметры `x` и `y` — это координаты точки, попадание которой в область клипа требуется проверить. Координаты точки должны быть заданы в глобальных координатах, то есть в системе координат `_root`.

Параметр форма/рамка принимает значение true или false и определяет, что понимается под областью, попадание точки в которую требуется проверить. Если параметр форма/рамка принимает значение true, то метод hitTest() проверяет попадание точки в область, совпадающую с формой клипа. Если параметр форма/рамка принимает значение false, то метод hitTest() проверяет попадание точки внутрь ограничивающей рамки клипа клип.

Если точка попадает в область клипа, метод hitTest() возвращает значение true, в противном случае — false.

Следующая строка кода проверяет, лежит ли точка с координатами (100; 150) внутри фактической формы клипа mc.

```
mc.hitTest(100, 150, true)
```

С помощью метода можно определить, попадает ли курсор мыши в фактическую область, занимаемую клипом (а не просто внутрь его ограничивающей рамки). Для этого в качестве первых двух параметров ему нужно передать координаты курсора мыши, указанные относительно _root:

```
mc.hitTest(_root._xmouse, root._ymouse, true);
```

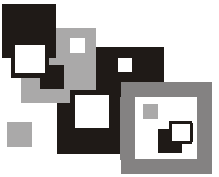
Вооружившись этими сведениями, мы можем использовать алгоритм перетаскивания, реализованный в листинге 12.6, и метод mc.hitTest() для контроля попадания перетаскиваемого клипа в заданную область. В листинге 12.20 эта задача была решена с помощью методов startDrag(), stopDrag() и свойства _droptarget. В листинге 12.22 предлагается более изящный способ ее решения.

Листинг 12.22. Контроль попадания объекта в заданную область с применением метода hitTest()

```
function dropToBase(clip:MovieClip, lockCenter:Boolean, initX, initY,
base:MovieClip) {
clip.onPress = function() { // При нажатии...
       // Делаем клип, по которому щелкнули самым верхним в стопке
       // курсором
       this.swapDepths(this._parent.getNextHighestDepth());
       // Сохраняем расстояние от курсора в момент нажатия до точки
       // регистрации клипа
       var xDist = this._x - this._parent._xmouse;
       var yDist = this._y - this._parent._ymouse;
       this.onMouseMove = function() { // При перемещении курсора...
              // Перетаскивание
              if(lockCenter){/* Если при перетаскивании курсор должен
              находиться в точке регистрации клипа... */
              this._x = this._parent._xmouse;
              this._y = this._parent._ymouse;
              }
```

```
            else {/* Если при перетаскивании курсор должен находиться
            в точке, где он был в момент нажатия на клип или, если
            параметр lockCenter не указан...*/
            this._x = this._parent._xmouse+xDist;
            this._y = this._parent._ymouse+yDist;
            }
            updateAfterEvent(); //Обновление экрана
        };
};
clip.onRelease = function() { // При отпускании кнопки мыши...
        /* Если клип "сброшен" в требуемую область (клип base),
        он центрируется по этой области */
        if(base.hitTest(_root._xmouse, _root._ymouse, true)){
        this._x = base._x;
        this._y = base._y;
        }
        /* Если клип "сброшен" мимо цели, возвращаем его в исходное
        положение */
        else {
        this._x = initX;
        this._y = initY;
        }
        delete this.onMouseMove; // Прекращаем перетаскивание
};
}
```

Глава 13



Динамическое рисование

– Никогда не рисуйте то, что можно скопировать.
– Никогда не копируйте то, что можно обвести.
– Никогда не обводите то, что можно вырезать и наклеить.

Правила рисования Вуда

До настоящего момента графическое содержимое всех рассматриваемых нами примеров создавалось в рабочей среде. ActionScript располагает программным интерфейсом, позволяющим создавать графику динамически, то есть непосредственно в процессе воспроизведения фильма. Такая возможность может оказаться весьма полезной, поскольку графика, создаваемая динамически, в значительно меньшей степени влияет на объем конечного документа, чем графика, создаваемая в рабочей среде. Кроме того, с динамически созданным изображением можно выполнять различные преобразования, видоизменяющие его форму и цвет элементов (контура и заливки), в то время как форма и цвет элементов клипа или кнопки, созданных в рабочей среде, не могут быть подвергнуты изменениям.

Динамическое рисование реализуется с помощью набора методов, предоставляемых классом `MovieClip`. Средства программного интерфейса рисования ActionScript достаточно скромны, поскольку встроенные методы позволяют реализовать только основные базовые возможности. Создание более сложных и изощренных инструментов рисования на основе этих средств возложено на разработчика. Программный инструментарий рисования позволяет выполнять следующие функции.

- Рисование прямых линий заданной толщины, цвета и прозрачности.
- Заполнение контуров сплошным цветом или градиентом. Градиент может быть линейным или радиальным.
- Рисование кривых, представляющих собой квадратичные кривые Безье.
- Удаление динамически созданной графики.
- Использование одного клипа в качестве маски для другого.

Несмотря на то, что данный перечень оказался весьма кратким, перечисленные возможности позволяют справиться с большим количеством возникающих на практике задач.

Динамическое рисование может выполняться на основной монтажной линейке (`_root`) или на монтажной линейке клипа. Следует иметь в виду, что все графиче-

ские объекты, созданные средствами динамического рисования, всегда оказываются на самом нижнем уровне стека содержащего их объекта. Это означает, что графическое содержимое, созданное в рабочей среде, в пределах одного стека всегда располагается поверх графического содержимого, созданного динамически.

Обычно динамическое рисование выполняется в специально предназначенном для этого клипе. Такой клип может быть создан в рабочей среде, хотя во многих случаях удобнее для его создания использовать метод `createEmptyMovieClip()`, создающий пустой клип, который играет роль контейнера для динамически созданной графики. В дальнейшем с помощью метода `swapDepths()` можно легко изменять расположение клипа с динамической графикой относительно других объектов стека, перемещая его на передний или, наоборот, на задний план. Кроме того, этот клип можно легко смещать и трансформировать, тем самым изменяя его содержимое.

Рисование прямолинейных сегментов

Рисование прямой линии — это наиболее простая процедура. Для того чтобы нарисовать прямолинейный сегмент, необходимо задать параметры линии, указать координаты начальной точки рисования и соединить начальную точку с конечной. Методы динамического рисования действуют так, как будто рисующее перо движется, не отрываясь от бумаги. При этом конец предыдущего сегмента совпадает с началом следующего, образуя ломаную линию (или кривую, если речь идет о рисовании кривых). Для того чтобы нарисовать новый сегмент, не связанный с предыдущим, необходимо задать новую точку начала рисования.

Задание начальной позиции рисования

Для задания точки, от которой будет выполняться рисование, используется метод `moveTo()`, имеющий следующий синтаксис:

клип.`moveTo(`*x*`, `*y*`)`

Здесь *клип* — это объект, на монтажной линейке которого выполняется рисование. В качестве такого объекта может использоваться `_root` или экземпляр клипа.

Параметры *x* и *y* — это координаты точки, с которой будет начато рисование. Координаты задаются в системе координат объекта, на монтажной линейке которого находится *клип*.

Метод как бы перемещает перо в позицию, с которой должно начаться рисование.

Создадим клип с именем `holder`, в котором будет выполняться рисование, и определим точку начала рисования:

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
holder.moveTo(100, 100);
```

Определение атрибутов контура

Для того чтобы нарисовать линию, необходимо перед началом рисования вызвать метод `lineStyle()` и передать ему в качестве параметров атрибуты линии. Метод `lineStyle()` имеет следующий синтаксис:

клип.lineStyle(*толщина*, *цвет*, *прозрачность*)

Здесь *клип* — это объект, на монтажной линейке которого выполняется рисование.

Параметр *толщина* определяет толщину контура. Допустимые значения данного параметра лежат в диапазоне от 0 до 255. Нулевое значение толщины соответствует линии со стилем **hairline**. Такая линия не оказывает влияния на размеры объекта. Если передаваемое значение данного параметра меньше 0, то линия будет иметь нулевую толщину. Если передаваемое значение превышает 255, используется максимальная толщина, то есть значение 255. Если данный параметр опущен, линия не рисуется.

Параметр *цвет* задается шестнадцатеричным числом и определяет цвет линии. Если данный параметр не указан, по умолчанию используется черный цвет.

Параметр *прозрачность* позволяет задать степень прозрачности линии. Допустимые значения лежат в диапазоне от 0 (полная прозрачность) до 100 (полная непрозрачность). Если передаваемое значение данного параметра меньше 0, то линия будет полностью прозрачна. Если передаваемое значение превышает 100, линия будет полностью непрозрачна. Если данный параметр опущен, по умолчанию используется значение 100.

Продолжим начатый пример и определим следующие параметры для линии: толщина — 5, цвет — красный, прозрачность — 70.

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
holder.lineStyle(5, 0xFF0000, 70);
holder.moveTo(100, 100);
```

Примечание

Метод `lineStyle()` вызывается только в том случае, если линия должна быть нарисована. Если требуется нарисовать форму, состоящую только из заливки, то данный метод не используется.

Параметры линии, определенные методом `lineStyle()`, будут применены ко всем контурам, создаваемым в вызвавшем данный метод объекте, до тех пор, пока не произойдет повторный вызов метода `lineStyle()` с другими параметрами. Метод `lineStyle()` может вызываться многократно на протяжении процесса рисования. Это позволит задать разные цвета для различных сегментов контура.

Соединение точек прямолинейным отрезком

После того как задана точка начала рисования и определены настройки линии, можно приступить к рисованию. Для создания прямолинейного отрезка используется метод `lineTo()`. Данный метод соединяет текущую точку начала рисования со

следующей точкой с помощью прямой линии. Метод `lineTo()` имеет следующий синтаксис:

клип.lineTo(*x*, *y*)

Здесь *клип* — это объект, на монтажной линейке которого выполняется рисование.

Параметры *x* и *y* — это координаты точки, с которой соединяется точка начала рисования.

В результате применения метода `lineTo()` начальная точка рисования смещается в точку с координатами, заданными параметрами данного метода. Это означает, что при следующем вызове метода `lineTo()` рисование будет выполняться из данной точки. Для того чтобы рисование выполнялось из другой произвольной точки, нужно воспользоваться методом `moveTo()`.

Завершим начатый пример, нарисовав квадрат размером 100×100 со сторонами разного цвета и разной толщины. Поскольку применение методов динамического рисования требует многократного обращения к одному и тому же объекту, для более компактной записи удобнее воспользоваться предложением `with`.

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
with (holder) {
       lineStyle(5, 0xFF0000, 70);
       moveTo(100, 100);
       lineTo(200, 100);
       lineStyle(3, 0x00FF00, 100);
       lineTo(200, 200);
       lineStyle(2, 0x0000FF, 100);
       lineTo(100, 200);
       lineStyle(4, 0xFF00FF, 100);
       lineTo(100, 100);
}
```

В данном примере перед рисованием очередной стороны квадрата вызывается метод `lineStyle()`, задающий новые параметры линии.

Для того чтобы нарисовать замкнутый контур, необходимо провести линию из конечной точки последнего сегмента в начальную точку, из которой было начато рисование.

Удаление графики

Для удаления динамически созданной графики используется метод `clear()`, имеющий следующий синтаксис:

клип.clear()

Здесь *клип* — это объект, в котором удаляется динамически созданная графика.

Метод *клип* удаляет все графические объекты, созданные с помощью методов динамического рисования из вызвавшего его объекта. Действие данного метода не распространяется на графические объекты, созданные в рабочей среде.

Следующий фрагмент кода удаляет из клипа `holder` всю динамически созданную графику при нажатии на кнопку `clearBut`:

```
clearBut.onPress = function(){
holder.clear();
};
```

Создание инструмента "карандаш"

Используя метод `lineTo()`, можно с помощью прямых линий рисовать разнообразные ломаные линии, получая контуры более сложной формы. Если длины прямолинейных отрезков, составляющих контур, будут достаточно малы, это позволит передать кривизну, то есть аппроксимировать кривую, используя прямолинейные отрезки. В качестве иллюстрации этих возможностей создадим простой рисующий инструмент "карандаш", снабдив его следующей функциональностью.

- При перемещении курсора с нажатой кнопкой мыши выполняется рисование непрерывной линии, состоящей из прямолинейных отрезков.
- При отпускании кнопки мыши рисование линии завершается. Для рисования новой линии достаточно нажать кнопку мыши в любой точке окна проигрывателя.
- Имеется возможность выбора цвета линии.
- Двойной щелчок в любой точке окна проигрывателя удаляет все нарисованные линии.

Рассмотрим процесс создания программного кода, реализующего данную функциональность.

Введем переменную `color` для хранения текущего цвета, используемого для рисования. По умолчанию рисование будет выполняться с применением белого цвета.

```
var color = 0xFFFFFF;
```

Поскольку рисование должно выполняться только при нажатой кнопке мыши, введем переменную-флаг `penUp`, принимающую значение `true`, если рисование не выполняется (перо поднято), и значение `false`, если рисование выполняется (перо опущено). Переменная `penUp` будет принимать значение `true` только при отпускании кнопки мыши. При нажатии кнопки мыши устанавливаем флаг в значение `false` и в качестве точки начала рисования задаем точку, в которой в момент нажатия находится курсор мыши. Здесь же задаем параметры линии, так чтобы каждый новый сегмент мог иметь требуемый цвет.

```
this.onMouseDown = function() {
        penUp = false; // Перо опущено
        this.lineStyle(5, color);
        this.moveTo(_xmouse, _ymouse);
};
```

При перемещении курсора, если кнопка мыши остается нажатой, соединяем точку, соответствующую текущей позиции рисования, с точкой, в которой в данный момент находится курсор. Чем большую частоту генерирования события `onMouseMove` способен обеспечить компьютер, тем короче будут отрезки, соединяющие соседние точки, и, соответственно, тем выше будет точность аппроксимации криволинейной траектории перемещения курсора.

```
this.onMouseMove = function() {
        if (!penUp) { // Если перо опущено...
                this.lineTo(_xmouse, _ymouse);
                updateAfterEvent();
        }
};
```

При отпускании кнопки мыши переключаем флаг `penUp` в состояние `true`, в результате чего блок кода условного предложения в теле метода обработчика `onMouseMove` перестает выполняться, и рисование прекращается.

```
this.onMouseUp = function() {
        penUp = true; // Перо поднято
};
```

В приведенном примере пользователь получает возможность выбора одного из 8 цветов. Для этих целей в рабочей среде были созданы 8 разноцветных клипов с именами `mc0`, `mc1`, ..., `mc8`. При щелчке на клипе в качестве текущего цвета рисования должен устанавливаться один из восьми цветов. Для того чтобы поставить каждому клипу в соответствие один из восьми цветов, используем предложение `switch`. Здесь выполняется проверка последнего символа (цифры) в имени клипа, и поскольку все клипы имеют уникальные имена, при нажатии на каждый из них переменная `color` принимает значение цвета, соответствующее данному клипу.

В *главе 18* будет описан подход, позволяющий динамически сгенерировать цветовую палитру, используя возможности класса `Color`.

```
for (i = 0; i < 8; i++) {
        this["mc" + i].onPress = function() {
                switch (Number(this._name.substr(-1))) {
                case 0 :
                        color = 0xffffff; //Белый
                        break;
                case 1 :
                        color = 0xff0000; //Красный
                        break;
                case 2 :
                        color = 0xff6600; //Оранжевый
                        break;
```

```
                case 3 :
                        color = 0xffff00; // желтый
                        break;
                case 4 :
                        color = 0x00ff00; // Зеленый
                        break;
                case 5 :
                        color = 0x00ffff; // Голубой
                        break;
                case 6 :
                        color = 0x0000ff; // Синий
                        break;
                case 7 :
                        color = 0xff00ff; // Фиолетовый
                        break;
                }
        };
}
```

При двойном щелчке левой кнопкой мыши вызываем метод `clear()` для удаления всех нарисованных линий. В данном случае действие метода `clear()` распространяется на объект, содержащий объявление функции обработчика `onDoubleClick`, то есть на объект, на монтажной линейке которого находится весь описываемый сценарий. Если данный сценарий находится на основной линейке фильма, можно было бы воспользоваться явной ссылкой `_root.clear()`. Использовать идентификатор `this` здесь нельзя, поскольку он возвращает ссылку на объект-приемник `myListener`, а не на объект, в котором выполняется рисование.

Возможность трансляции пользовательских событий, и в частности, обнаружение двойного щелчка мыши, обсуждалась в *главе 11*, соответствующий сценарий с комментариями может быть найден в листинге 11.3.

```
myListener.onDoubleClick = function() {
        clear();
};
```

Описанный сценарий полностью приведен в листинге 13.1.

Примечание

Следует иметь в виду, что динамическое рисование большого количества объектов существенно снижает быстродействие проигрывателя. Это становится хорошо заметно при использовании нашего карандаша. Линии, нарисованные вначале, обладают значительно большей гладкостью, поскольку содержат больше опорных точек, в то время как по мере накопления точек проигрыватель уже не может отрабатывать событие `onMouseMove` с прежней частотой.

Листинг 13.1. Создание инструмента "карандаш"

```
var color = 0xFFFFFF; // Цвет по умолчанию
// Начинаем рисование при нажатии левой кнопки мыши
this.onMouseDown = function() {
       penUp = false; // Перо опущено
       this.lineStyle(5, color);
       this.moveTo(_xmouse, _ymouse);
};
// Если кнопка мыши нажата, при перемещении курсора мыши продолжаем
рисование
this.onMouseMove = function() {
       if (!penUp) { // Если перо опущено...
              this.lineTo(_xmouse, _ymouse);
              updateAfterEvent();
       }
};
// Прекращаем рисование при отпускании кнопки мыши
this.onMouseUp = function() {
       penUp = true; // Перо поднято
};
//-------------ВЫБОР ЦВЕТА ИЗ ПАЛИТРЫ-----------------
for (i = 0; i < 8; i++) {
       this["mc" + i].onPress = function() {
              switch (Number(this._name.substr(-1))) {
              case 0 :
                     color = 0xffffff; //Белый
                     break;
              case 1 :
                     color = 0xff0000; //Красный
                     break;
              case 2 :
                     color = 0xff6600; //Оранжевый
                     break;
              case 3 :
                     color = 0xffff00; // желтый
                     break;
              case 4 :
                     color = 0x00ff00; // Зеленый
                     break;
```

```
                case 5 :
                        color = 0x00ffff; // Голубой
                        break;
                case 6 :
                        color = 0x0000ff; // Синий
                        break;
                case 7 :
                        color = 0xff00ff; // Фиолетовый
                        break;
                }
        };
}
// ---ОБНАРУЖЕНИЕ ДВОЙНОГО ЩЕЛЧКА---------
var delay:Number = 300;
var myBroadcaster:Object = new Object();
AsBroadcaster.initialize(myBroadcaster);
var myListener:Object = new Object();
myBroadcaster.addListener(myListener);
Mouse.addListener(myListener);
myListener.onMouseDown = function() {
        clickInterval = getTimer() - clickTime;
        clickTime = getTimer();
        if (clickInterval < delay) {
                myBroadcaster.broadcastMessage("onDoubleClick");
                clickTime = -delay;
        }
};
//-----ОЧИСТКА ЭКРАНА ПРИ ДВОЙНОМ ЩЕЛЧКЕ------------
myListener.onDoubleClick = function() {
        clear();
};
```

Применение сплошной заливки

Для того чтобы залить динамически созданный контур сплошным цветом, используется метод `beginFill()`. Данный метод имеет следующий синтаксис:

клип`.beginFill(`*цвет*`, `*прозрачность*`)`

Здесь *клип* — это объект, на монтажной линейке которого выполняется рисование.

Параметр *цвет* задается шестнадцатеричным числом и определяет цвет заливки. Если данный параметр не указан или принимает значение `undefined`, заливка отсутствует.

Параметр *прозрачность* позволяет задать степень прозрачности заливки. Допустимые значения лежат в диапазоне от 0 (полная прозрачность) до 100 (полная непрозрачность). Если передаваемое значение данного параметра меньше 0, то заливка будет полностью прозрачна. Если передаваемое значение превышает 100, заливка будет полностью непрозрачна. Если данный параметр опущен, по умолчанию используется значение 100.

Следующий фрагмент кода рисует квадрат с синей обводкой и красной полупрозрачной заливкой.

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
with (holder) {
       lineStyle(1, 0x0000FF);
       beginFill(0xFF0000, 50);
       moveTo(100, 100); // 1-я точка
       lineTo(200, 100); // 2-я точка
       lineTo(200, 200); // 3-я точка
       lineTo(100, 200); // 4-я точка
       lineTo(100, 100); // Замыкаем контур, соединяя 4-ю и 1-ю точки
}
```

Для получения только заливки без обводки метод `lineStyle()` не используется.

Действие метода `beginFill()` с заданными параметрами заливки после его вызова распространяется на все создаваемые в вызвавшем данный метод объекте фигуры до тех пор, пока не будет выполнено одно из следующих действий.

- Повторно вызван метод `beginFill()` с новыми параметрами.
- Вызван метод `beginGradientFill()`, осуществляющий заливку градиентом.
- Вызван метод `endFill()`, прекращающий действие метода `beginFill()` или `beginGradientFill()`.

Так, если в предыдущем примере добавить строки кода, создающие новый квадрат, он будет иметь такую же обводку и заливку, как и первый.

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
with (holder) {
       lineStyle(1, 0x0000FF);
       beginFill(0xFF0000, 50);
       moveTo(100, 100);//Рисуем первый красный квадрат
       lineTo(200, 100);
       lineTo(200, 200);
       lineTo(100, 200);
```

```
        lineTo(100, 100);
        moveTo(200, 200);//Рисуем второй красный квадрат
        lineTo(300, 200);
        lineTo(300, 300);
        lineTo(200, 300);
        lineTo(200, 200);
}
```

Для того чтобы отменить использование заливки и прекратить действие метода `beginFill()`, можно воспользоваться методом `endFill()`, имеющим следующий синтаксис:

клип`.endFill()`

Здесь *клип* — это объект, в котором требуется прекратить действие метода `beginFill()`.

Данный метод осуществляет заливку контура, созданного на момент его вызова, и отменяет дальнейшее использование заливки. При заполнении контура в результате вызова метода `endFill()` используются параметры заливки, заданные предшествующим вызовом метода `beginFill()`. Так, если после рисования первого квадрата в предыдущем примере вызвать метод `endFill()`, то второй квадрат будет нарисован без заливки.

Если при вызове метода `endFill()` контур оказывается незамкнутым, Flash автоматически выполняет замыкание контура, соединяя прямолинейным отрезком точку текущей позиции рисования с начальной точкой, заданной последним вызовом метода `moveTo()`.

Следует отметить, что если после вызова метода `beginFill()` в результате применения методов рисования текущая точка рисования совпадает с начальной точкой контура, заданной методом `moveTo()`, происходит замыкание контура, и он заполняется заливкой. Если же после вызова метода `beginFill()` контур остается незамкнутым, то есть его начальная и конечная точки не совпадают, то результат может оказаться неожиданным. В некоторых ситуациях контур замыкается автоматически, однако в других случаях можно получить достаточно неожиданный и малопригодный для практического использования результат. В связи с этим необходимо запомнить следующие правила, выполнение которых позволит избежать получения некорректных результатов.

- Если после вызова метода `beginFill()` в результате применения методов рисования конечная точка контура совмещается с начальной, контур замыкается и заполняется заливкой с заданными параметрами. При этом дальнейшее применение метода `endFill()` не является обязательным. Применение метода `endFill()` в этом случае может быть необходимо для продолжения рисования контура без заливки.
- Если после вызова метода `beginFill()` конечная точка контура не совмещается с его начальной точкой, необходимо после окончания рисования вызвать метод `endFill()`, выполняющий замыкание и заливку контура.

Следующий пример иллюстрирует применение метода `endFill()`. После вызова метода `beginFill()` рисуется ломаная незамкнутая линия. В результате вызова метода `endFill()` контур замыкается, образуя квадрат, который заполняется заливкой с параметрами, указанными при вызове `beginFill()`.

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
with (holder) {
       lineStyle(1, 0x0000FF);
       beginFill(0xFF0000, 50);
       moveTo(100, 100);// 1-я точка
       lineTo(200, 100);// 2-я точка
       lineTo(200, 200);// 3-я точка
       lineTo(100, 200);// 4-я точка
       endFill(); // Замыкаем контур, соединяя 4-ю и 1-ю точки
}
```

Рисование многоугольников и звезд

В качестве иллюстрации создания контуров, состоящих из прямолинейных сегментов, и применения к ним сплошной заливки рассмотрим создание функции, позволяющей рисовать правильные выпуклые многоугольники с произвольным количеством сторон и звезды с произвольным количеством лучей. Результат действия нашей функции будет схож с результатом применения инструмента рабочей среды **Polystar**, реализованного с помощью программного интерфейса JavaScript Flash, поддерживаемого в Flash MX 2004.

Рассмотрим принцип, используемый при построении выпуклых равносторонних многоугольников и звезд. Вначале рассмотрим построение выпуклого многоугольника, который представляет собой вырожденную звезду.

Любой выпуклый равносторонний многоугольник может быть вписан в окружность, центр которой совпадает с его центром. Все вершины такого многоугольника лежат на общей окружности, причем каждая вершина отстоит от соседней на один и тот же угол. Радианную меру этого угла можно легко найти, разделив число 2π на количество вершин многоугольника. Так, в пятиугольнике каждая вершина отстоит от соседней на 0,25π (72°). Соответственно, для того чтобы нарисовать многоугольник, необходимо найти координаты его вершин, расположенных на окружности, и соединить их прямыми линиями.

Координаты точек, соответствующих вершинам многоугольника, можно найти из следующих уравнений (рис. 13.1).

$$X_i = X_{\text{центр}} + R \cdot \cos(i \cdot \varphi);$$
$$Y_i = Y_{\text{центр}} + R \cdot \sin(i \cdot \varphi).$$
$$(i = 0, 1, ..., N - 1)$$

Здесь:

- X_i и Y_i — это координаты i-ой вершины;
- $X_{\text{центр}}$ и $Y_{\text{центр}}$ — это координаты центра многоугольника (окружности);

- R — радиус окружности, то есть расстояние от центра до вершины многоугольника;
- φ — угол между соседними вершинами многоугольника;
- N — это количество вершин многоугольника.

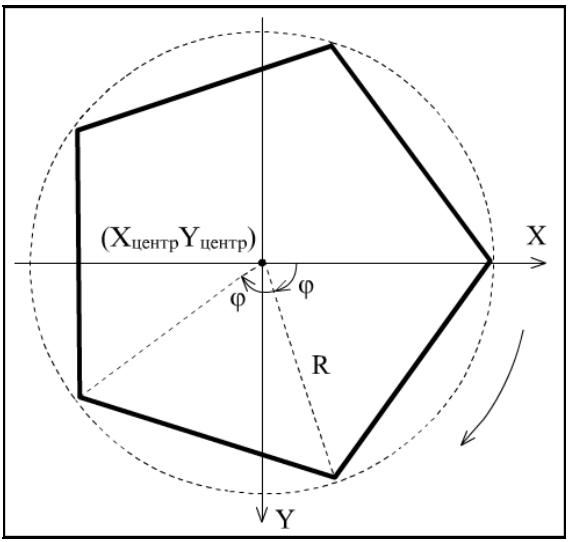


Рис. 13.1. Построение правильного выпуклого многоугольника

Принцип построения звезды лишь немногим сложнее, чем построение многоугольника. Как видно из рис. 13.2, вершины звезды лежат на двух окружностях с общим центром. Вершины лучей звезды лежат на внешней окружности. Остальные вершины лежат на внутренней окружности. Углы между соседними вершинами лучей звезды и ее точками, лежащими на внутренней окружности, равны и определяются точно так же, как и в случае с многоугольником. При этом следует отметить, что точка звезды, лежащая на внешней окружности, смещена по отношению к ближайшей точке звезды, расположенной на внутренней окружности, на угол, равный половине угла между соседними точками звезды, расположенными на одной окружности.

Соотношение радиусов внутренней и внешней окружностей определяет степень остроконечности звезды. Если эти радиусы равны, получаем выпуклый равносторонний многоугольник, число сторон которого равно $2N$, где N — количество лучей звезды.

В соответствии со сказанным, для построения звезды с заданным количеством лучей нам потребуется вдвое больше точек, чем для построения многоугольника с соответствующим количеством сторон. Половина этих точек расположена на внешней окружности, другая половина — на внутренней. Координаты точек, соответствующих вершинам звезды, можно найти из следующих уравнений.

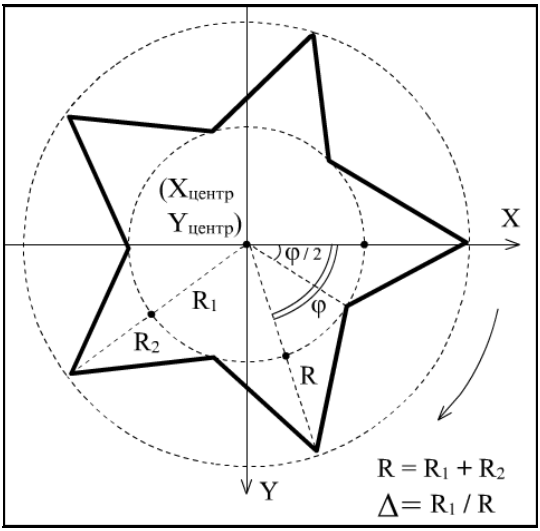


Рис. 13.2. Построение звезды

Координаты точек звезды, расположенных на внешней окружности:

$$X_i = X_{\text{центр}} + R \cdot \cos(i \cdot \varphi);$$
$$Y_i = Y_{\text{центр}} + R \cdot \sin(i \cdot \varphi).$$
$$(i = 0, 2, ..., 2(N - 1))$$

Координаты точек звезды, расположенных на внутренней окружности:

$$X_i = X_{\text{центр}} + R \cdot \Delta \cos(\varphi \cdot (i - 0.5));$$
$$Y_i = Y_{\text{центр}} + R \cdot \Delta \sin(\varphi \cdot (i - 0.5)).$$
$$(i = 1, 3, ..., 2N\text{-}1)$$

Здесь:

- R — радиус внешней окружности, то есть расстояние от центра до вершин лучей звезды;
- Δ — это отношение радиуса внутренней окружности к радиусу внешней окружности;
- N — это количество лучей звезды.

После нахождения координат точек их нужно просто соединить прямолинейными сегментами. Следует обратить внимание, что при создании как многоугольника, так и звезды необходимо в конце рисования замкнуть контур, то есть соединить последнюю вершину фигуры с ее первой вершиной.

В листинге 13.2 представлена функция `polyStar()`, позволяющая рисовать многоугольники и звезды с произвольным количеством сторон и лучей соответственно. Приведенная функция обладает следующей функциональностью.

- Позволяет задать тип фигуры, которую требуется нарисовать: многоугольник или звезда.

- Позволяет задать размер фигуры. Под размером понимается диаметр окружности, в которую вписана результирующая фигура.
- Позволяет определить наличие, толщину и цвет обводки, а также наличие и цвет заливки.
- Размещает результирующую фигуру в динамически созданном клипе. Клип создается на монтажной линейке заданного объекта. Размещение фигуры, создаваемой при каждом вызове функции в новом клипе, дает возможность впоследствии использовать каждую фигуру в отдельности для выполнения с ней различных манипуляций (например, вращения, изменения прозрачности) посредством воздействия на содержащий ее клип.
- Позволяет задать координаты клипа, в котором выполняется рисование.
- Для звезды позволяет задать соотношение радиусов внутренней и внешней окружностей, определяющее степень остроконечности звезды. В целях избежания возможности получения некорректных результатов значение данного параметра может изменяться в диапазоне от 0,1 до 0,9. Если пользователь задает значение, выходящее за пределы данного диапазона, оно приводится к допустимой величине.
- Возвращает ссылку на созданный данной функцией клип, на линейке которого находится нарисованная фигура. Эту ссылку можно сохранить в переменной с тем, чтобы в дальнейшем можно было обращаться к клипу.

Листинг 13.2. Функция для рисования правильных выпуклых многоугольников и звезд

```
function polyStar(
//----------------------- ПАРАМЕТРЫ ФУНКЦИИ-------------------------
        obj:Object, /* Объект, на монтажной линейке которого будет создан
        клип, содержащий требуемую фигуру */
        num:Number, /Количество сторон для многоугольника или лучей для
        звезды */
        size:Number, /* Диаметр окружности, в которую вписана требуемая
        фигура */
        x:Number, y:Number, /* Координаты динамически созданного клипа, в
        котором будет выполняться рисование */
        strokeThickness:Number, /* Толщина контура. Если обводка
        отсутствует, задается значение null */
        strokeColor:Number, /* Цвет контура. Если обводка отсутствует,
        задается значение null */
        fillColor:Number, /* Цвет заливки. Если заливка отсутствует,
        задается значение null */
        shape:String, /* Строка, определяющая тип фигуры. Допустимое
        значение "star" - для рисования звезды. Для рисования
        многоугольника данный параметр не указывается */
```

```
        starPointSize:Number /* Задается только для звезды. Отношение
        внутреннего радиуса к внешнему */
                ):MovieClip{
//--------------------------ТЕЛО ФУНКЦИИ-------------------------
        // Приведение параметра starPointSize к допустимой величине
        if (shape == "star") {
                if (starPointSize == undefined)starPointSize = 0.5;
                // Ограничения на максимальное и минимальное значения
                starPointSize=Math.min(Math.max(starPointSize, 0.1), 0.9);
                // Предыдущая строка эквивалентна двум следующим:
                //if (starPointSize < 0.1) starPointSize = 0.1;
                //if (starPointSize > 0.9) starPointSize = 0.9;
        }
        // Угол между соседними вершинами
        var stepAngle = 2 * Math.PI / num;
        // Массивы для хранения координат точек по X и по У
        var xCoordinates = new Array();
        var yCoordinates = new Array();
        // Расчет координат точек
        for (var i = 0; i < num; i++) {
                // Точки внутренней окружности (только для звезды)
                if (i != 0 && shape == "star") {
                        var xCurIn = size / 2 * starPointSize * ↵
                        Math.cos(i * stepAngle - .5 * stepAngle);
                var yCurIn = size / 2 * starPointSize * ↵
                        Math.sin(i * stepAngle - .5 * stepAngle);
                        xCoordinates.push(xCurIn);
                        yCoordinates.push(yCurIn);
                }
                /* Точки внешней окружности (для многоугольника и для
                звезды) */
                var xCurOut = size / 2 * Math.cos(i * stepAngle);
                var yCurOut = size / 2 * Math.sin(i * stepAngle);
                xCoordinates.push(xCurOut);
                yCoordinates.push(yCurOut);
        }
        /* Расчет координат последней точки на внутренней окружности для
        звезды */
        if (shape == "star") {
```

```
        var xCurIn = size / 2 * starPointSize * Math.cos(num *↓
        stepAngle - .5 * stepAngle);
        var yCurIn = size / 2 * starPointSize * Math.sin(num *↓
        stepAngle - .5 * stepAngle);
        xCoordinates.push(xCurIn);
        yCoordinates.push(yCurIn);
}
/* Вспомогательная переменная, используемая для генерирования
имени клипа, в котором будет выполняться рисование. Поскольку
может возникнуть необходимость в рисовании нескольких фигур, нужно
чтобы клип, содержащий каждую из них, имел уникальное имя */
var depth = obj.getNextHighestDepth();
/* Создаем новый клип, в котором будет выполняться рисование, на
ближайшем верхнем свободном уровне стека obj */
obj.createEmptyMovieClip("holder"+depth, depth);
with(obj["holder"+depth]){
// Позиционируем клип
_x = x;
_y = y;
// Рисование
moveTo(xCoordinates[0], yCoordinates[0]);
// Обводка создается только в случае, если заданы ее атрибуты
if (strokeColor != null && strokeThickness != null ) {
        lineStyle(strokeThickness, strokeColor);
}
// Заливка создается только, если ее цвет указан
if (fillColor != null) {
        beginFill(fillColor);
}
/* Для звезды точек в два раза больше, чем для многоугольника,
поэтому в цикле больше итераций */
if (shape == "star") {
        for (var k = 1; k < 2 * num; k++) {
                lineTo(xCoordinates[k], yCoordinates[k]);
        }
}
else {
        // Для многоугольника
        for (var k = 1; k < num; k++) {
                lineTo(xCoordinates[k], yCoordinates[k]);
        }
```

```
        }
        /* Рисуем последнюю сторону, замыкая фигуру. Вместо этой строки
        можно было бы просто вызвать метод endFill() */
        lineTo(xCoordinates[0], yCoordinates[0]);
        }
        /* После завершения рисования возвращаем ссылку на клип, в котором
        содержится нарисованная фигура */
        return obj["holder"+depth];
}
```

Применение градиентной заливки

Наряду со сплошной заливкой динамически созданные формы могут иметь градиентное заполнение. Градиент представляет собой плавный цветовой переход или "растяжку" между несколькими цветами. Цвета, входящие в состав градиента, называются порогами градиента. Инструментарий динамического рисования, как и векторный редактор Flash, позволяет синтезировать два типа градиента: линейный (linear) и круговой или радиальный (radial). В линейном градиенте цветовые пороги расположены последовательно в соответствии с направлением градиента. В радиальном градиенте цветовые пороги расположены в виде концентрических кругов или овалов. Максимальное количество цветовых порогов в градиенте не может превышать восьми. В составе градиента допускается использование прозрачных цветов.

Для применения градиентной заливки к динамически нарисованным формам используется метод `beginGradientFill()`. Этот метод имеет следующий синтаксис:

клип`.beginGradientFill(`*тип*`, `*цвета*`, `*прозрачности*`, `*распределение*`, `*матрица*`)`

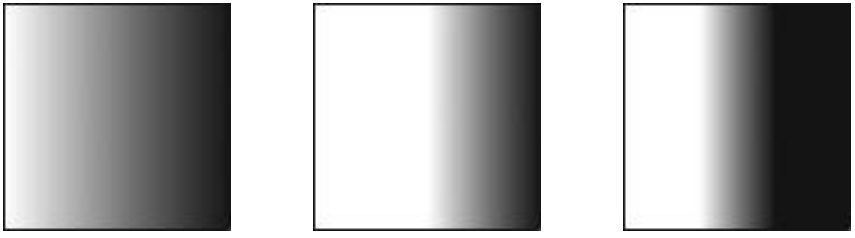
Здесь *клип* — это объект, на монтажной линейке которого выполняется рисование.

Данный метод предоставляет возможности полного контроля внешнего вида градиента, в связи с чем количество принимаемых им параметров достаточно велико. Рассмотрим их назначение подробно.

- Параметр *тип* — это строка, определяющая тип градиента: линейный или радиальный. Допустимыми значениями данного параметра являются строки `"linear"` (линейный градиент) и `"radial"` (радиальный градиент).
- Параметр *цвета* — это массив, элементами которого являются шестнадцатеричные значения цветов, соответствующих цветовым порогам создаваемого градиента. Поскольку максимальное количество цветов в градиенте не может превышать восьми, учитываются только значения первых восьми элементов передаваемого методу массива *цвета*. Данный параметр может быть указан с помощью литерала массива или идентификатором, содержащим ссылку на массив. В большинстве случаев удобнее использовать идентификатор массива.
- Параметр *прозрачности* — это массив, определяющий степень прозрачности каждого цветового порога градиента, заданного предыдущим параметром *цвета*. Порядок следования элементов в массиве *прозрачности* должен соответствовать

порядку размещения элементов в массиве *цвета*, так чтобы заданная прозрачность применялась к соответствующему цветовому порогу градиента. Каждый элемент массива задается числом от 0 (полная прозрачность) до 100 (полная непрозрачность). Если значение элемента оказывается меньше 0, используется значение 0, если значение элемента оказывается больше 100, используется значение 100. Данный параметр может быть указан с помощью литерала массива или идентификатором, содержащим ссылку на массив.

- Параметр *распределение* — это массив, определяющий распределение цветовых порогов в градиенте. Данный параметр позволяет определить, какую часть или долю градиента занимает каждый входящий в его состав цвет. Порядок следования элементов в массиве *распределение* должен соответствовать порядку размещения элементов в массивах *цвета* и *прозрачности*. Элементами данного массива являются числовые значения в диапазоне от 0 до 255. Значения элементов массива также иногда задаются с помощью шестнадцатеричных значений в диапазоне от 0x00 до 0xFF. Элементы массива *распределение* указывают, в какой части градиента находятся соответствующие им цветовые пороги, заданные массивом *цвета*. Применение данного параметра эквивалентно перемещению маркеров редактирования градиента на панели **Color Mixer** в рабочей среде. Точка местоположения цветового порога задается как доля от общей протяженности градиента. При этом весь градиент принимается за 255. Так, если в состав градиента входит два цвета, то в зависимости от заданных значений элементов массива *распределение* будут иметь место следующие результаты (рис. 13.3):
  - [0, 255] — цветовые пороги находятся в разных краях градиента (рис. 13.3, *а*);
  - [128, 255] — первый цветовой порог находится в центре градиента, второй — с краю. Половина градиента представляет собой сплошной цвет, соответствующий первому цветовому порогу (рис. 13.3, *б*);
  - [85, 170] — первый цветовой порог расположен на первой трети градиента; второй — на второй трети. Цветовой переход занимает только вторую треть градиента. Первая и третья трети градиента заполнены сплошными цветами, соответствующими его цветовым порогам (рис. 13.3, *в*).



а *б* *в*

Рис. 13.3. Результат распределения цветовых порогов в градиенте

- Параметр *матрица* — представляет собой объект, определяющий матрицу, которая задает такие характеристики градиента, как масштаб, угол скоса и поворота, а также его местоположение. Ввиду особой важности и обширности данной темы описание параметра *матрица* выносится в самостоятельный раздел.

Определение матрицы трансформации градиента

Создание градиента с помощью метода `beginGradientFill()` начинается с создания единичного градиента, центр которого расположен в начале координат. Для задания значений размеров, угла поворота, скоса (для радиального градиента) и координат градиента используется матрица трансформации, представляющая собой геометрический оператор, позволяющий выполнять преобразования градиента для придания ему требуемого вида. Такая матрица задается последним параметром метода `beginGradientFill()` в качестве объекта, содержащего определенный набор свойств. Для задания данной матрицы можно использовать один из двух возможных способов. Выбор одного или другого способа влияет на набор свойств и их значений, которые должны находиться в объекте, передаваемом методу `beginGradientFill()`.

Первый способ определения параметров градиента. Матрица типа *"box"*

Параметр *матрица* может быть задан с помощью объекта, включающего следующий обязательный набор свойств:

- `matrixType` — свойство, единственным и обязательным значением которого является строка `"box"`. Наличие данного свойства указывает, что для задания матрицы трансформации используется данный способ;
- `x` — данное свойство задает горизонтальную координату левого верхнего угла градиента;
- `y` — данное свойство задает вертикальную координату левого верхнего угла градиента;
- `w` — данное свойство задает ширину градиента в пикселах, то есть расстояние между его крайними цветовыми порогами по горизонтали;
- `h` — данное свойство задает высоту градиента в пикселах, то есть расстояние между его крайними цветовыми порогами по вертикали;
- `r` — данное свойство задает угол поворота градиента в радианах. Для преобразования градусов в радианы используется следующая формула:

$$\varphi_{рад} = \varphi_{град} / 180 \cdot \pi,$$

где $\varphi_{рад}$ — угол в радианах; $\varphi_{град}$ — угол в градусах.

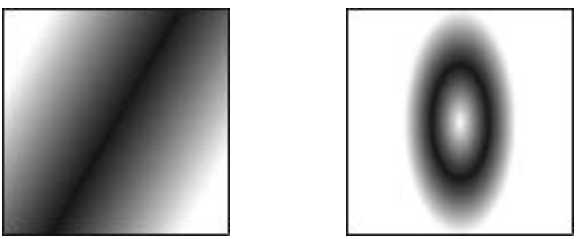
Важно отметить, что координаты градиента и угол его поворота задаются относительно начала координат объекта, в котором выполняется рисование, а не относительно нарисованной фигуры. Задавая значение свойства `r`, можно поворачивать только линейный градиент. Изменение значения данного свойства для радиального градиента не дает видимого результата.

Если значения свойств `w` и `h` различны, то радиальный градиент приобретает форму овала. Если значение свойства `r` равно нулю, то есть поворот отсутствует, то на внешний вид линейного градиента влияет только значение свойства `w`. Тем не менее, если линейный градиент повернут, то его высота устанавливается в соответствии со значением свойства `h`.

Следующий код рисует квадрат размером 100×100 пикселов и заполняет его линейным градиентом, состоящим из трех цветов. Градиент повернут на 30° по часовой стрелке (рис. 13.4, *а*). Поскольку угол поворота должен быть задан в радианах, для получения числа π мы воспользовались встроенной константой класса `Math`.

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
colors = [0xFFFFFF, 0x000000, 0xFFFFFF];
alphas = [100, 100, 100];
ratios = [0, 128, 255];
matrix = {matrixType:"box", x:0, y:0, w:100, h:100, r:Math.PI/6}
with (holder) {
       lineStyle(1, 0x000000);
       beginGradientFill("linear", colors, alphas, ratios, matrix);
       moveTo(0, 0);
       lineTo(100, 0);
       lineTo(100, 100);
       lineTo(0, 100);
       lineTo(0, 0);
}
```

Теперь внесем в этот код некоторые изменения, для того чтобы получить радиальный градиент, отношение ширины к высоте у которого составляет 0,5 (рис. 13.4, *б*).



а *б*

Рис. 13.4. Применение линейного и радиального градиентов с использованием матрицы типа `"box"`

Обратите внимание, что поскольку ширина нашего радиального градиента составляет 50 пикселов, то для того, чтобы его центр находился точно в центре квадрата,

длина стороны которого составляет 100 пикселов, необходимо задать свойству `x` объекта `matrix` значение 25.

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
colors = [0xFFFFFF, 0x000000, 0xFFFFFF];
alphas = [100, 100, 100];
ratios = [0, 128, 255];
matrix = {matrixType:"box", x:25, y:0, w:50, h:100, r:0}
with (holder) {
       lineStyle(1, 0x000000);
       beginGradientFill("radial", colors, alphas, ratios, matrix);
       moveTo(0, 0);
       lineTo(100, 0);
       lineTo(100, 100);
       lineTo(0, 100);
       lineTo(0, 0);
}
```

Описанный метод определения матрицы трансформации градиента обладает своими преимуществами и недостатками. К преимуществам можно отнести очевидную простоту его применения. В то же время существенным недостатком данного метода является невозможность выполнения таких преобразований, как поворот радиального градиента, а также вертикальный и горизонтальный скос. Эти недостатки могут быть устранены с помощью другого способа определения матрицы трансформации, который будет описан далее в этой главе.

Анимация линейного градиента

В качестве иллюстрации применения матрицы типа `"box"` для управления градиентом приведем пример создания функции `loopLinearGradient()`, выполняющей рисование прямоугольной формы, заполненной линейным градиентом, который перемещается внутри этой формы наподобие бесконечной ленты справа налево или слева направо. Приведем перечень функциональных требований к функции `loopLinearGradient()`.

- Функция должна создавать клип, в котором будет выполняться рисование. При этом необходимо предусмотреть возможности передачи ей в качестве параметра ссылки на объект, на монтажной линейке которого будет размещен создаваемый клип.
- Размеры и координаты клипа, содержащего градиент, передаются функции в качестве параметров.
- Линейный градиент, создаваемый этой функцией, должен состоять из двух цветов. При этом значения данных цветов и степени их прозрачности передаются в качестве параметров.

- В функции должна быть предусмотрена возможность задания скорости и направления движения градиента.

Для реализации бесконечного движения градиента внутри прямоугольной формы используется следующий принцип. В качестве параметров функции передаются два цвета, на основе которых создаются пять цветовых порогов градиента таким образом, чтобы цвета в градиенте чередовались, начиная со второго. Цветовые пороги должны размещаться в градиенте на равном расстоянии друг от друга. Затем градиент масштабируется по горизонтали, увеличиваясь в два раза, и размещается так, чтобы в пределах прямоугольной формы были видны только три правых цветовых порога (рис. 13.5). После этого остается только постепенно с требуемым шагом изменять его горизонтальную координату, которая задается свойством `x` объекта `matrix`. Для того чтобы движение выполнялось непрерывно без скачков наподобие "бесконечной ленты", необходимо, чтобы в момент, когда крайний порог градиента, до тех пор остававшийся за пределами формы, становился виден, происходило смещение градиента в исходное положение. Таким образом, в пределах области видимости всегда остаются только три цветовых порога градиента.

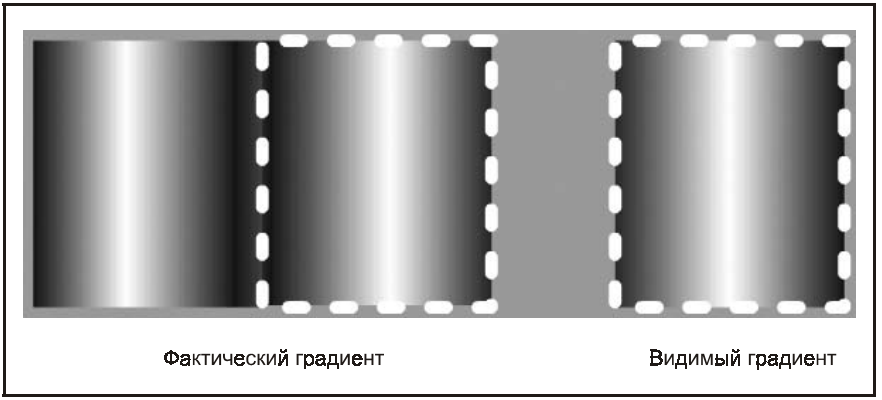


Рис. 13.5. Принцип реализации бесконечного движения линейного градиента из двух цветов

Исходный код функции `loopLinearGradient()` с подробными комментариями приводится в листинге 13.3.

Листинг 13.3. Бесконечное движение линейного градиента

```
function loopLinearGradient(
                clip, /* Ссылка на объект, на монтажной линейке которого
                будет создан клип с градиентом */
                color1, color2, // Цвета, составляющие градиент
                alpha1, alpha2, // Прозрачность порогов градиента
```

```
                x, y, /* Координаты левого верхнего угла клипа с
                градиентом */
                width, height, // Ширина и высота клипа с градиентом
                speed, // Скорость смещения градиента
                dir /* Флаг, определяющий направление движения градиента:
                слева направо - true, справа налево - false или undefined
                */
                ){
/* Формируем массив цветов из цветов, переданных в качестве параметров
функции так, чтобы имело место чередование. Это необходимо для создания
впечатления движения "бесконечной ленты" */
var colors:Array = [color2, color1, color2, color1, color2];
// Формируем массив из значений прозрачности для цветов градиента
var alphas:Array = [alpha2, alpha1, alpha2, alpha1, alpha2];
/* Распределяем цветовые пороги равномерно на протяжении всего градиента
с интервалом, равным 255/4= 63,75 */
var ratios:Array = [0, .25 * 255, .5 * 255, 0.75 * 255, 255];
/* Создаем объект matrix, растягивающий градиент в ширину в два раза и
позиционирующий его так, чтобы в пределах нарисованного прямоугольника
были видны только три правых цветовых порога */
var matrix:Object = {matrixType:"box", x:-width, y:0, w:2 * width, h:0,
r:0};
/* На монтажной линейке указанного параметром clip объекта создаем клип,
в который будет помещен градиент */
clip.createEmptyMovieClip("gradientHolder", clip.getNextHighestDepth());
// Позиционируем клип, содержащий градиент.
gradientHolder._x = x; // Левая граница градиента
gradientHolder._y = y; // Верхняя граница градиента
gradientHolder.onEnterFrame = function() {
        // Движение градиента слева направо...
        if (dir || dir == undefined) {
                /* Если крайний левый цветовой порог градиента появился в
                пределах видимости, возвращаем градиент в исходные коорди-
                наты, т.о. реализуется зацикливание движения */
                if (matrix.x >= 0) {
                        matrix.x = -width;
                }
                matrix.x += speed;
        }
        // Движение градиента справа налево...
        else {
                /* Если крайний правый цветовой порог градиента появился в
                пределах видимости, возвращаем градиент в исходные коорди-
                наты, т. о. реализуется зацикливание */
```

```
        if (matrix.x <= -width) {
                        matrix.x = 0;
                }
                matrix.x -= speed;
        }
        // Рисование...
        with (gradientHolder) {
                clear();
                beginGradientFill("linear", colors, alphas, ratios,
                matrix);
                moveTo(0, 0);
                lineTo(width, 0);
                lineTo(width, height);
                lineTo(0, height);
                lineTo(0, 0);
                endFill();
        }
};
}
```

Для перемещения градиента, состоящего из темно-синего и голубого непрозрачных цветов внутри прямоугольника размером 300×50 пикселов со скоростью 120 пикселов в секунду в направлении справа налево, нужно вызвать функцию `loopLinearGradient()` со следующими параметрами (клип, создаваемый этой функцией, будет помещен на основную монтажную линейку):

```
loopLinearGradient(_root, 0x00CCFF, 0x0000CC, 100, 100, 100, 100, 300,↵
50, 12, false);
```

Второй способ определения параметров градиента. Матрица общего преобразования

Альтернативой рассмотренному выше подходу определения матрицы трансформации градиента является передача методу `beginGradientFill()` в качестве последнего параметра объекта, включающего обязательный набор следующих свойств: `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`.

Этот объект представляет собой матрицу двумерного преобразования для однородных координат размерностью 3×3 следующего вида:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}.$$

Мы не будем здесь вдаваться в описание применения матричного аппарата. Информацию по этой теме читатель сможет получить в *главе 26*, где будет приведено описание применения матриц для осуществления трехмерных преобразований. Для понимания механизма управления внешним видом градиента необходимо иметь общее представление о матрицах и знать правила их умножения. Читатели, не знакомые с этими вопросами, могут обратиться к *главе 26* или для получения более подробной информации — к специальной литературе, посвященной данной теме.

Как было сказано ранее, создание градиента начинается с создания единичного градиента. Единичный градиент имеет размер 1×1 пиксел, и его центр расположен в начале координат объекта, на монтажной линейке которого выполняется рисование. Такой градиент может быть описан матрицей следующего вида:

$$E = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Данную матрицу можно представить в виде объекта, используемого в качестве последнего параметра метода `beginGrasdientFill()`.

```
var matrix = {a:1, b:0, c:0, d:0, e:1, f:0, g:0, h:0, i:1}
```

Естественно сам по себе такой градиент не имеет практической значимости, поскольку увидеть его можно только при довольно большом увеличении. Для того чтобы выполнить с градиентом требуемые трансформации, такие как масштабирование, вращение, скос и перемещение, необходимо умножить единичную матрицу на матрицу, отвечающую за соответствующую трансформацию. После этого элементы результирующей матрицы присваиваются в качестве значений соответствующим свойствам объекта, который передается в качестве последнего параметра методу `beginGrasdientFill()`. Рассмотрим матрицы преобразования для выполнения перечисленных выше трансформаций.

Масштабирование градиента

Для масштабирования градиента используется матрица трансформации следующего вида:

$$T_{\text{масштаб}} = \begin{pmatrix} xscale & 0 & 0 \\ 0 & yscale & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Здесь *xscale* — это коэффициент масштабирования градиента по горизонтали, *yscale* — коэффициент масштабирования градиента по вертикали. Следует иметь в виду, что масштабирование выполняется относительно точки начала координат монтажной линейки, на которой выполняется рисование.

Объект, с помощью которого может быть представлена матрица, осуществляющая масштабирование, имеет следующий вид:

```
var scaleObject = {a:xscale, b:0, c:0, d:0, e:yscale, f:0, g:0, h:0, i:1}
```

Приведенный ниже код рисует квадрат со стороной 100 пикселов, залитый радиальным градиентом. Масштаб градиента по горизонтали составляет 100 пикселов, а по

вертикали 200 (то есть он в сто раз шире и в двести раз выше единичного градиента). Поскольку к градиенту не применялось смещение, его центр по-прежнему остается в начале координат (которое в данном примере совпадает с левым верхним углом квадрата).

```
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
colors = [0xFFFFFF, 0x000000, 0xFFFFFF];
alphas = [100, 100, 100];
ratios = [0, 128, 255];
matrix = {a:100, b:0, c:0, d:0, e:200, f:0, g:0, h:0, i:1};
with (holder) {
       lineStyle(1, 0x000000);
       beginGradientFill("radial", colors, alphas, ratios, matrix);
       moveTo(0, 0);
       lineTo(100, 0);
       lineTo(100, 100);
       lineTo(0, 100);
       lineTo(0, 0);
}
```

Результат работы этого сценария приведен на рис. 13.6.

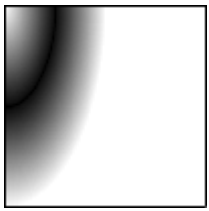


Рис. 13.6. Масштабирование радиального градиента

Смещение градиента

Смещение градиента выполняется с помощью матрицы преобразования следующего вида:

$$T_{\text{смещение}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x & y & 1 \end{pmatrix}.$$

Здесь x и y — это значения, задающие смещение градиента относительно начала координат по горизонтали и вертикали соответственно.

Объект, с помощью которого может быть представлена матрица, осуществляющая смещение градиента, имеет следующий вид:

```
var moveObject = {a:1, b:0, c:0, d:0, e:1, f:0, g:x, h:y, i:1}
```

Для выполнения одновременного масштабирования и смещения градиента можно использовать матрицу комбинированного преобразования, представляющую собой произведение двух матриц соответствующих преобразований:

$$T_{\text{масштаб, смещение}} = \begin{pmatrix} xscale & 0 & 0 \\ 0 & yscale & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x & y & 1 \end{pmatrix} = \begin{pmatrix} xscale & 0 & 0 \\ 0 & yscale & 0 \\ x & y & 1 \end{pmatrix}.$$

Соответственно объект, с помощью которого может быть представлена матрица, осуществляющая масштабирование и смещение градиента, имеет следующий вид:

```
var scaleAndMoveObject = {a:xscale, b:0, c:0, d:0, e:yscale, f:0, g:x,↵
h:y, i:1}
```

На рис. 13.7 показан радиальный градиент, полученный с помощью предыдущего сценария, в котором объект `matrix` имеет следующий вид.

```
matrix = {a:100, b:0, c:0, d:0, e:50, f:0, g:50, h:50, i:1};
```

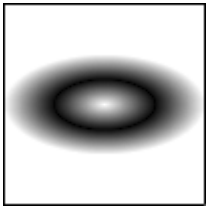


Рис. 13.7. Масштабирование и смещение радиального градиента

Поворот градиента

Поворот градиента выполняется с помощью матрицы преобразования следующего вида:

$$T_{\text{поворот}} = \begin{pmatrix} \cos(\varphi) & \sin(\varphi) & 0 \\ -\sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Здесь φ — это угол поворота градиента относительно начала координат.

Объект, с помощью которого может быть представлена матрица, осуществляющая поворот градиента, имеет следующий вид:

```
with(Math){

var rotateObject = {a:cos(φ), b:sin(φ), c:0, d:-sin(φ), e:cos(φ), f:0,↵
g:0, h:0, i:1}

}
```

Вычисление тригонометрических функций в ActionScript выполняется с помощью методов встроенного класса `Math`. Эти методы являются статическими, то есть вызываются непосредственно из класса, например, `Math.cos(a)`, `Math.sin(a)`, `Math.tan(a)`. Для того чтобы всякий раз при вызове данных методов не использовать прямую ссылку на класс `Math`, для более компактной записи здесь используется предложение `with`.

Для получения видимого результата преобразование поворота необходимо скомбинировать с преобразованием масштабирования и/или смещения.

Скос градиента

Скос градиента выполняется с помощью матрицы преобразования следующего вида:

$$T_{\text{скос}} = \begin{pmatrix} 1 & \tan(\varphi_y) & 0 \\ \tan(-\varphi_x) & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Здесь φ_x и φ_y — это углы скоса градиента относительно начала координат по горизонтали и вертикали соответственно.

Объект, с помощью которого может быть представлена матрица, осуществляющая скос градиента, имеет следующий вид:

```
with(Math){
var skewObject={a:1, b:tan(φy), c:0, d:tan(-φx), e:1, f:0, g:0, h:0, i:1}
}
```

Для получения видимого результата преобразование скоса необходимо скомбинировать с преобразованием масштабирования и/или смещения.

Выполнение комбинированных преобразований градиента

Итак, мы выяснили, что трансформация градиента выполняется путем умножения единичной матрицы исходного градиента на матрицу соответствующей трансформации. Замечательной особенностью матричных преобразований является то, что они имеют накопительный эффект. Так, пусть у нас имеется исходная матрица единичного градиента:

$$E = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Для того чтобы увеличить масштаб этого градиента, достаточно умножить исходную единичную матрицу на матрицу, выполняющую масштабирование:

$$M_{\text{рез}} = E \cdot T_{\text{масштаб}},$$

где $M_{\text{рез}}$ — это результирующая матрица преобразования, описывающая градиент со всеми примененными к нему трансформациями.

Для того чтобы повернуть отмасштабированный градиент, необходимо последовательно перемножить три матрицы:

$$M_{рез} = E \times T_{масштаб} \times T_{поворот}.$$

Чтобы к отмасштабированному и повернутому градиенту применить скос, перемножаем четыре матрицы:

$$M_{рез} = E \times T_{масштаб} \times T_{поворот} \times T_{скос}.$$

И, наконец, для того чтобы последовательно применить к градиенту масштабирование, поворот, скос и смещение, нужно последовательно перемножить пять матриц:

$$M_{рез} = E \times T_{масштаб} \times T_{поворот} \times T_{скос} \times T_{смещение}.$$

После выполнения умножения получившуюся в результате матрицу $M_{рез}$ нужно представить в виде объекта со свойствами `a`, `b`, ..., `i`, значения которых соответствуют значениям элементов данной матрицы. Таким образом, будет получен объект, описывающий вид градиента, который должен быть передан методу `beginGradientFill()` в качестве последнего параметра.

При выполнении матричных преобразований важно иметь в виду следующее.

- Поскольку операция умножения матриц некоммутативна (то есть $AB \neq B \cdot A$), имеет значение порядок выполнения преобразований. В большинстве случаев вначале выполняется масштабирование градиента, а в конце — смещение, хотя постановка конкретной задачи может требовать иного порядка выполнения операций.
- Все перечисленные преобразования не изменяют значений свойств `c:0`, `f:0`, `i:1` объекта, представляющего матрицу трансформации градиента. То есть третий столбец этой матрицы всегда остается неизменным. Его наличие обусловлено тем, что для выполнения перемещения в двумерном пространстве с помощью матричных преобразований необходимо использование однородных координат и, соответственно, матрицы размерностью 3×3.

В листинге 13.4 реализуется подход, позволяющий управлять градиентом, используя рассмотренный аппарат матричных преобразований.

Листинг 13.4. Создание градиента с помощью матрицы общего преобразования

```
/* Создаем объект, представляющий исходную единичную матрицу градиента.
Значения свойств этого объекта будут в дальнейшем изменяться, отражая
выполнение требуемых преобразований */

var matrix:Object = {a:1, b:0, c:0, d:0, e:1, f:0, g:0, h:0, i:1};

/* Функция, выполняющая умножение двух матриц, элементами которых
являются свойства a, b, ..., i объектов, описывающих исходный градиент
(obj1), и заданную матрицу трансформации (obj2) соответственно. Умножение
выполняется слева направо. Результат возвращается в виде объекта со
свойствами a, b, ..., i, описывающего трансформированный градиент */

function transform(obj1:Object, obj2:Object):Object {

       var resultObj = new Object(); /* Объект, описывающий
       трансформированный градиент */
```

```
        resultObj.a = obj1.a * obj2.a + obj1.b * obj2.d + obj1.c *
obj2.g;
        resultObj.b = obj1.a * obj2.b + obj1.b * obj2.e + obj1.c *
obj2.h;
        resultObj.c = 0 // Всегда 0
        resultObj.d = obj1.d * obj2.a + obj1.e * obj2.d + obj1.f *
obj2.g;
        resultObj.e = obj1.d * obj2.b + obj1.e * obj2.e + obj1.f *
obj2.h;
        resultObj.f = 0 // Всегда 0
        resultObj.g = obj1.g * obj2.a + obj1.h * obj2.d + obj1.i *
obj2.g;
        resultObj.h = obj1.g * obj2.b + obj1.h * obj2.e + obj1.i *
obj2.h;
        resultObj.i = 1// Всегда 1
        return resultObj;
}
// Функция, выполняющая масштабирование градиента
function scaleGradient(xscale:Number, yscale:Number):Void {
        // Матрица масштабирования
        var scaleObject = {a:xscale, b:0, c:0, d:0, e:yscale, f:0, g:0,
        h:0, i:1};
        /* Умножение текущей матрицы преобразования градиента на матрицу
        масштабирования и сохранение результата умножения в объекте
        matrix, описывающем требуемый градиент */
        matrix = transform(matrix, scaleObject);
}
// Функция, выполняющая позиционирование градиента
function moveGradient(x:Number, y:Number):Void {
        // Матрица перемещения
        var moveObject = {a:1, b:0, c:0, d:0, e:1, f:0, g:x, h:y, i:1};
        /* Умножение текущей матрицы преобразования градиента на матрицу
        перемещения и сохранение результата умножения в объекте matrix,
        описывающем требуемый градиент */
        matrix = transform(matrix, moveObject);
}
// Функция, выполняющая поворот градиента
function rotateGradient(alpha:Number):Void {
        with (Math) {
                var alpha = alpha / 180 * PI; /* Преобразование из градусов
                в радианы */
```

```
                //Матрица поворота
                var rotateObject = {a:cos(alpha), b:sin(alpha), c:0,↳
                d:-sin(alpha), e:cos(alpha), f:0, g:0, h:0, i:1};
                /* Умножение текущей матрицы преобразования градиента на
                матрицу поворота и сохранение результата умножения в
                объекте matrix, описывающем требуемый градиент */
                matrix = transform(matrix, rotateObject);
        }
}
// Функция, выполняющая горизонтальный и вертикальный скос градиента
function skewGradient(alphaX:Number, alphaY:Number):Void {
        with (Math) {
                // Преобразование из градусов в радианы
                var alphaX = alphaX / 180 * PI;
                var alphaY = alphaY / 180 * PI;
                // Матрица скоса
                var skewObject = {a:1, b:tan(alphaY), c:0, d:tan(-alphaX),↳
                e:1, f:0, g:0, h:0, i:1};
        }
        /* Умножение текущей матрицы преобразования градиента на матрицу
        скоса и сохранение результата умножения в объекте matrix,
        описывающем требуемый градиент */
        matrix = transform(matrix, skewObject);
}
/* Выполнение трансформаций исходного единичного градиента, описываемого
объектом matrix */
scaleGradient(50, 100); // Масштабирование в 50 раз по X и в 100 раз по Y
rotateGradient(45); // Поворот на 45 градусов по часовой стрелке
skewGradient(45, 45); /* Скос на 45 градусов по горизонтали и по вертика-
ли (эквивалентно повороту на 45 градусов по часовой стрелке) */
moveGradient(50, 50); /* Смещение на 50 пикселов по X и по Y относительно
начала координат */
// Рисование формы, применение градиента...
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
colors = [0xFFFFFF, 0x000000, 0xFFFFFF];
alphas = [100, 100, 100];
ratios = [0, 128, 255];
with (holder) {
        lineStyle(1, 0x000000);
        beginGradientFill("radial", colors, alphas, ratios, matrix);
        moveTo(0, 0);
```

```
        lineTo(100, 0);
        lineTo(100, 100);
        lineTo(0, 100);
        lineTo(0, 0);
}
```

Изображение, полученное в результате выполнения этого сценария, представлено на рис. 13.8, *а*. Рисунки 13.8, *б*, 13.8, *в* и 13.8, *г* иллюстрируют результат, полученный с использованием того же программного кода, но при выполнении других преобразований.

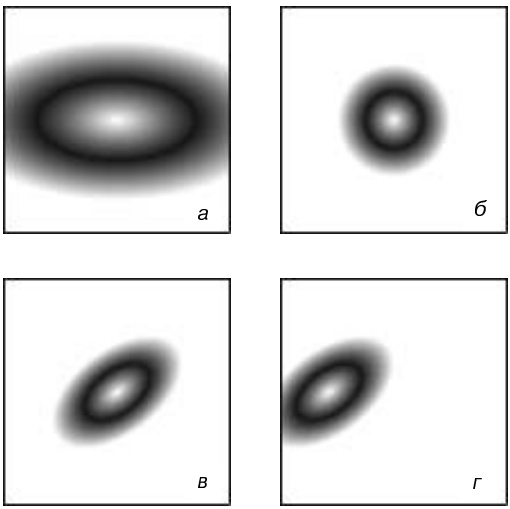


Рис. 13.8. Трансформация градиента с помощью матрицы общего преобразования

Для выполнения пропорционального масштабирования единичного радиального градиента в пятьдесят раз и его центрирования относительно нарисованного квадрата необходимо выполнить следующие преобразования (13.8, *б*):

```
scaleGradient(50, 50);
moveGradient(50, 50);
```

Для выполнения масштабирования, скоса и центрирования градиента необходимо выполнить следующие преобразования (13.8, *в*):

```
scaleGradient(50, 50);
skewGradient(30, 0);
moveGradient(50, 50);
```

Обратите внимание, что если вначале выполняется смещение, а потом скос, то результат будет иным ((13.8, *г*). Это объясняется тем, что все трансформации осущест-

вляются относительно начала координат, которое в данном случае совпадает с левой верхней вершиной квадрата:

```
scaleGradient(50, 50);
moveGradient(50, 50);
skewGradient(30, 0);
```

Рисование криволинейных сегментов

Программный интерфейс рисования ActionScript наряду с возможностью рисования прямых линий позволяет также рисовать кривые. Для рисования кривых в Flash применяются квадратичные кривые Безье. Квадратичная кривая Безье обладает следующими характерными особенностями (рис. 13.9):

- определяется многочленом второй степени;
- расположена внутри треугольника, координаты вершин которого однозначно определяют внешний вид кривой;
- две вершины определяющего треугольника совпадают с конечными точками кривой;
- векторы касательных в конечных точках кривой пересекаются в третьей (не лежащей на кривой) вершине определяющего треугольника;
- кривая пересекает любую прямую линию не чаще, чем определяющий ее треугольник (уменьшение вариации).

Начальная и конечная точки кривой называются опорными или якорными (anchor points). Точка, лежащая на пересечении касательных к опорным точкам, называется управляющей (control points). Амплитуда и радиус кривизны кривой определяются взаимным расположением этих трех точек.

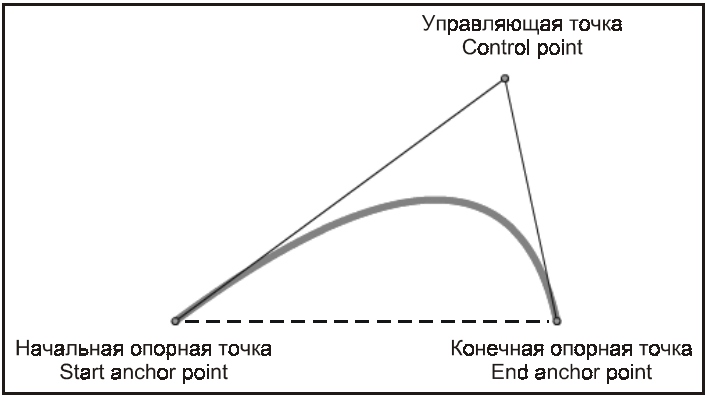


Рис. 13.9. Квадратичная кривая Безье и определяющие ее точки

Для рисования криволинейного сегмента используется метод `curveTo()`. Этот метод имеет следующий синтаксис:

клип.curveTo(*Хуправляющая*, *Yуправляющая*, *Хопорная*, *Yопорная*)

Здесь *клип* — это объект, на монтажной линейке которого выполняется рисование.

Параметры *Хуправляющая* и *Yуправляющая* — это координаты управляющей точки кривой относительно начала координат клипа *клип*. Эти параметры задаются числовыми литералами или выражениями, возвращающими число.

Параметры *Хопорная* и *Yопорная* — это координаты конечной опорной точки кривой относительно начала координат клипа *клип*. Эти параметры задаются числовыми литералами или выражениями, возвращающими число.

Местоположение начальной опорной точки кривой задается методом `moveTo()`. После вызова метода `curveTo()` текущая позиция рисования перемещается в последнюю опорную точку созданной кривой. Таким образом, в результате последовательного многократного вызова метода `curveTo()` создается непрерывный контур, состоящий из криволинейных сегментов. Следующий сценарий рисует фигуру, изображенную на рис. 13.10.

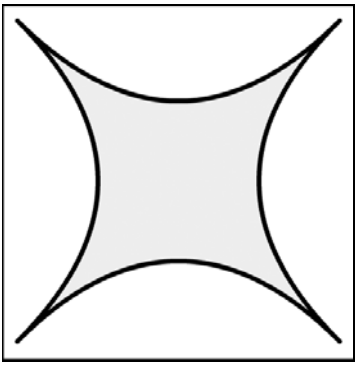


Рис. 13.10. Применение метода `curveTo()`

```
_root.createEmptyMovieClip("mc", 1);
with (_root.mc) {
       lineStyle(6, 0x000000, 100);
       beginFill(0xEEEEEE);
       moveTo(0, 0);
       curveTo(200, 200, 400, 0);
       curveTo(200, 200, 400, 400);
       curveTo(200, 200, 0, 400);
       curveTo(200, 200, 0, 0);
       endFill();
}
```

В качестве иллюстрации, позволяющей понять механику использования квадратичных кривых, приведем сценарий, который рисует кривую, состоящую из произвольного количества сегментов, и обеспечивает возможность управления этими сегментами. Сценарий, приведенный в листинге 13.5, рисует синусоидальную кривую, и дает возможность управлять положением ее опорных и управляющих точек (рис. 13.11). Для реализации этой задачи в библиотеке документа находятся два символа типа MovieClip, представляющие опорную и управляющую точки. Для этих символов определены идентификаторы связи `anchor` и `control` соответственно. С помощью метода `attachMovie()` выполняется динамическое создание экземпляров этих клипов. Поскольку требуется, чтобы изначально кривая имела вид синусоиды, опорные точки должны быть расположены на равном расстоянии друг от друга и лежать на одной прямой. Управляющие точки по горизонтали должны располагаться строго между соответствующими им опорными точками, а по вертикали они должны, чередуясь, располагаться на одном и том же расстоянии, но с разных сторон от линии, соединяющей опорные точки. Расчет координат клипов выполняется для рабочей области размером 550×400 пикселов (используя свойства класса `Stage` можно унифицировать приведенный код для любого размера рабочей области).

Задача организации перетаскивания уже многократно описывалась в предыдущих главах. В данном случае для ее решения используется метод `startDrag()`, причем область, в которой выполняется перетаскивание, ограничивается размерами сцены.

После размещения клипов и обеспечения возможности их перетаскивания остается только соединить точки, соответствующие координатам клипов, представляющих опорные точки, кривыми, причем местоположение соответствующих управляющих точек определяется координатами представляющих их клипов. Для большей наглядности клипы опорных точек кривых соединяются с соответствующими им клипами управляющих точек с помощью прямых линий. Поскольку в качестве параметров методов рисования используются координаты клипов, их перемещение отражается на кривой. Важно только выполнять очистку экрана (с помощью метода `clear()`) перед каждой следующей перерисовкой изображения.

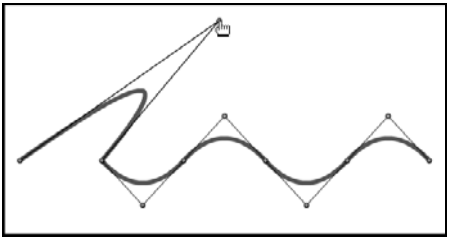


Рис. 13.11. Динамическое управление кривыми

Листинг 13.5. Динамическое управление кривыми

```
var maxPoints = 6; // Число опорных точек
var j = -1; /* Вспомогательная переменная, используемая в дальнейшем при
позиционировании контрольных точек кривых */
```

```
// Присоединение экземпляров
for (i = 1; i <= maxPoints; i++) {
        //Создание и размещение клипов для представления опорных точек
        this.attachMovie("anchor", "anchor" + i, i);
        /* Опорные точки по горизонтали размещаются на равном расстоянии
        друг от друга с одинаковым отступом от границ сцены */
        this["anchor" + i]._x = 550 * ((i-1)/maxPoints + 1 / ↳
        maxPoints/2);
        // Координата опорных точек по вертикали (константа)
        this["anchor" + i]._y = 200;
        if (i != 1) { //Так как управляющих точек на одну меньше, чем
                      //опорных
                //Создание и размещение клипов для представления
                управляющих точек
                this.attachMovie("control", "control" + i, i + maxPoints);
                /* Управляющие точки по горизонтали размещаются между
                опорными */
                this["control" + i]._x = (this["anchor" + (i-1)]._x + ↳
                this["anchor" + i]._x)/2;
                /* Управляющие точки по вертикали размещаются на расстоянии
                50 пикселов от опорных */
                this["control" + i]._y = 200 + 50*j;
                j=-j; /* Меняем знак для того, чтобы управляющие точки
                располагались чередуясь сверху и снизу от кривой */
                }
//--------ОРГАНИЗАЦИЯ ПЕРЕТАСКИВАНИЯ КЛИПОВ, ПРЕДСТАВЛЯЮЩИХ ТОЧКИ--------
        this["anchor" + i].onPress = function() {
                startDrag(this, false, 0, 0, 550, 400);
        };
        this["anchor" + i].onRelease = function() {
                stopDrag();
        };
        this["control" + i].onPress = function() {
                startDrag(this, false, 0, 0, 550, 400);
        };
        this["control" + i].onRelease = function() {
                stopDrag();
        };
}
//-------------------------------РИСОВАНИЕ-----------------------------
this.onEnterFrame = function() {
```

```
        // Очистка экрана перед очередной перерисовкой
        this.clear();
        // Начинаем с первой опорной точки
        this.moveTo(anchor1._x, anchor1._y);//Первая точка - текущая
        // Соединяем опорные точки с помощью кривых
        for (i = 2; i <= maxPoints; i++) {
          this.lineStyle(5, 0xFF0000, 100);
           /*Привязываем координаты опорных и управляющих точек к
           координатам изображающих их клипов */
           this.curveTo(this["control" + i]._x, this["control" + i]._y,↵
           this["anchor" + i]._x, this["anchor" + i]._y);
        }
        /* Соединяем опорные точки с соответствующими им управляющими
        точками прямыми линиями */
        this.moveTo(anchor1._x, anchor1._y);
        for (i = 2; i <= maxPoints; i++) {
          this.lineStyle(0, 0x000000, 100);
          this.lineTo(this["control" + i]._x, this["control" + i]._y);
          this.lineTo(this["anchor" + i]._x, this["anchor" + i]._y);
        }
};
```

Динамическое маскирование

Наряду с возможностями динамического рисования класс `MovieClip` предоставляет возможность динамического создания масок на основе экземпляров клипов. При этом один клип выполняет функции маски для другого клипа, называемого маскируемым. Эффект маски заключается в том, что графическое содержимое маскируемого объекта видно только сквозь маску, представляющую собой своеобразное отверстие. В качестве динамической маски может быть использован клип, монтажная линейка которого состоит из произвольного количества слоев и кадров. Графическое содержимое такого клипа также может быть создано динамически. Для создания динамической маски используется метод `setMask()`, имеющий следующий синтаксис:

маскируемый клип`.setMask(`*клип-маска*`)`

Здесь *маскируемый клип* — это ссылка на маскируемый клип. В качестве маскируемого объекта также может быть использована основная монтажная линейка, то есть `_root`.

Параметр *клип-маска* — это ссылка на клип, используемый в качестве маски.

Следующая строка кода заставляет клип с именем mask выполнять функцию маски для клипа с именем masked.

```
masked.setMask(mask);
```

Следует обратить внимание на то, что клип, выполняющий функцию динамической маски, в отличие от маски, создаваемой в рабочей среде, может маскировать только один объект. Последовательный вызов метода setMask() для разных объектов с одним и тем же параметром, приведет к тому, что маскирование будет применено только к объекту, последним вызвавшему данный метод. Так, если имеется два клипа с именами a и b и для каждого из них вызывается метод setMask() с передачей ему ссылки на клип-маску с именем mask, то в результате выполнения кода маскирование будет применено только к клипу b.

```
a.setMask(mask); // Применение маски к клипу a
b.setMask(mask); // Действие маски переключается с клипа a на клип b
```

Действие метода setMask() имеет приоритет над действием маски, созданной в рабочей среде с использованием слоев типа Mask (маска) и Masked (маскируемый). Так, пусть в рабочей среде на слое типа Mask находится клип с именем mask, выполняющий роль маски, а на слое типа Masked находится маскируемый клип с именем masked. Следующий код приведет к тому, что клипы поменяются своими функциями, в результате чего клип Masked станет маской для клипа Mask.

```
mask.setMask(masked)
```

Как известно, действие маски, созданной в рабочей среде, не распространяется на текст, набранный с применением машинно-независимого шрифта (device font). Тем не менее машинно-независимый шрифт в составе клипа может маскироваться с помощью динамической маски. Единственное возникающее при этом ограничение заключается в том, что вне зависимости от фактической формы маски она всегда будет иметь прямоугольные очертания, соответствующие ограничивающей рамке клипа, используемого в качестве маски. Эта особенность иллюстрируется рис. 13.12. Машинно-независимый шрифт сам, однако, не может быть использован в качестве маски.

При необходимости отключения действия динамической маски нужно передать методу setMask() в качестве параметра значение null. Следующая строка кода прекращает действие динамической маски на клип с именем masked.

```
masked.setMask(null)
```

В качестве иллюстрации возможности применения динамической маски рассмотрим пример, в котором реализуется эффект линзы, представленный на рис. 13.13. Действие эффекта линзы выражается в том, что при перемещении изображения линзы над картинкой внутри линзы отображается та же картинка, но с большим масштабом. При этом необходимо, чтобы при нахождении линзы над определенной точкой исходной картинки в ее центре оказывалась та же точка увеличенной картинки.

Для реализации этого примера мы создаем на сцене два экземпляра одного и того же клипа, содержащего картинку с изображением карты. Масштаб верхнего экземпляра должен быть больше масштаба нижнего в соответствии с заданным коэффициентом масштабирования.

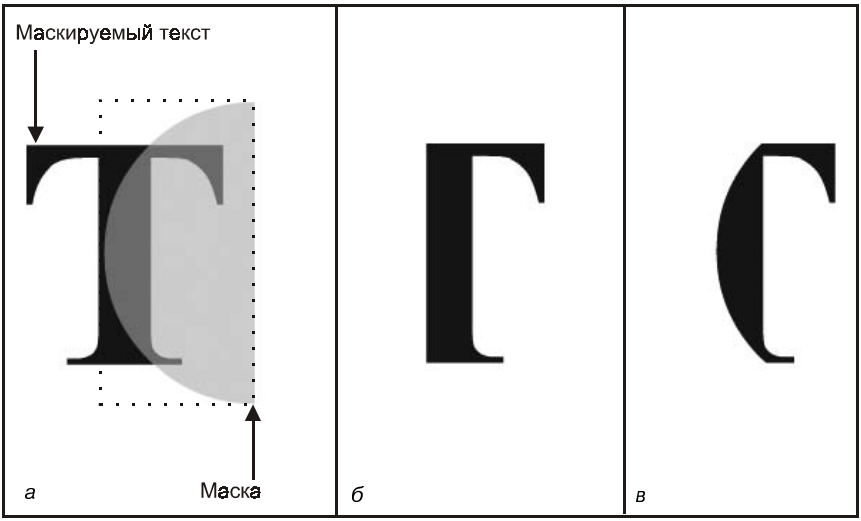


Рис. 13.12. Особенность динамического маскирования машинно-независимых шрифтов: Исходная маска (*а*); результат применения маски к машинно-независимому шрифту (*б*); результат применения маски к встроенному шрифту (*в*)



Рис. 13.13. Эффект линзы с применением динамического маскирования

В приведенном примере данные экземпляры создаются динамически с помощью метода `attachMovie()` и им присваиваются имена `mapSmall` и `mapLarge` соответственно. Увеличение размера клипа выполняется с помощью объекта инициализации, передаваемого методу `attachMovie()` в качестве параметра. Поскольку точка регистрации клипа, содержащего изображение карты, находится в его левом верхнем углу, применение дополнительного кода для выравнивания изображений не требуется.

Клип, изображающий линзу, имеет имя `lens` и содержит на своей монтажной линейке вложенный клип с именем `mask`, форма которого совпадает с формой стекла линзы. Данный клип будет выполнять роль маски для клипа `mapLarge`, сквозь которую зритель сможет видеть увеличенное изображение карты. Поскольку клип `mask` входит в состав клипа `lens`, при перемещении линзы маска также будет перемещаться.

Для того чтобы в центре линзы зритель видел ту же точку увеличенного изображения, над которой в данный момент находится линза, клип `mapLarge` необходимо смещать в направлении, противоположном направлению движения линзы, на расстояние, кратное масштабу увеличения. Зависимость между координатами клипа `lens` и координатами клипа `mapLarge` определяется следующим уравнением:

$$P_{mapLarge} = -P_{lens} \times (\text{scaleRatio} - 1),$$

где $P_{mapLarge}$ — координаты клипа `mapLarge`;

P_{lens} — координаты клипа `lens`;

scaleRatio — коэффициент увеличения.

Для того чтобы событие `onPress` могло быть обнаружено при нажатии на любую точку линзы, отверстие в изображении линзы заполняется прозрачным цветом. Такого заполнения не будет видно, однако оно позволяет привести форму области реагирования линзы в соответствие с ее видимыми очертаниями.

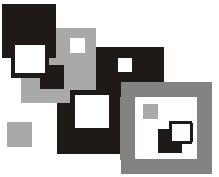
Конечный программный код приведен в листинге 13.6.

Листинг 13.6. Эффект линзы с применением динамического маскирования

```
var scaleRatio:Number = 1.5; // Коэффициент масштабирования
// Помещаем на сцену исходную картинку
this.attachMovie("map", "mapSmall", this.getNextHighestDepth());
// Помещаем на сцену увеличенную копию картинки
this.attachMovie("map", "mapLarge", this.getNextHighestDepth(),↳
{_xscale:scaleRatio * 100, _yscale:scaleRatio * 100});
/* Помещаем на сцену клип, изображающий линзу, на линейке которого
находится клип mask, предназначенный для исполнения роли маски */
this.attachMovie("lens", "lens", this.getNextHighestDepth(), {_x:200,↳
_y:200});
/* Позиционируем увеличенную картинку в соответствии с местоположением
линзы. Это необходимо сделать для того, чтобы еще до перемещения линзы
видимое в ней увеличенное изображение находилось в соответствии с
исходным местоположением линзы */
```

```
mapLarge._x = -lens._x * (scaleRatio - 1);
mapLarge._y = -lens._y * (scaleRatio - 1);
// Маскируем увеличенную картинку клипом mask
mapLarge.setMask(lens.mask);
/* Организация управления линзой и смещения увеличенной картинки в
соответствии с местоположением линзы */
lens.onPress = function() {
       this.onMouseMove = function() {
              this._x = this._parent._xmouse;
              this._y = this._parent._ymouse;
              mapLarge._x = -this._x * (scaleRatio - 1);
              mapLarge._y = -this._y * (scaleRatio - 1);
              updateAfterEvent();
       };
};
// Прекращение перетаскивания линзы при отпускании кнопки мыши
lens.onRelease = function() {
       delete this.onMouseMove;
};
```

Глава 14



Применение динамического и пользовательского текста

Если вам непонятно какое-то слово в техническом тексте,
не обращайте на него внимания. Текст полностью сохраняет смысл и без него.

Закон Купера

Текст является одним из ключевых элементов мультимедийного проекта, обеспечивая его информационное наполнение и способствуя взаимодействию с пользователем. ActionScript предоставляет широкие возможности управления текстовым содержимым Flash-фильма, которые позволяют решить большинство практических задач. К числу таких задач можно отнести, например, вывод текстовых сообщений в качестве реакции на те или иные действия пользователя или процессы, связанные с воспроизведением фильма, обработку текстовой информации, введенной пользователем, создание динамических текстовых эффектов и т. д.

Flash MX 2004 поддерживает стандарт кодирования текста Unicode. Данный стандарт, разработанный ассоциацией Unicode в 1991 году, позволяет охватить символы большинства мировых языков, включая знаки пунктуации, математические и технические символы и т. д. Проигрыватель Flash Player 7 поддерживает 8-битную и 16-битные (UTF-16 BE, Big Endian и UTF-16 LE, Little Endian) кодировки Unicode. Формат UTF-8 использует для кодирования символов последовательности байтов различной длины, которые преобразуются в коды Unicode. UTF-16 — это 16-битная кодировка, которая кодирует каждый символ, используя двухбайтовую последовательность. При разработке Flash-проектов рекомендуется использовать формат кодирования UTF-8. Формат Unicode позволяет отображать текст на различных языках при условии, что в системе установлены соответствующие шрифты.

Большинство средств программной обработки текста, существующих в ActionScript, применимо исключительно к динамическому (Dynamic) и пользовательскому (Input) тексту. В связи с этим при упоминании текстового блока здесь и в дальнейшем мы будем подразумевать текстовый блок типа Dynamic или Input. Когда речь пойдет об использовании статического текста (Static), это будет либо ясно из контекста, либо отмечено особо.

Примечание

Следует иметь в виду, что статический текстовый блок, для которого в панели **Properties** включается режим **Selectable** (Выделяемый), в действительности реализуется как динамический текстовый блок с применением HTML-форматирования. Этот блок получает идентификатор (имя экземпляра), который генерируется автоматически и состоит из слова "instance" и порядкового номера.

В отличие от статического текста, который создается исключительно в рабочей среде, динамический или пользовательский текстовые блоки могут быть созданы как вручную (в рабочей среде), так и динамически с помощью ActionScript. Форматирование текстовых блоков также может выполняться либо вручную в рабочей среде, либо динамически.

Динамический или пользовательский текстовый блок, вне зависимости от того, создан он вручную или программным образом, представляет собой объект, являющийся экземпляром встроенного класса `TextField`. Как и любой другой объект, текстовый блок должен быть снабжен уникальным идентификатором, позволяющим осуществлять обращение к нему и динамическую обработку его содержимого. Идентификатором текстового блока (как и для клипов с кнопками) служит имя экземпляра (Instance Name). Если имя экземпляра не задается разработчиком, оно генерируется автоматически. Используя имя экземпляра текстового блока, можно программным образом осуществлять полный контроль его внешнего вида и содержимого, используя свойства, методы и обработчики событий ряда встроенных классов, описание которых приводится в настоящей главе.

Наряду с присвоением имени текстовому блоку может быть сопоставлена переменная (variable). С помощью этой переменной можно выводить и извлекать текстовое содержимое динамического блока. Значение переменной, сопоставленной текстовому блоку, выводится в нем в виде строки. При вводе данных в пользовательский текстовый блок соответствующая строка присваивается переменной, сопоставленной данному текстовому блоку. В большинстве случаев для управления содержимым текстового блока удобнее использовать имя экземпляра и возможности встроенных классов, а не переменную.

Создание и настройка параметров текстового блока в рабочей среде

Создание динамического или пользовательского текстового блока в рабочей среде осуществляется с помощью инструмента Text Tool так же, как и создание статического текстового блока. Однако в отличие от последнего, маркер динамического или пользовательского текстового блока располагается в правом *нижнем* углу, а невыделенный текстовый блок отображается на сцене пунктирной рамкой. При создании динамического текстового блока можно строго задать длину строки и высоту блока. При этом если в процессе воспроизведения текст, помещаемый в данный блок сценарием ActionScript, превышает размер текстового блока, текст будет срезан его границами.

Поскольку в динамический текстовый блок информация выводится программным образом, а содержимое пользовательского текстового блока создается либо пользователем, либо программно (как и в случае динамического текста), в рабочей среде текстовый блок, как правило, остается пустым. Таким образом, создание текстового блока в рабочей среде предполагает только задание его размера и настройку разнообразных параметров, используемых при отображении текста. Само текстовое наполнение помещается в текстовый блок уже в процессе воспроизведения фильма.

Настройка параметров текстового блока в рабочей среде

Для форматирования текстового блока в рабочей среде его необходимо выделить и выполнить необходимые настройки в панели **Properties** или с помощью соответствующих команд раздела **Text** главного меню. Параметры настройки и форматирования динамического и пользовательского текста практически идентичны. На рис. 14.1. изображена панель **Properties** с настройками параметров динамического текстового блока.



Рис. 14.1. Параметры динамического текста. Панель **Properties**

Ниже представлен перечень всех параметров настройки динамического и пользовательского текстовых блоков, которые могут быть заданы в рабочей среде:

- выпадающий список **Text Type** (Тип текстового блока) — позволяет определить тип текстового блока: **Static** (Статический), **Dynamic** (Динамический) или **Input** (Пользовательский);
- поле **Instance Name** (Имя экземпляра) — данное поле используется для задания имени экземпляра текстового блока. Имя экземпляра является уникальным идентификатором текстового блока. Используя имя экземпляра, можно программно манипулировать данным текстовым блоком, контролировать его содержимое и внешний вид. Имя экземпляра должно быть задано в соответствии с правилами именования идентификаторов;
- поля **Width**, **Height**, **X-location**, **Y-location** (Размеры текстового блока, Координаты текстового блока) — с помощью этих полей можно задать размеры и координаты текстового блока. Следует иметь в виду, что изменение размеров приводит к трансформации текстового блока как объекта наложенного уровня вместе с его содержимым. В результате масштабирования пропорции шрифта могут быть искажены. Масштабирование текстового блока не приводит к изменению значения параметра **Font Size** (Кегль), изменяется только размер текстового блока (вместе со шрифтом). Для того чтобы отменить все произведенные с текстовым блоком трансформации, нужно выполнить команду **Modify>Transform>Remove transform** или нажать кнопку **Reset** панели **Transform**;
- выпадающий список **Font** (Шрифт) — позволяет выбрать любой из установленных в системе шрифтов, используемый для вывода текста;
- выпадающий список **Font Size** (Кегль) — размер символов шрифта, задаваемый в пунктах (один пункт равен 0,35 мм). Допустимые значения лежат в диапазоне от 1 до 2500 пунктов. В некоторых случаях шрифт определенного размера в Flash выглядит мельче, чем шрифт, набранный тем же кеглем в другом приложении;
- цветовой образец **Font Color** (Цвет) — щелчок на цветовом образце позволяет задать для текста любой цвет, содержащийся в текущем каталоге цветов. Текст может быть окрашен только сплошным цветом (solid);
- кнопка **Bold style** (Жирное начертание) — использование полужирного начертания (выделенный текст);
- кнопка **Italic style** (Курсивное начертание) — использование курсивного начертания;
- кнопки выравнивания абзаца — данные параметры идентичны соответствующим настройкам текстового редактора (например, Microsoft Word):
  - **Align Left** — выключка по левому краю;
  - **Align Center** — выключка по центру;
  - **Align Right** — выключка по правому краю;
  - **Justify** — выключка по формату.
- кнопка **Alias text** (Сглаживание) — эта кнопка имитирует растрирование шрифта всего текста (или отдельных выделенных символов) и применяется при работе с текстом маленького размера, делая его более четким и читабельным. В Flash сглаживание для шрифтов включено по умолчанию, и это приводит к тому, что

текст небольшого размера выглядит размытым и плохо читается. При активации режима **Alias text** очертания границ символов шрифта формируются из прямоугольников, что напоминает эффект растрирования и приводит к появлению характерной ступенчатости. Однако в результате применения этого режима некоторые символы шрифта могут претерпеть существенные изменения и утратить свою оригинальную форму. Стоит иметь в виду, что шрифты с засечками (serif) при маленьких размерах отображаются хуже, чем рубленые шрифты без засечек (sans serif); кроме того, курсивное или полужирное начертание также приводит к снижению читаемости текста;

- кнопка **Format** (Форматирование абзаца) — нажатие на эту кнопку приводит к появлению диалогового окна **Format Options**. Здесь имеются следующие параметры:
  - **Indent** — отступ абзаца (красная строка);
  - **Line spacing** — интерлиньяж (межстрочное расстояние);
  - **Left margin** — левое поле;
  - **Right margin** — правое поле;
- выпадающий список **Line type** (Параметры строки) — используется для указания параметров переноса строки:
  - **Single Line** — текст выводится в одну строку;
  - **Multiline** — перенос строки осуществляется автоматически при достижении границы тестового блока. При выборе этого значения разбиение на строки будет выполняться автоматически в зависимости от ширины текстового блока и длины строки;
  - **Multiline no wrap** — перенос строки осуществляется только при использовании символа перевода строки (клавиша <Enter>) или соответствующей Escape-последовательности ("\r", "\n");
  - **Password** — это значение присутствует только для пользовательского текстового блока. В результате его выбора символы в пользовательском текстовом блоке заменяются звездочками (*). Такая возможность обычно применяется для ввода паролей;
- кнопка **Selectable** (Выделяемый текст) — этот режим используется только для динамического или статического текста. Активизация данного режима позволит пользователю выделять динамический текст в конечном фильме и, соответственно, копировать его в буфер с помощью команды **Copy**. Если режим **Selectable** отключен, пользователь не сможет получить доступ к содержимому соответствующего динамического текстового блока. Содержимое пользовательского текстового блока может быть выделено всегда;
- кнопка **Render text as HTML** (Поддержка HTML) — включение данного режима активизирует режим поддержки HTML. Это позволяет программно форматировать текст, используя некоторые теги HTML, поддерживаемые проигрывателем Flash Player;
- кнопка **Show border around text** (Рамка вокруг текстового блока, белый фон) — включение этого режима приводит к тому, что текст в текстовом блоке будет выводиться на белом непрозрачном фоне, а сам блок будет заключен в черную рам-

ку. По умолчанию текстовый блок не имеет фона (то есть он прозрачен) и рамки. Задать произвольные цвета для фона и рамки можно только программными средствами;

- **URL link** (Текстовая гиперссылка) — данное поле позволяет назначить гиперссылку на все содержимое текстового блока или конкретный выделенный фрагмент текста. Эта опция доступна только для динамического текста при условии, что режим поддержки HTML-форматирования активен. Для этого в поле **URL link** необходимо ввести URL с указанием протокола (например, **http://www.avalon.ru**). Для того чтобы создать ссылку на адрес электронной почты, необходимо использовать ключевое слово **mailto:** (например, **mailto:book_flash@mail.ru**). Текстовая гиперссылка в рабочей среде Flash подчеркивается пунктирной линией, но в конечном фильме, в отличие от традиционной HTML-ссылки, автоматически никак особо не выделяется. При наведении на текст, содержащий гиперссылку, в конечном фильме курсор принимает вид руки.
- **Target** (Целевое окно или фрейм) — данное поле становится активным при создании гиперссылки. С помощью выпадающего списка можно указать, в какое окно или фрейм (при фреймовой структуре) будет загружен загружаемый ресурс:
  - **_blank** — в новое окно браузера;
  - **_parent** — в родительский по отношению к текущему фрейм;
  - **_self** — в текущий фрейм текущего окна;
  - **_top** — в самый верхний фрейм фреймовой структуры текущего окна.

  В данное поле также можно ввести пользовательское имя фрейма, в который будет осуществлена загрузка.
- поле **Var** (Переменная) — в это поле вводится имя переменной, сопоставленной данному блоку. Строковое значение этой переменной будет выводиться в качестве содержимого данного текстового блока. Символы, введенные пользователем в редактируемый текстовый блок, формируют строку, которая присваивается данной перемененной в качестве значения. Переменная должна быть названа в соответствии с правилами именования переменных;
- кнопка **Character** (Встраивание очертаний символов шрифта) — нажатие этой кнопки приводит к появлению диалогового окна **Character Options** (Параметры символов), которое позволяет указать, очертания каких символов должны быть встроены в конечный фильм. Окно **Character Options** содержит следующие параметры:
  - **No Characters** — не встраивать очертания шрифта. В этом случае объем конечного файла будет минимален, но в случае отсутствия используемого шрифта в системе пользователя текст будет отображен с помощью наиболее близкого к нему машинно-независимого шрифта (device font). Если символы шрифта не встраиваются, то к ним не будет применяться сглаживание;
  - **Specify Ranges** — данная опция позволяет выбрать любой диапазон символов (или несколько с помощью клавиш <Shift> или <Ctrl>), из представленных в расположенном здесь списке. Каждая строка содержит название диапазона, начальный и конечный символы (или слово **All**, означающее все символы) и общее число символов данного диапазона;

Примечание

При желании читатель сам может определить, какие диапазоны символов должны выводиться в окне **Character Options**. Для этого необходимо внести изменения в файл UnicodeTable.xml, расположенный примерно по следующему адресу: **C:\WINDOWS\Application Data\Macromedia\Flash MX 2004\en\Configuration\FontEmbedding**. Для указания требуемого диапазона символов необходимо присвоить атрибутам `min` и `max` тега `<range>` значения, соответствующие кодовой позиции Unicode начального и конечного символов диапазона. Если символы следуют не подряд, необходимо использовать несколько тегов `<range>` для каждой последовательности. Так, для того чтобы в окне **Character Options** были указаны диапазоны, соответствующие прописным и строчным буквам русского алфавита, можно добавить в файл UnicodeTable.xml следующие элементы (следует иметь в виду, что в Unicode буквы "ё" и "Ё" стоят особняком от остальных символов русского алфавита):

```
<glyphRange name="Russian Uppercase " id="10" >
<range min="0x0410" max ="0x042F" />
<range min="0x0401" max ="0x0401" />
</glyphRange>

<glyphRange name="Russian Lowercase" id="11" >
<range min="0x0430" max ="0x044F" />
<range min="0x0451" max ="0x0451" />
</glyphRange>
```

- **Include these characters** — в это поле можно вручную ввести произвольный набор символов, включая знаки пунктуации и специальные знаки, которые будут встроены в конечный фильм. Знаки вводятся подряд без разделителей. Если один и тот же набор символов используется в нескольких динамических или пользовательских текстовых блоках, его достаточно встроить один раз;
- **Auto Fill** — эта кнопка позволяет автоматически заполнить поле **Include these characters**, символами, которые введены в соответствующий текстовый блок, расположенный на сцене. Так, при необходимости встроить специфический фиксированный набор символов можно вручную ввести их в текстовый блок и затем воспользоваться данной кнопкой. После этого текстовый блок можно очистить;
- **Total number of glyphs** — здесь указывается общее количество встраиваемых символов;

❒ поле **Maximum Characters** используется только для пользовательского текстового блока. Оно позволяет ограничить количество вводимых в текстовое поле символов, указав максимальное допустимое число. Например, если пользовательский текстовый блок предназначен для указания года рождения, очевидно, что для этого нужно отвести не более четырех символов. Нулевое значение соответствует неограниченному количеству символов в текстовом поле.

Встроенные и машинно-независимые шрифты

При работе с динамическим или пользовательским текстом у разработчика имеется выбор: встраивать очертания используемого шрифта или нет. Оба варианта имеют свои плюсы и минусы, поэтому необходимо четко понимать необходимость встраивания или последствия того, что шрифт не будет встроен в конечный документ.

Если очертания символов используемого шрифта не будут встроены в конечный документ, это приведет к следующему.

- Использование текста в минимальной степени отразится на объеме конечного документа. Произойдет лишь несущественное увеличение объема даже при использовании большого количества различных текстовых символов. Эта особенность является наиболее сильным аргументом за отключение встраивания шрифтов.
- Если шрифт не встроен, к нему не будет применяться сглаживание. Отсутствие сглаживания выражается в том, что буквы будут иметь ступенчатые неровные границы. Особенно сильно это заметно при использовании курсивов и шрифтов с засечками.
- Если у конечного пользователя в системе отсутствует используемый разработчиком шрифт, то вместо этого шрифта будет применяться ближайший к нему системный шрифт (device font). Это может привести к тому, что внешний вид и даже размеры текста у конечного пользователя могут существенно отличаться от оригинала.
- Текст нельзя трансформировать, используя операции поворота, скоса, отражения. Выполнение этих операций приведет к исчезновению невстроенных символов шрифта. Применение некоторых свойств, поддерживаемых классом `TextField`, не будет оказывать влияния на текстовый блок. К числу таких свойств относится, например, свойство `_alpha`, позволяющее изменять прозрачность текстового блока.

С другой стороны, встраивание очертаний шрифта будет иметь следующие последствия.

- Объем документа будет увеличен на тот объем, который занимает встраиваемый шрифт или отдельные встраиваемые символы этого шрифта. При этом следует иметь в виду, что большинство шрифтов имеют четыре модификации, отличающиеся типом начертания: обычное (regular), курсивное (italic), жирное (bold) и курсивное жирное (bold italic). Если при встраивании шрифта применяются все четыре типа начертания, то объем информации, необходимой для описания шрифта, увеличивается в четыре раза.
- К встроенным шрифтам применяется сглаживание. В связи с этим их границы выглядят ровно и аккуратно. Правда, при использовании шрифтов маленького размера (менее 10 пунктов) применение сглаживания может сильно снизить читабельность текста, поскольку маленькие буквы будут расплываться и выглядеть нечетко.
- Встроенный шрифт всегда корректно отображается на компьютере конечного пользователя, даже если этот шрифт отсутствует у него в системе.

- Встраивание очертаний отдельных символов шрифта для пользовательского текста приведет к тому, что в соответствующий текстовый блок можно будет ввести только встроенные символы. Таким образом, при работе с пользовательским текстом необходимо либо встроить весь набор символов, которые потенциально могут быть введены в текстовое поле, либо не встраивать очертания шрифта вообще. В последнем случае для вывода текста будет использоваться несглаженный шрифт из системы пользователя или машинно-независимый шрифт.

Поясним значение термина "машинно-независимый шрифт". В Flash имеется три таких шрифта. Они располагаются в начале списка доступных шрифтов и их названия начинаются с символа нижнего подчеркивания "_":

- _sans;
- _serif;
- _typewriter.

Каждый из указанных шрифтов соответствует одной из наиболее широко используемых гарнитур (семейств шрифта).

Шрифт _sans соответствует гарнитуре Arial, шрифт _serif соответствует гарнитуре Times New Roman, шрифт _typewriter соответствует гарнитуре Courier.

Применение этих шрифтов позволяет избежать встраивания очертаний шрифтов, уменьшая тем самым конечный объем файла. Машинно-независимые шрифты не сглаживаются, поэтому при их использовании в конечном фильме символы имеют ступенчатые неровные границы. Отключение сглаживания делает шрифт более разборчивым при небольших размерах (менее 10 пунктов).

При использовании машинно-независимых шрифтов к тексту не могут быть применены операции трансформации (вращение, скос), поскольку это приведет к исчезновению текста в конечном фильме. При масштабировании машинно-независимого шрифта изменяется только размер символов, пропорции же остаются неизменными.

Машинно-независимый шрифт не может быть маскирован в рабочей среде с помощью слоя маски. Для того чтобы использовать эффект маски применительно к текстовому блоку, использующему машинно-независимый шрифт, необходимо создавать маску программно с помощью метода `setMask()`.

Использование текстового блока, созданного в рабочей среде

Как было сказано выше, вывод и обработку содержимого текстового блока можно осуществлять либо с помощью сопоставленной ему переменной, либо используя возможности класса `TextField`.

Переменная, которая сопоставляется текстовому блоку, имеет строковый тип, и ее значение всегда представляет собой строку, состоящую из символов, помещенных в данный текстовый блок. С помощью переменной можно осуществлять как вывод текста, так и обработку текста, введенного пользователем.

Рассмотрим два простых примера.

1. Создадим на сцене динамический текстовый блок и сопоставим ему переменную `txt`, указав ее имя в поле **var** панели **Properties**. В этот же кадр поместим следующую строку кода:

   ```
   txt = "Привет, Flash !";
   ```

   При просмотре конечного фильма в текстовом блоке будет выведено сообщение "Привет, Flash !"

2. Создадим на сцене два текстовых блока: пользовательский и динамический. Динамическому блоку сопоставим переменную с именем `greetingTxt`, а пользовательскому — переменную с именем `nameTxt`. Здесь же рядом создадим клип с именем `submit`. Поместим в кадр следующий сценарий:

   ```
   submit.onPress = function() {
       greetingTxt = "Привет, " + nameTxt + "!";
   };
   ```

   Если теперь в конечном фильме в пользовательское поле ввести свое имя, а затем нажать на кнопку `submit`, то в динамическом поле появится приветствие.

Используя переменную, можно управлять только содержимым текстового блока. Для того чтобы контролировать многочисленные параметры, управляющие не только содержанием, но также внешним видом, положением и состоянием текстового блока, необходимо использование встроенных методов и свойств, доступ к которым можно получить с помощью имени экземпляра текстового блока.

Управление текстовым блоком, созданным в рабочей среде, осуществляется точно так же, как и текстовым блоком, созданным динамически. Детальное описание этих возможностей приводится далее в настоящей главе.

Использование шрифтовых символов

При работе над большими проектами, состоящими из нескольких разделов, представляющих собой отдельные файлы SWF, приходится многократно использовать один или несколько шрифтов, встраивая их в каждый файл. Это приводит к увеличению объема файлов и, соответственно, к увеличению времени их загрузки из сети. В качестве альтернативы встраиванию шрифта в каждый файл проекта Flash предлагает возможность создания общего шрифтового ресурса, помещенного в отдельный SWF-документ-источник и загружаемого в соответствующий SWF-документ-получатель непосредственно в процессе его воспроизведения. Файл, содержащий общий шрифтовой ресурс (или несколько), должен быть загружен один раз, после чего к нему может быть осуществлен доступ со стороны всех остальных файлов проекта, использующих данный шрифт. При этом использование общего шрифта несколькими файлами практически не увеличивает их конечный объем. Общий шрифтовой ресурс может быть использован текстовым блоком любого типа (Dynamic, Input, Static). К тексту, который выводится с применением общего шрифтового ресурса, применяется сглаживание.

Для использования общего шрифтового ресурса необходимо выполнить две процедуры.

1. Объявление общего шрифтового ресурса (или нескольких) путем создания документа-источника, содержащего общую библиотеку (**Shared library**).
2. Включение общего шрифтового ресурса (или нескольких) в документы-получатели.

Объявление общего шрифтового ресурса осуществляется в рабочей среде Flash путем создания *шрифтового символа* (Font symbol). Для этого необходимо выполнить следующие действия.

1. В документе, который будет использоваться в качестве общей библиотеки шрифтов, открыть библиотеку документа (**Window>Library**) и из контекстного меню выбрать команду **New Font...**.
2. В появившемся окне **Font Symbol Properties** (рис. 14.2) в поле **Name** ввести название шрифтового символа (например, Pragmatica_symb), в списке **Font** выбрать шрифт, используемый в качестве общего шрифтового ресурса (например, Pragmatica). Установка флажков **Bold** или **Italic** позволяет указать начертание шрифта. Флажок **Alias text** отключает сглаживание шрифта. Один шрифтовой символ может включать описание только одного начертания шрифта (regular, bold или italic) или сочетание начертаний (например, bold и italic одновременно). При необходимости использования нескольких начертаний одного шрифта (например, отдельно bold и отдельно italic) нужно для каждого из них создавать отдельные шрифтовые символы. Задав все параметры, необходимо нажать на кнопку **OK**. В результате выполненных действий в библиотеке появится новый элемент, помеченный пиктограммой .
3. Задать параметры связи. Для этого щелкнуть правой кнопкой мыши по шрифтовому символу в библиотеке и из всплывающего контекстного меню выбрать команду **Linkage...** или, выделив шрифтовой символ, выбрать эту команду из контекстного меню библиотеки. В появившемся диалоговом окне **Linkage Properties** установить флажок **Export for runtime sharing.** В поле **Identifier** ввести имя, служащее уникальным идентификатором данного шрифта. В поле **URL** задается абсолютный или относительный путь к файлу SWF, содержащему библиотеку. Относительная ссылка указывается относительно местоположения файла-получателя, в котором должен быть использован общий шрифтовой ресурс. Если файл-получатель размещается на HTML-странице, то в поле **URL** должен быть указан путь к файлу, содержащему общую библиотеку, относительно данной страницы. Так, например, если файл, содержащий общую библиотеку, называется **prgm_lib.swf** и помещен в один каталог с HTML-страницей, содержащей файл, использующий общий шрифтовой ресурс, то в поле **URL** нужно ввести **prgm_lib.swf**. Если файл с общей библиотекой prgm_lib.swf будет помещен в отдельный каталог с названием fonts, который, в свою очередь находится в том же каталоге, что и HTML-страница, содержащая файл, использующий шрифт, то путь будет выглядеть следующим образом: **fonts/ prgm_lib.swf** и т. д.
4. Сохранить документ, содержащий шрифтовой символ, указав соответствующее имя. Если имя исходного документа с общей библиотекой, заданное при сохранении, будет совпадать с названием файла SWF, указанным в поле **URL** диалого-

вого окна **Linkage Properties** (в нашем случае **prgm_lib.fla**), то при публикации файл SWF с соответствующим именем будет сгенерирован автоматически. В противном случае необходимо будет задать имя результирующего SWF-файла в соответствии с именем, указанным в поле **URL**. Необходимо иметь в виду, что по умолчанию при публикации результирующий файл записывается в тот же каталог, где был сохранен исходный документ.

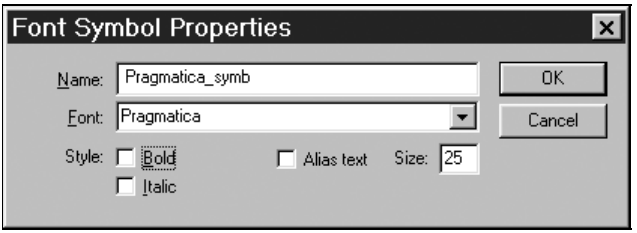


Рис. 14.2. Диалоговое окно **Font Symbol Properties** для создания шрифтового символа

5. Опубликовать исходный документ с общей библиотекой в формате SWF.

Для включения общего шрифтового ресурса в документ-получатель необходимо выполнить следующие действия.

1. Открыть (или создать) документ, в котором будет использован шрифт общей библиотеки **File>Open...** (**File>New...**).
2. Открыть библиотеку документа, содержащего общую библиотеку **File>Import> Open External Library...**. В рабочей среде откроется окно библиотеки внешнего файла.
3. Импортировать шрифтовой символ, перетащив его курсором из внешней библиотеки на сцену или в библиотеку текущего документа. При этом параметры связи будут заданы автоматически. В окне **Linkage Properties** должен быть установлен флажок **Import for runtime sharing**, поле **Identifier** должно содержать идентификатор, заданный на шаге 3 предыдущей процедуры, а в поле **URL** должен быть указан путь к общей библиотеке относительно месторасположения SWF-файла-получателя, полученного в результате публикации текущего документа.
4. Создать текстовый блок, выделить его и в списке доступных шрифтов выбрать шрифт с названием, указанным в поле **Name** диалогового окна **Font Symbol Properties**. В списке это название помечено звездочкой (например, Pragmatica_symb*). Ввести текст, задав необходимые параметры форматирования.
5. Задать тип текстового блока: динамический (Dynamic), пользовательский (Input) или статический (Static).
6. Параметры импортированного шрифтового символа, заданные при его создании, можно увидеть в окне **Font Symbol Properties**. Чтобы открыть это окно, необходимо либо дважды щелкнуть на пиктограмме шрифтового символа в библиотеке, либо воспользоваться командой **Properties...** контекстного меню самого символа или библиотеки. Следует иметь в виду, что в документе-получателе должно быть

использовано то же начертание шрифта, которое определено в шрифтовом символе. В противном случае вместо ресурса общей библиотеки в документе будет использован шрифт из системы, к которому не будет применяться сглаживание. При этом связь с общей библиотекой будет разорвана.

7. Сохранить и опубликовать документ **File>Publish**. Результирующий SWF-файл-получатель должен быть размещен в соответствующем каталоге таким образом, чтобы ссылка на документ-источник, заданная в поле **URL** окна **Linkage Properties**, разрешалась относительно его текущего местоположения.

Примечание

При использовании общего шрифтового ресурса следует отказаться от применения кириллических символов, поскольку в конечном фильме они отображаются некорректно или исчезают.

При размещении проекта в сети Internet необходимо поместить на сервер файл SWF, содержащий общую библиотеку и файлы, использующие общий шрифт.

После создания общей библиотеки и включения шрифтового символа во все файлы проекта достаточно изменить шрифт в файле общей библиотеки и опубликовать его, после чего соответствующий шрифт будет изменен во всех файлах-получателях.

Динамическое создание и форматирование текста

Для динамического создания текстового блока используется метод `createTextField()` класса `MovieClip`. Созданный в результате его применения текстовый блок представляет собой экземпляр класса `TextField`, предоставляющего возможности программного управления этим блоком. Динамически созданный текстовый блок помещается на заданный уровень в стек объекта, вызвавшего метод `createTextField()`. Все параметры, определяющие содержимое, внешний вид и состояние текстового блока, задаются программным образом. Метод имеет следующий синтаксис:

клип`.createTextField(`*имя*`, `*уровень*`, `*x*`, `*y*`, `*ширина*`, `*высота*`)`

Здесь *клип* — это объект, в стек которого будет помещен созданный текстовый блок. Таким объектом может быть клип или основная монтажная линейка, то есть `_root`.

Параметр *имя* — это строка, которая задает имя экземпляра создаваемого текстового блока. Имя экземпляра должно отвечать требованиям к идентификаторам, описанным в *главе 3*.

Параметр *уровень* указывает уровень стека, на который будет помещен текстовый блок. Данный параметр может задаваться числовым литералом или выражением, возвращающим число.

Параметры *x* и *y* задают координаты текстового блока относительно начала координат объекта *клип*. В качестве точки регистрации текстового блока используется ле-

вый верхний угол его ограничивающей рамки. Вне зависимости от значений этих параметров текстовый блок в дальнейшем может быть позиционирован с помощью свойств _x и _y.

Параметры *ширина* и *высота* задают размеры текстового блока. Дальнейшее применение свойств _width, _height, autoSize может привести к тому, что фактические размеры текстового блока будут отличаться от размеров, заданных с помощью этих параметров.

Текстовый блок, созданный в результате применения метода createTextField(), является пустым, то есть он не имеет содержимого. Следующая строка кода создает на ближайшем свободном уровне стека основной монтажной линейки пустой текстовый блок с именем txt, размером 100×50 пикселов:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 100, 50);
```

Следует иметь в виду, что адресация текстовых блоков выполняется точно так же, как и адресация клипов или кнопок. Если текстовый блок создан на монтажной линейке клипа, он существует в его локальной области видимости. При обращении к текстовому блоку с другой монтажной линейки необходимо в указании пути использовать идентификатор родительского объекта.

Класс *TextField*. Настройка параметров текстового блока

В классе TextField определен целый ряд свойств, позволяющих контролировать различные параметры текстового блока. Часть из них совпадает с рассмотренными ранее свойствами клипов и кнопок. К их числу относятся:

_x, _y, _width, _height, _xscale, _yscale, _alpha, _rotation, _visible, _name, _parent, _url, _target, menu.

Следует иметь в виду, что применение таких свойств, как _alpha, _rotation, а также _xscale и _yscale с отрицательными значениями (отражение), может иметь место только для встроенных шрифтов. Поскольку подробное описание этих свойств применительно к клипам уже было приведено ранее, здесь мы сосредоточим внимание на свойствах, которые являются специфичными для объектов TextField.

Примечание

Необходимо отметить, что применение свойств _width и _height позволяет управлять размерами текстового блока, однако не отражается на размерах отображаемого текста. Другими словами, данные свойства определяют размеры границ текстового блока, а не размеры его содержимого. Для выполнения масштабирования текстового блока вместе с его содержимым используются свойства _xscale и _yscale. Причем для выполнения масштабирования символов текста, а не только ограничивающей рамки текстового блока, необходимо использовать встроенные шрифты.

В качестве точки регистрации текстового блока используется левый верхний угол его ограничивающей рамки. Это необходимо принимать в расчет при работе с такими свойствами, как _x, _y, _width, _height, _xscale, _yscale, _rotation, _xmouse, _ymouse.

В данном разделе будут рассмотрены основные свойства класса TextField. Остальные свойства будут подробно описаны в других разделах этой главы в соответствующих контекстах.

Тип текстового блока. Свойство *type*

Для указания типа текстового блока используется свойство type. Данное свойство принимает одно из двух строковых значений: "dynamic" или "input". Использование значения "dynamic" определяет динамический текстовый блок, содержимое которого не может быть изменено пользователем. Значение "input" определяет пользовательский текстовый блок, содержимое которого может быть введено пользователем или программным образом.

По умолчанию свойство type принимает значение "dynamic".

Текст. Свойство *text*

Для того чтобы вывести в текстовое поле содержимое, можно воспользоваться свойством text. Данное свойство принимает строковое значение, соответствующее содержимому текстового блока. Данное свойство можно использовать как для чтения, так и для записи. Для того чтобы вывести текст в текстовый блок, свойству text этого блока нужно присвоить соответствующее строковое значение. При необходимости обработки содержимого текстового блока нужно извлечь значение данного свойства.

Следующий код создает пустой текстовый блок и выводит в него текстовое сообщение:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 100, 20);
txt.text = "Привет, Flash";
```

При необходимости обработки содержимого текстового блока нужно извлечь значение свойства text. Следующая строка кода сохраняет содержимое текстового блока в переменной.

```
var message:String = txt.text;
```

Применение свойства text позволяет выполнить ту же задачу, что и сопоставление переменной текстовому блоку.

Текстовый блок, создаваемый с помощью метода createTextField(), по умолчанию является динамическим, то есть имеет тип Dynamic. По умолчанию для вывода текста используется шрифт Times New Roman черного цвета и кегль 12. Для последующего форматирования текста используются возможности классов TextField и TextFormat, которые описываются далее в этой главе.

Переменная текстового блока. Свойство *variable*

Как уже было сказано ранее, текст, который содержится в текстовом блоке, может выводиться и обрабатываться с помощью переменной, сопоставленной данному блоку. Для сопоставления переменной текстовому блоку в рабочей среде достаточно ввести ее имя в поле **var** панели **Properties**. Для динамического сопоставления переменной текстовому блоку используется свойство variable.

В качестве значения данное свойство принимает строку, содержащую имя переменной, которая должна быть сопоставлена текстовому блоку. Значение данной переменной всегда представляет собой строку содержимого текстового блока. Для вывода текста в текстовый блок достаточно присвоить соответствующей переменной требуемое значение. И наоборот, для получения содержимого текстового блока достаточно извлечь значение сопоставленной ему переменной. Следующий код создает текстовый блок и выводит в него строку приветствия, используя динамически сопоставленную данному блоку переменную:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 150, 50);
txtVar = "Привет, Flash!";
txt.variable = "txtVar";
```

Этот же результат может быть получен с помощью свойства `text`:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 150, 50);
txt.text = "txtVar";
```

По умолчанию свойство `variable` принимает значение `null`.

Цвет текста. Свойство *textColor*

Свойство `textColor` позволяет изменить цвет текста, используемый по умолчанию. Данное свойство принимает числовое значение, которое, как правило, задается шестнадцатеричным числом. Свойство `textColor` может быть использовано как для чтения, так и для записи. По умолчанию данное свойство принимает значение 0x000000 (черный цвет).

Следующий фрагмент кода создает динамический текстовый блок и выводит в него текст, используя красный цвет:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 100, 20);
txt.text = "Привет, Flash";
txt.textColor = 0x0000FF;
```

Фон. Свойства *background* и *backgroundColor*

Текстовый блок вне зависимости от того, создан он в рабочей среде или динамически, по умолчанию не имеет фона. Таким образом, текстовый блок прозрачен, и сквозь промежутки между буквами можно видеть нижележащие объекты. В рабочей среде можно задать только белый непрозрачный фон. Для управления наличием и цветом фона можно воспользоваться встроенными свойствами `background` и `backgroundColor`. Данные свойства могут применяться как для чтения, так и для записи.

Свойство `background` принимает булево значение `true` или `false` и определяет наличие фона у текстового блока. Если это свойство принимает значение `true`, текстовый блок имеет непрозрачный фон, цвет которого задается свойством `backgroundColor`. Если свойство `background` принимает значение `false`, фон у текстового блока отсутствует. По умолчанию свойство `background` принимает значение `false`. Если значение этого свойства устанавливается в `true`, в качестве цвета

фона по умолчанию используется белый цвет (0xFFFFFF). Значение свойства background может быть изменено в любой момент, изменения сразу же вступают в силу.

Для указания цвета фона используется свойство backgroundColor, принимающее числовое значение, которое обычно задается шестнадцатеричным числом с помощью числового литерала или выражения, возвращающего число. Для того чтобы результат применения этого свойства стал виден, необходимо, чтобы свойство background имело значение true. Следующий код создает текстовый блок с оранжевым фоном:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 100, 20);
txt.text = "Привет, Flash";
txt.textColor = 0x0000FF;
txt.background = true;
txt.backgroundColor = 0xFFCC00;
```

Цвет фона может быть задан или изменен вне зависимости от значения свойства background. Однако для того, чтобы фон стал виден, свойство background должно быть установлено в значение true.

В следующем фрагменте программного кода фон текстового блока появляется при наведении курсора на кнопку с именем highlightBut и отключается при уходе курсора с этой кнопки.

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 100, 20);
txt.text = "Привет, Flash";
txt.textColor = 0x0000FF;
txt.backgroundColor = 0xFFCC00; // Задаем цвет фона
highlightBut.onRollOver = function(){
txt.background = true; // Включаем фон
};
highlightBut.onRollOut = function(){
txt.background = false; // Отключаем фон
};
```

Рамка. Свойства *border* и *borderColor*

По умолчанию текстовый блок не имеет рамки. Для включения рамки и выбора ее цвета используются свойства border и borderColor. Данные свойства могут применяться как для чтения, так и для записи.

Свойство border принимает булевы значения true или false и управляет отображением рамки. Если свойство border установлено в значение true, текстовый блок ограничен рамкой. Если свойство border имеет значение false, рамка отсутствует. По умолчанию свойство принимает значение false. Если данное свойство принимает значение true, по умолчанию в качестве цвета рамки используется черный цвет (0x000000).

Цвет рамки текстового блока задается с помощью свойства `borderColor`, принимающего числовое значение, которое обычно задается шестнадцатеричным числом с помощью числового литерала или выражения, возвращающего число. Для того чтобы рамка была видна, свойство `border` должно принимать значение `true`.

Следующий код создает текстовый блок с оранжевым фоном и серой рамкой:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 100, 20);
txt.text = "Привет, Flash";
txt.textColor = 0x33333;
txt.background = true;
txt.backgroundColor = 0xFFCC00;
txt.border = true;
txt.borderColor = 0x33333;
```

Перенос строк и подгонка размеров текстового блока. Свойства *wordWrap*, *multiline* и *autoSize*

Поскольку размер текстового блока жестко задается во время его создания, при размещении в этом блоке текстового содержимого может потребоваться выполнить автоматический перенос строк, так чтобы текст занял предоставленную ему область. В другом случае может возникнуть необходимость изменить размеры текстового блока, подстроив его под выводимое в нем текстовое содержимое. Для решения этих задач используются свойства `wordWrap`, `multiline` и `autoSize`.

Свойство *wordWrap*

Свойство `wordWrap` позволяет выполнять в текстовом блоке автоматический перенос текста на следующую строку. Данное свойство в качестве значения принимает булево значение `true` или `false`. Если свойство `wordWrap` принимает значение `true`, выполняется автоматический перенос символов, выходящих за правую границу текстового блока. В результате этого текст не будет срезаться по ширине текстового блока. Правда, перенос не гарантирует от того, что высота текстового блока позволит отобразить весь выводимый в нем текст. Если высота текстового блока меньше, чем это необходимо для отображения всех строк текста, часть текста будет срезана нижней границей.

Установка свойства `wordWrap` в значение `true` не приводит к тому, что в строку, выводимую в текстовом блоке, добавляются служебные символы (например, символ переноса строки).

Если свойство `wordWrap` принимает значение `false`, автоматический перенос строки выполняться не будет.

Свойство действует одинаково применительно и к динамическому, и к пользовательскому текстовым блокам. По умолчанию свойство `wordWrap` принимает значение `false`.

Свойство *multiline*

Свойство `multiline` определяет, может ли пользователь при вводе текста в пользовательский текстовый блок выполнять перенос строки. Данное свойство принимает булево значение `true` или `false` и применяется в основном только для текстовых блоков типа Input. Если свойство `multiline` принимает значение `true`, пользователь при вводе текста может выполнять перевод строки с помощью клавиши <Enter>. Таким образом, значение `true` позволяет вводить в текстовое поле многострочный текст. Если свойство `multiline` установлено в `false`, применение клавиши <Enter> во время ввода текста не дает никакого результата, текст вводится в одну строку.

Следует иметь в виду, что при установке свойства `wordWrap` в значение `true` перенос символов на новую строку в динамическом или пользовательском текстовом блоке будет выполняться вне зависимости от значения свойства `multiline`. Таким образом, свойство `multiline` просто позволяет ограничить пользователя в возможности переноса строки по его желанию. По умолчанию свойство `multiline` принимает значение `false`.

Свойство *autoSize*

Данное свойство выполняет автоматическую подгонку размеров текстового блока под выводимое в нем содержимое. Результат, оказываемый применением данного свойства, также зависит от значения свойства `wordWrap`.

Свойство `autoSize` может принимать одно из четырех строковых значений: `"none"`, `"left"`, `"center"`, `"right"` или булевы значения `true` и `false`. Ниже приводится описание этих значений. Результат их применения иллюстрируется рис. 14.3.

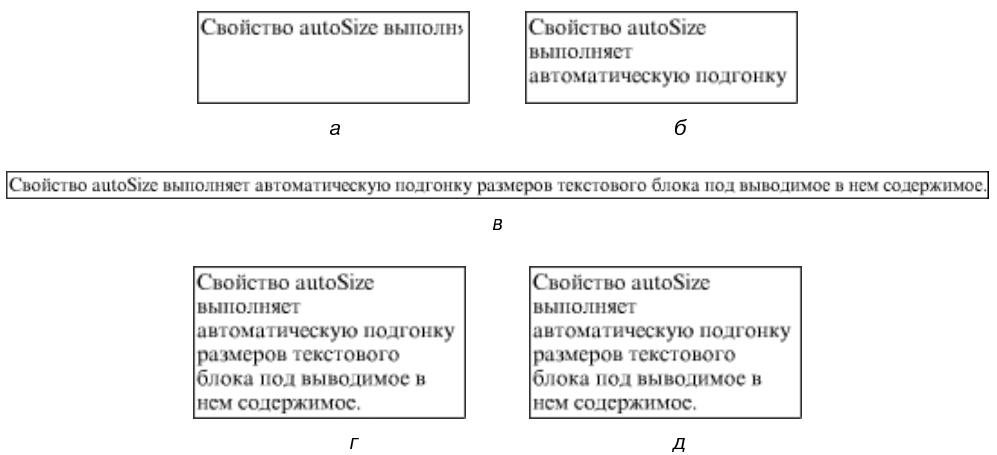


Рис. 14.3. Применение свойства:
а — `autoSize = "none", wordWrap = false`; *б* — `autoSize = "none", wordWrap = true`; *в* — `autoSize = "left", wordWrap = false`; *г* — `autoSize = "left", wordWrap = true`; *д* — `autoSize = "center", wordWrap = true`

- "none" — изменение размеров текстового блока не выполняется. Данное значение используется по умолчанию. Эквивалентом данного значения является булево значение false.
- "left" — при установке данного значения левая граница текстового блока остается неизменной, а правая граница смещается так, чтобы весь текст, расположенный в текстовом блоке был виден. Если текст содержит символ перевода строки ("\r", "\n"), нижняя граница текстового блока тоже будет смещена так, чтобы все строки были видны. Эквивалентом данного значения является булево значение true.

  Если значение свойства wordWrap установлено в true, левая и правая границы текстового блока остаются неизменными, а нижняя граница будет смещена так, чтобы все содержимое текстового блока было видно.
- "center" — при установке данного значения обе границы текстового блока равномерно смещаются так, чтобы весь расположенный в нем текст был виден. Если текст содержит символ перевода строки ("\r", "\n"), нижняя граница текстового блока тоже будет смещена так, чтобы все строки были видны.

  Если значение свойства wordWrap установлено в true, левая и правая границы текстового блока остаются неизменными, а нижняя граница будет смещена так, чтобы все содержимое текстового блока было видно.
- "right" — при установке данного значения правая граница текстового блока остается неизменной, а левая граница смещается так, чтобы весь текст, расположенный в текстовом блоке, был виден. Если текст содержит символ перевода строки ("\r", "\n"), нижняя граница текстового блока тоже будет смещена так, чтобы все строки были видны.

  Если значение свойства wordWrap установлено в true, левая и правая границы текстового блока остаются неизменными, а нижняя граница будет смещена так, чтобы все содержимое текстового блока было видно.

Следует отметить, что применение свойства autoSize не приводит к выравниванию текста внутри текстового блока, а определяет, каким образом текстовый блок будет изменять свой исходный размер, для того чтобы отобразить весь помещенный в него текст. Для выполнения выравнивания текста внутри текстового блока используется свойство align класса TextFormat.

Выделение текста. Свойство *selectable*

Свойство selectable позволяет указать, может ли текст, находящийся в текстовом блоке, быть выделен пользователем. Данное свойство принимает булево значение true или false. Если свойство selectable установлено в true, при наведении курсора мыши на текстовый блок он принимает вид курсора для работы с текстом, позволяющего выполнять выделение символов. При этом щелчок правой кнопкой мыши выводит контекстное меню, содержащее перечень стандартных команд по работе с текстом, таких как, например, команда **Copy**.

Для того чтобы исключить возможность выделения текста пользователем (это, в частности, приведет к невозможности копировать содержимое текстового блока), необходимо установить значение свойства selectable в false. При этом следу-

ет иметь в виду, что отключение возможности выделения для пользовательского текста приведет к тому, что пользователь вообще не сможет ввести свой текст в текстовый блок.

Ограничение набора вводимых символов. Свойство *restrict*

Свойство `restrict` позволяет указать произвольный набор символов, которые могут быть введены пользователем в текстовый блок типа Input. Символы, не входящие в этот набор, не могут быть введены с клавиатуры в соответствующий текстовый блок. Следует отметить, что это ограничение не распространяется на программный вывод символов. Сценарий ActionScript может вывести в текстовое поле любой набор символов вне зависимости от значения свойства `restrict`. В качестве значения свойство может принимать строку, задающую набор разрешенных символов или значение `null`.

Если свойство `restrict` принимает значение `null`, это означает отсутствие ограничений на вводимые символы.

Для того чтобы указать набор разрешенных для ввода символов, необходимо присвоить свойству `restrict` строку, содержащую перечень этих символов.

Для указания большого диапазона последовательно расположенных (в кодировке Unicode) символов можно использовать тире (-). Так, следующая строка кода разрешает вводить в текстовое поле `txt` только прописные буквы латинского алфавита:

```
txt.restrict = "A-Z";
```

Можно разрешить ввод сразу нескольких диапазонов символов, например, прописных букв латинского и русского алфавитов, а также цифр:

```
txt.restrict = "0-9A-ZA-Я";
```

Диапазон разрешенных для ввода символов также может быть задан с применением Escape-последовательностей вида `\uxxxx`. Так, следующая строка кода разрешает вводить в текстовое поле `txt` все (строчные и прописные буквы) русского алфавита:

```
txt.restrict = "\u0410-\u044F";
```

Наряду с указанием диапазонов символов с помощью тире можно использовать в строке символ (^), позволяющий разрешить ввод всех предшествующих ему символов и исключить возможность ввода всех следующих за ним символов. Если строка начинается с (^), разрешается ввод любых символов, за исключением следующих после "(^). Если (^) находится в середине строки, то разрешается ввод символов, предшествующих (^), а символы, находящиеся после (^), исключаются. Следующая строка разрешает ввод всех символов русского алфавита, за исключением прописных букв Ь, Ъ и Ы:

```
txt.restrict = "\u0410-\u044F^ЬЪЫ";
```

Для того чтобы включить в список разрешенных для ввода символы (-) и (^), необходимо использовать Escape-последовательность с применением наклонной черты: /- и /^ соответственно. Поскольку наклонная черта Escape-последовательности должна входить в строку, присваиваемую свойству `restrict`, ее также необходимо

представить с помощью Escape-последовательности, то есть двойной черты. Следующая строка кода позволяет вводить в текстовый блок символы (–) и (^):

```
txt.restrict = "\\-\\^";
```

По умолчанию свойство `restrict` принимает значение `null`.

Ограничение количества вводимых символов. Свойство *maxChars*

Свойство `maxChars` позволяет задать максимальное количество символов, которое пользователь сможет ввести в пользовательское текстовое поле. Следует иметь в виду, что ограничение на количество вводимых символов распространяется только на текст, вводимый пользователем с клавиатуры. Данное свойство не ограничивает количество символов, которые выводятся в текстовый блок программным образом.

Свойство `maxChars` принимает числовые значения или значение `null`. Присваивание данному свойству числового значения приведет к тому, что пользователь сможет ввести в соответствующее текстовое поле только заданное количество символов. Значение `null` или его числовой эквивалент 0 соответствуют неограниченному количеству вводимых символов.

По умолчанию свойство `maxChars` принимает значение `null`.

Режим пароля. Свойство *password*

Применение данного свойства дает результат, аналогичный использованию параметра `password`, устанавливаемого в панели **Properties**.

Данное свойство принимает булевы значения `true` или `false`. Присваивание данному свойству значения `true` приводит к тому, что все символы, отображаемые в текстовом поле, заменяются звездочками (`*`). При этом у пользователя отсутствует возможность копировать текст из этого текстового поля. Значение `false` соответствует обычному режиму отображения символов.

Значение данного свойства можно изменить в любой момент, результат изменений сразу же вступает в силу.

По умолчанию свойство `password` принимает значение `false`.

Размеры и длина текста. Свойства *textWidth*, *textHeight* и *length*

Фактический размер текста, выводимого в текстовом блоке, может отличаться от размеров самого блока. Свойства `textWidth` и `textHeight` содержат информацию соответственно о фактической ширине и высоте в пикселах текста, расположенного в текстовом блоке. Данные свойства учитывают реальное состояние текста, на которое, в частности, влияют значения свойств `autoSize` и `wordWrap`. Так, если весь текст не умещается в пределах текстового блока по ширине и его часть срезается правой границей текстового блока, свойство `textWidth` содержит значение, соответствующее полной ширине текста, независимо от того, что он не виден весь целиком. Аналогично дело обстоит и со свойством `textHeight`, указывающим фактическую высоту строк текста, расположенного в текстовом блоке.

Свойства textWidth и textHeight используются только для чтения.

Следует понимать различия между свойствами _width и _height и свойствами textWidth и textHeight. Свойства _width и _height управляют размерами текстового блока и не отражают размер выводимого в нем текста, в то время как свойства textWidth и textHeight содержат информацию о фактических размерах текста, расположенного в текстовом блоке.

Для того чтобы определить количество символов, находящихся в текстовом блоке, используется свойство length. Данное свойство используется только для чтения и возвращает число, соответствующее количеству символов, находящихся в текстовом блоке.

Класс *TextFormat*. Свойства форматирования текста

В предыдущем разделе мы научились управлять различными параметрами текстового блока, используя встроенные свойства класса TextField. Тем не менее класс TextField не позволяет осуществлять форматирование текста. С его помощью нельзя, например, изменить шрифт или его размер, задать тип начертания, выполнить выравнивание текста относительно границ текстового блока и т. д. Для решения этих и многих других задач, связанных с форматированием текста, используется другой встроенный класс, который называется TextFormat. Прежде чем перейти к описанию возможностей этого класса, рассмотрим общий подход к динамическому форматированию текста и выясним, каким образом осуществляется взаимосвязь между объектом TextField и объектом TextFormat.

Для того чтобы применить к текстовому блоку форматирование, необходимо создать объект, являющийся экземпляром класса TextFormat. Затем, используя встроенные свойства класса TextFormat, в данном объекте нужно определить все необходимые параметры форматирования. После этого объект с описанием форматирования может быть применен к любому текстовому блоку.

Создание экземпляра класса TextFormat осуществляется с помощью оператора new и функции конструктора. По умолчанию все свойства нового экземпляра класса принимают значение null, то есть не определены. В следующем примере создается новый экземпляр класса TextFormat, ссылка на который сохраняется в переменной myFormat.

```
var myFormat:TextFormat = new TextFormat();
```

Применение форматирования. Методы *setTextFormat()* и *setNewTextFormat()*

Для применения параметров форматирования, определенных в созданном объекте TextFormat, к конкретному текстовому блоку или нескольким блокам необходимо использовать метод setTextFormat() или метод setNewTextFormat() класса TextField. Рассмотрим синтаксис и назначение этих методов.

Метод *setTextFormat()*

Метод `setTextFormat()` позволяет применить настройки форматирования, определенные в объекте `TextFormat`, к текущему содержимому указанного текстового блока. Форматирование может быть применено к отдельному символу, к диапазону последовательно расположенных символов или ко всему содержимому текстового блока. Каждая из этих задач предполагает применение соответствующей формы синтаксиса при вызове метода `setTextFormat()`.

- Для того чтобы применить форматирование к конкретному символу, используется следующий синтаксис:

  текстовый блок.setTextFormat(*индекс*, *объект форматирования*)

  Здесь *текстовый блок* — это имя экземпляра текстового блока, к содержимому которого применяется форматирование.

  Параметр *индекс* задает индекс символа, к которому будет применено форматирование. Данный параметр должен иметь целочисленное неотрицательное значение и может быть задан числовым литералом или выражением, возвращающим число. Следует иметь в виду, что первый символ в текстовом блоке имеет индекс 0.

  Параметр *объект форматирования* — это экземпляр класса `TextFormat`, в котором определен набор свойств, задающих требуемые значения параметров форматирования.

- Для того чтобы применить форматирование к последовательному диапазону символов, используется следующий синтаксис:

  текстовый блок.setTextFormat(начальный *индекс*, конечный индекс,↵
  объект форматирования)

  Параметр начальный *индекс* задает индекс первого символа диапазона, к которому будет применено форматирование. Данный параметр должен иметь целочисленное неотрицательное значение и может быть задан числовым литералом или выражением, возвращающим число.

  Параметр конечный индекс задает индекс первого символа, следующего за диапазоном символов, к которому применяется форматирование. То есть сам символ с индексом, заданным этим параметром, форматироваться не будет.

- И, наконец, для того чтобы применить форматирование ко всему текущему содержимому текстового блока, используется следующий синтаксис:

  текстовый блок.setTextFormat(объект форматирования)

Необходимо отметить, что в результате действия метода `setTextFormat()` форматирование применяется только к текущему содержимому текстового блока, то есть к тому тексту, который находился в текстовом блоке в момент вызова данного метода. Новый текст, который будет в дальнейшем помещен в текстовый блок (программным образом или пользователем) после текущей позиции курсора (то есть после текущего набора символов), будет содержать форматирование, используемое для

данного текстового блока по умолчанию, а не форматирование, заданное при вызове метода `setTextFormat()`[1].

Форматирование, используемое текстовым блоком по умолчанию, задается с помощью метода `setNewTextFormat()`.

Метод *setNewTextFormat()*

Данный метод задает форматирование, которое по умолчанию применяется к новому тексту, вводимому в текстовый блок пользователем или средствами ActionScript. Данный метод имеет следующий синтаксис:

текстовый блок`.setNewTextFormat(`*объект форматирования*`)`

Здесь *текстовый блок* — это имя экземпляра текстового блока, к содержимому которого применяется форматирование.

Параметр *объект форматирования* — это экземпляр класса `TextFormat`, в котором определен набор свойств, задающих требуемые значения параметров форматирования.

Примеры применения обоих методов будут приведены далее в настоящем разделе при описании свойств класса `TextFormat`, определяющих параметры форматирования.

Шрифт. Свойство *font*

Свойство `font` позволяет указать шрифт, используемый при выводе и наборе текста в текстовом блоке. В качестве значения свойство `font` принимает строку, содержащую имя используемого шрифта. Следующий фрагмент кода создает динамический текстовый блок и выводит в него текст, используя шрифт `Pragmatica`:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 150, 50);
txt.autoSize = true;
txt.text ="Vita brevis ars longa";
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Pragmatica";
txt.setTextFormat(myFormat);
```

Встраивание шрифта

По умолчанию шрифт, используемый для отображения текста в текстовом блоке, не встраивается в документ. Следствием этого, в частности, являются неровные зубчатые границы символов, а также потенциальная вероятность того, что у конечного пользователя в системе может не оказаться данного шрифта.

Для того чтобы программным образом встроить очертания символов используемого шрифта, необходимо выполнить следующие действия.

1. В рабочей среде в библиотеке документа создать шрифтовой символ (font symbol), выбрав требуемый шрифт и начертание. Следует иметь в виду, что для

[1] Следует отметить, что если новый текст будет добавлен не после текущего содержимого текстового блока, а в его середину, то к новому тексту будет применяться форматирование, используемое текущим текстом. — *Прим. автора*.

каждого типа начертания (обычного, жирного или курсивного) одного и того же шрифта необходимо создавать отдельный шрифтовой символ.

2. В окне настройки параметров связи **Linkage Properties** созданного шрифтового символа установить флажок **Export for ActionScript** и ввести уникальный идентификатор в поле **Identifier**.
3. В тексте сценария для требуемого текстового блока установить свойство `embedFonts` в значение `true`. Данное свойство класса `TextField` определяет, нужно ли встраивать очертания шрифта для текста, выводимого в соответствующем текстовом блоке. Очертания шрифта могут быть встроены только в том случае, если в библиотеке документа имеется соответствующий шрифтовой символ. По умолчанию свойство `embedFonts` принимает значение `false`.
4. В качестве значения свойства `font` объекта `TextFormat` необходимо указать строку, содержащую идентификатор связи шрифтового символа, заданный в окне **Linkage Properties** на шаге 2.

Следующий код выводит в текстовый блок с именем `txt` сообщение, используя встроенный шрифт, для которого в библиотеке документа создан шрифтовой символ, имеющий идентификатор связи `myFont`.

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 150, 50);
txt.text ="Vita brevis ars longa";
txt.embedFonts = true;
var myFormat:TextFormat = new TextFormat();
myFormat.font = "myFont";
txt.setTextFormat(myFormat);
```

Следует иметь в виду, что при использовании встроенных шрифтов текст выводится с применением начертания, заданного при создании соответствующего шрифтового символа. Применение свойств `italic` и `bold` не оказывает влияния на начертание встроенного шрифта.

Кегль. Свойство *size*

Свойство `size` отвечает за размер символов текста, выводимых в текстовом блоке. Величина кегля задается в пунктах. Следующий код выводит текст, используя кегль величиной 18 пунктов:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 150, 50);
txt.text = "Cogitum Ergo Sum";
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Arial";
myFormat.size = 18;
txt.setTextFormat(myFormat)
```

Цвет. Свойство *color*

Свойство `color` класса `TextFormat` является аналогом свойства `textColor` класса `TextField` и применяется для изменения цвета текста. Данное свойство принимает числовое значение, которое, как правило, задается шестнадцатеричным числом. Следующий код выводит текст, используя красный цвет:

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 180, 50);
txt.text = "Per Aspera Ad Astrum";
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Arial";
myFormat.size = 18;
myFormat.color = 0xFF0000;
txt.setTextFormat(myFormat)
```

Начертание. Свойства *bold*, *italic*, *underline*

Свойство `bold` используется для отображения текста с применением жирного начертания. Свойство `italic` используется для отображения текста с применением курсивного начертания. Применение этих свойств является эквивалентом использования соответствующих опций форматирования панели **Properties**. Свойство `underline` используется для отображения текста с применением подчеркивания. В то время как жирное и курсивное начертание может быть задано для текстового блока в рабочей среде, подчеркивание может быть применено только при динамическом форматировании текста.

Каждое из перечисленных свойств принимает булево значение `true` или `false`, определяющее, включен или выключен соответствующий режим. Для того чтобы текст выводился с применением обычного начертания и без подчеркивания, нужно присвоить этим свойствам значения `false` или `null`.

```
_root.createTextField("txt",_root.getNextHighestDepth(), 0, 0, 150, 50);
txt.autoSize = true;
txt.text ="Vita brevis ars longa";
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Arial";
myFormat.size = 18;
myFormat.bold = true;
myFormat.italic = true;
myFormat.underline = true;
txt.setTextFormat(myFormat)
```

Выравнивание текста. Свойство *align*

Свойство `align` позволяет установить выравнивание текста относительно боковых границ текстового блока.

Данное свойство принимает одно из трех допустимых строковых значений:

- "left" — выравнивание по левой границе текстового блока;
- "right" — выравнивание по правой границе текстового блока;
- "center" — выравнивание по центру.

Маркированный список. Свойство *bullet*

Свойство bullet позволяет отображать содержимое тестового блока в виде маркированного списка. При этом каждый абзац текста располагается с отступом, а слева от его красной строки находится маркер элемента списка.

Данное свойство принимает булево значение true или false.

Приведенный программный код создает тестовый блок и в цикле выводит в него перечень элементов списка. Результат действия этого сценария приведен на рис. 14.4.

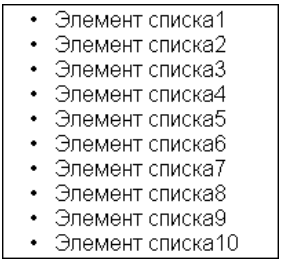


Рис. 14.4. Создание маркированного списка

```
_root.createTextField("txt", 1, 0, 0, 200, 100);
txt.autoSize = true;
txt.wordWrap = true;
txt.border = true;
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Arial";
myFormat.size = 16;
myFormat.align = "left";
myFormat.bullet = true;
txt.setNewTextFormat(myFormat);
for (i = 1; i <= 10; i++) {
       txt.text += "Элемент списка" + i + "\n";
}
```

Следует обратить внимание на то, что в приведенном примере для применения параметров форматирования используется метод setNewTextFormat(), а не setTextFormat(). Этот метод задает параметры форматирования, используемые по

умолчанию для текстового блока txt. Поэтому текст, помещенный в данный блок, будет автоматически форматироваться в соответствии со значениями свойств объекта TextFormat, ссылка на который хранится в переменной myFormat.

Тот же результат может быть получен и с помощью метода setTextFormat(), однако в этом случае вызов данного метода должен быть помещен после закрывающей скобки тела цикла. Это обусловлено тем, что метод setTextFormat() применяет форматирование только к текущему содержимому текстового блока и его действие не распространяется на содержимое, помещаемое в текстовый блок после вызова данного метода.

Параметры абзаца. Свойства *indent, blockIndent, leading, leftMargin, rightMargin*

Свойства, объединенные в категорию "Параметры абзаца", позволяют программным образом выполнять форматирование абзаца так же, как это осуществляется в рабочей среде с помощью диалогового окна **Format Options**.

Свойство indent задает отступ красной строки, то есть расстояние от левого поля текстового блока до первого символа абзаца. В качестве значения данное свойство принимает целое неотрицательное число.

Свойство blockIndent задает отступ всех строк текста от левой границы текстового блока. Использование данного свойства сдвигает левое поле текста влево на заданное расстояние. Данное свойство можно применять в сочетании со свойством indent. При этом отступ красной строки, который определяется с помощью свойства indent, задается относительно позиции левого поля, установленной свойством blockIndent. В качестве значения данное свойство принимает целое неотрицательное число.

Свойство leading устанавливает интерлиньяж, то есть промежуток между базовыми линиями строк текста, расположенного в текстовом блоке. Значение интерлиньяжа задается в пунктах. По умолчанию промежуток между строками составляет два пункта. В качестве значения данное свойство принимает целое неотрицательное число.

И наконец, свойства leftMargin и rightMargin позволяют установить ширину левого и правого полей соответственно. Ширина поля задается в пикселах как целое неотрицательное число.

Следующий код создает текстовый блок и устанавливает для него следующие параметры, используемые по умолчанию: отступ красной строки — 20, интерлиньяж — 15, левое и правое поля — по 10.

```
_root.createTextField("txt", 1, 0, 0, 200, 100);
var myFormat:TextFormat = new TextFormat();
myFormat.indent = 20;
myFormat.leading = 15;
myFormat.leftMargin = myFormat.rightMargin = 10;
txt.setNewTextFormat(myFormat);
```

Установка позиций табуляции. Свойство *tabStops*

Свойство `tabStops` позволяет задать положение позиций табуляции, используемых в текстовом блоке. Для выполнения табуляции нужно воспользоваться Escape-последовательностью `\t`. По умолчанию отступ табуляции примерно соответствует ширине символа, однако при необходимости это значение может быть изменено с помощью свойства `tabStops`. В качестве значения данному свойству должен быть присвоен массив, элементами которого являются целые положительные числа, определяющие позиции табуляции, используемые при выводе текста. Позиции табуляции задаются в пикселах, причем элементы массива `tabStops` соответствуют последовательности применения управляющего символа табуляции (`\t`). Так, первая позиция табуляции задается первым элементом массива `tabStops`, вторая позиция задается вторым элементом массива `tabStops` и т. д. В следующем примере создается динамический текстовый блок, в который выводится текст, при этом позиции табуляции расположены с интервалом в 30 пикселов:

```
_root.createTextField("txt", _root.getNextHighestDepth(), 0, 0, 150, 50);
txt.border = true;
txt.multiline = true;
txt.autoSize = true;
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Arial Cyr";
myFormat.size = 20;
myFormat.tabStops = [30, 60, 90, 120];
txt.text = "F\tL\tA\tS\tH";
txt.setTextFormat(myFormat);
```

Результат действия этого сценария приведен на рис. 14.5.

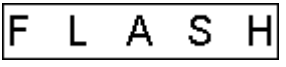


Рис. 14.5. Применение свойства `tabStops` для установки пользовательских позиций табуляции

Создание всплывающих подсказок

Хорошей иллюстрацией применения возможностей создания и настройки параметров динамического текста может служить задача, связанная с созданием всплывающих подсказок. Всплывающая подсказка появляется рядом с курсором мыши при наведении на объект и содержит информацию, поясняющую назначение данного объекта. Применение всплывающих подсказок обеспечивает лучшее взаимодействие с пользователем и позволяет ему быстрее освоиться с интерфейсом приложения.

Предлагаемый здесь подход заключается в следующем. Всплывающая подсказка представляет собой динамический текстовый блок, снабженный фоном и рамкой.

Реализация функциональности всплывающих подсказок осуществляется с помощью функции `addTextHint()`, принимающей следующие параметры:

- `obj` — объект (клип или кнопка), для которого должна выводиться всплывающая подсказка;
- `hintMessage` — строка, содержащая текст, используемый при выводе всплывающей подсказки.
- `hintStyleObj` — объект, определяющий ряд параметров форматирования всплывающей подсказки. Данный объект должен содержать следующие свойства:
  - `hintFont` — строка с названием шрифта;
  - `hintFontSize` — число, определяющее кегль;
  - `textCol` — число, определяющее цвет текста;
  - `frameCol` — число, определяющее цвет рамки;
  - `backgroundCol` — число, определяющее цвет фона.
- Остальные три параметра являются необязательными и представляют собой ссылки на функции, которые должны вызываться при наведении на элемент (событие `rollOver`), при уходе с элемента (событие `rollOut`) и при уходе с элемента с нажатой кнопкой мыши (событие `dragOut`). Наличие данных параметров обусловлено тем, что реализация всплывающей подсказки осуществляется с помощью определения трех соответствующих методов обработчиков для объекта, на который назначается всплывающая подсказка. Однако довольно часто объект сам должен реагировать на эти три события, в связи с чем возникает конфликт, связанный с тем, что метод обработчика, определяемый последним, будет замещать такой же метод обработчика, определенный ранее. Как следствие, будет работать либо только подсказка, либо инструкции, определенные в созданном после вызова функции `addTextHint()` методе обработчика объекта. Чтобы избежать этой проблемы в нашем случае, следует помнить, что для объектов, которым назначаются всплывающие подсказки, нельзя непосредственно определять методы `onRollOver()`, `onRollOut()` и `onDragOut()`. Если объект должен реагировать на эти события, необходимо создать функцию, выполняющую требуемые действия, и передать ссылку на нее в качестве параметра функции `addTextHint()`. Следует иметь в виду, что при вызове этой функции ей в качестве первого параметра всегда будет передаваться ссылка на объект, которому назначается подсказка. Это сделано для большего удобства, чтобы из функции можно было просто обратиться к соответствующему объекту.

Для создания всплывающей подсказки необходимо вызвать функцию `addTextHint()`, передав ей ссылку на объект, для которого должна выводиться подсказка; текст самой подсказки; объект с настройками форматирования всплывающей подсказки и необязательными ссылками на функции, которые будут вызываться при обнаружении событий `onRollOver`, `onRollOut`, `onDragOut`, связанных с объектом, для которого должна выводиться подсказка. Функция `addTextHint()` приведена в листинге 14.1. Программный код снабжен подробными комментариями, поэтому приведем только краткое описание этапов алгоритма.

1. В теле функции `addTextHint()` объявляется функция `showHint()`, выполняющая создание, позиционирование и форматирвание всплывающей подсказки

2. Функция showHint() выполняет создание пустого текстового блока с произвольным размером и координатами. Поскольку всплывающая подсказка должна быть всегда видна, текстовый блок будет помещен на ближайший верхний свободный уровень стека основной монтажной линейки (_root). Имя создаваемого текстового блока задается динамически и состоит из слова "hint" и числа, соответствующего уровню стека, на котором находится данный блок.
3. Настройка параметров текстового блока и его форматирование выполняются на основе значений свойств объекта, который передается функции addTextHint() в качестве параметра. При этом свойство autoSize устанавливается в значение true, чтобы весь текст подсказки был виден вне зависимости от размеров блока, заданных при его создании. Кроме того, свойство selectable устанавливается в значение false, чтобы исключить возможность наведения курсора на текстовый блок.
4. Включение видимости подсказки выполняется путем вызова функции showHint(). Отключение видимости подсказки выполняется путем удаления текстового блока с помощью метода removeTextField(), который будет описан далее в этой главе. Поскольку при уходе курсора с объекта кнопка мыши может оставаться нажатой, необходимо отключить видимость подсказки не только с помощью метода обработчика onRollOut(), но также и onDragOut().

Листинг 14.1. Создание всплывающей подсказки

```
function addTextHint(obj:Object, hintMessage:String, hintStyleObj:Object,
rolloverFunc:Function, rolloutFunc:Function, dragoutFunc:Function) {
    // Переменная, содержащая номер ближайшего свободного уровня
    // стека
    var depth = _root.getNextHighestDepth();
    /* Функция, выполняющая создание, позиционирование и
    форматирование всплывающей подсказки для объекта obj */
    function showHint() {
        // Создаем текстовый блок для вывода подсказки
        _root.createTextField("hint" + depth, depth, _root._xmouse
        + 5, _root._ymouse + 15, 1, 1);
        // Настройка параметров текстового блока
        with (_root["hint" + depth]) {
            // Рамка
            border = true;
            // Цвет рамки
            borderColor = hintStyleObj.frameCol;
            // Фон
            background = true;
            // Цвет фона
            backgroundColor = hintStyleObj.backgroundCol;
```

```
                /* Подгонка размеров блока под содержимое
                с выключкой по левой границе */
                autoSize = true;
                // Отключение возможности выделения текста
                selectable = false;
                // Вывод текста подсказки
                text = hintMessage;
        }
        // Форматирование текста подсказки
        var hintFormat:TextFormat = new TextFormat();
        // Шрифт
        hintFormat.font = hintStyleObj.hintFont;
        // Кегль
        hintFormat.size = hintStyleObj.hintFontSize;
        // Цвет текста
        hintFormat.color = hintStyleObj.textCol;
        // Прменение форматирования
        _root["hint" + depth].setTextFormat(hintFormat);
}
/* ПОЯВЛЕНИЕ И ИСЧЕЗАНИЕ ПОДСКАЗКИ ПРИ НАВЕДЕНИИ НА КЛИП И УХОДЕ С
НЕГО */
obj.onRollOver = function() {
        //При наведении...
        /* Вызываем функцию, определяющую действия, которые должны
        быть выполнены при наведении на obj, и в качестве параметра
        передаем ей ссылку на obj */
        rolloverFunc(this);
        // Включаем отображение подсказки
        showHint();
};
// При уходе...
obj.onRollOut = function() {
        /*Вызываем функцию, определяющую действия, которые должны
        быть выполнены при уходе с obj, и в качестве параметра
        передаем ей ссылку на obj*/
        rolloutFunc(this);
        // Отключаем отображение подсказки
        _root["hint" + depth].removeTextField();
};
obj.onDragOut = function() {
```

```
            dragoutFunc(this);
            _root["hint" + depth].removeTextField();
        };
}
```

Приведем несколько примеров практического применения данной функции. Следующий код создает всплывающие подсказки для кнопок управления анимацией с именами `play`, `pause` и `rewind`:

```
var hintStyle:Object = {hintFont:"Arial", hintFontSize:14,↳
textCol:0x000000, frameCol:0x000000, backgroundCol:0xFEFFCC};
addTextHint(play, "Воспроизведение", hintStyle);
addTextHint(pause, "Пауза", hintStyle);
addTextHint(rewind, "Перемотка", hintStyle);
```

Следующий код для всех экземпляров клипов, расположенных на основной монтажной линейке, создает всплывающие подсказки, содержащие их имена. Кроме того, при наведении или уходе курсора с клипа в текстовое поле выводится имя этого клипа и название события (`rollOver`, `rollOut`, `dragOut`). Обратите внимание, что в объявлении функций, ссылки на которые передаются в качестве параметров функции `addTextHint()`, указывается формальный параметр. В теле функции его можно использовать как ссылку на объект, которому назначается всплывающая подсказка.

```
function rolloverFunc (o:Object){
/* o - возвращает ссылку на объект, которому назначается всплывающая
подсказка */
        eventTxt.text = o+": " + "rollOver";
};
function rolloutFunc (o:Object){
        eventTxt.text = o+": " + "rollOut";
};
function dragoutFunc (o:Object){
        eventTxt.text = o+": " + "DragOut";
};
var hintStyle:Object = {hintFont:"Arial", hintFontSize:14,
textCol:0x000000, frameCol:0x000000, backgroundCol:0xFEFFCC};
for (i in _root){
        if (_root[i] instanceof MovieClip){
        addTextHint (_root[i], _root[i]._name, hintStyle,
        rolloverFunc,rolloutFunc, dragoutFunc);
        }
}
```

Используемый здесь оператор `instanceof` возвращает значение `true` в случае, если его левый операнд является экземпляром класса, заданного правым операндом.

Получение информации о параметрах форматирования текста

Для получения информации о параметрах форматирования, применяемых к тексту, можно воспользоваться методами `getTextFormat()` и `getNewTextForamat()`. Применение данных методов похоже на использование методов `setTextFormat()` и `setNewTextFormat()`, однако они не форматируют текст, а возвращают объект, содержащий используемые параметры форматирования.

Метод `getTextFormat()` возвращает объект `TextFormat`, содержащий информацию о параметрах форматирования отдельного символа, диапазона символов или всего текущего содержимого текстового блока. Как и метод `setTextFormat()`, метод `getTextFormat()` имеет три различные формы вызова.

- Для того чтобы получить информацию о параметрах форматирования отдельного символа, используется следующий синтаксис:

  `текстовый блок``.getTextFormat(`*`индекс`*`)`

  Здесь *`текстовый блок`* — это имя экземпляра текстового блока, в котором находится символ.

  Параметр *`индекс`* задает индекс символа, параметры форматирования которого необходимо определить. Данный параметр должен иметь целочисленное неотрицательное значение и может быть задан числовым литералом или выражением, возвращающим число. Следует иметь в виду, что первый символ в текстовом блоке имеет индекс 0.

- Для того чтобы получить информацию о параметрах форматирования диапазона последовательно расположенных символов, используется следующий синтаксис:

  `текстовый блок``.getTextFormat(`*`начальный индекс`*`, `*`конечный индекс`*`)`

  Параметр *`начальный индекс`* задает индекс первого символа диапазона. Данный параметр должен иметь целочисленное неотрицательное значение и может быть задан числовым литералом или выражением, возвращающим число.

  Параметр *`конечный индекс`* задает индекс первого символа, следующего за диапазоном символов, параметры форматирования которого требуется определить.

- Для того чтобы получить информацию о параметрах форматирования всего текущего содержимого текстового блока, используется следующий синтаксис:

  `текстовый блок``.getTextFormat()`

Метод возвращает объект `TextFormat`, значения свойств которого соответствуют форматированию, используемому для заданного набора символов. Все свойства в возвращаемом методом `getTextFormat()` объекте определены, то есть их значения не равны `null`. Объект, возвращаемый методом `getTextFormat()`, отражает форматирование, применяемое к символам, которые уже находятся в текстовом блоке в момент вызова данного метода.

Следующий код выводит в панель **Output** значения всех свойств форматирования, применяемых к тексту, который выводится в динамически созданный текстовый блок:

```
this.createTextField("txt", 1, 0, 0, 1, 1);
txt.autoSize = true;
txt.text = "Продолжительность брака обратно пропорциональна расходам на↵
свадьбу";
myFormat = new TextFormat();
myFormat.font = "Arial Cyr";
myFormat.size = 18;
myFormat.display = false;
txt.setTextFormat(myFormat);
var formatProps:TextFormat = txt.getTextFormat();
for (i in formatProps) {
       trace(i + " = " + formatProps[i]);
}
```

Для получения информации о параметрах форматирования вновь добавленного или введенного пользователем шрифта нужно воспользоваться методом `getNewTextFormat()`. Данный метод возвращает объект `TextFormat`, содержащий набор свойств форматирования, значения которых были установлены с помощью метода `setNewTextFormat()`. Метод `getNewTextFormat()` имеет следующий синтаксис:

текстовый блок`.getNewTextFormat()`

Все свойства в возвращаемом методом `getTextFormat()` объекте определены, то есть их значения не равны `null`.

Получение информации о метриках шрифта

Для осуществления более точного контроля над размещением текста в пределах текстового блока может возникнуть необходимость в определении параметров метрик шрифта, то есть некоторых специфических размеров, определяющих местоположение символов в строке и размер текстового блока. Такая потребность может возникнуть, например, в ситуации, требующей создания текстового блока со строго фиксированными размерами, заданными таким образом, чтобы выводимый в этом блоке текст был виден целиком.

Для получения некоторой информации о метриках используемого текста с учетом применяемого к нему форматирования можно воспользоваться единственным методом класса `TextFormat`, который называется `getTextExtent()`. Метод `getTextExtent()` имеет следующий синтаксис:

объект форматирования`.getTextExtent(`*текст*`, `*ширина*`)`.

Здесь *объект форматирования* — это объект `TextFormat` с набором свойств, задающих параметры форматирования.

Параметр *текст* — это строка текста, для которой требуется получить сведения о метриках шрифта. Данный параметр может быть задан строковым литералом или выражением, возвращающим строку.

Параметр *ширина* применяется при необходимости использования режима автоматического переноса строк (word wrap). Данный параметр задает максимальную длину строки, при достижении которой будет выполняться переход на следующую строку. На основании значения данного параметра определяется количество строк, на которые будет разбита исходная строка, заданная параметром *текст*, и, соответственно, общая высота всех этих строк. Параметр является необязательным. Значение данного параметра может быть задано числовым литералом или выражением, возвращающим целое положительное число.

Метод `getTextExtent()` возвращает объект, содержащий набор свойств, описывающих метрики шрифта для строки, заданной параметром *текст*. При этом учитываются все настройки форматирования, которые в виде свойств содержатся в объекте *объект форматирования*, вызвавшем метод `getTextExtent()`. Далее приводится перечень и описание всех свойств объекта, возвращаемого методом `getTextExtent()`. Рисунок 14.6 иллюстрирует основные метрические характеристики текстового блока и шрифта:

- `width` — данное свойство содержит ширину текста в пикселах;
- `height` — данное свойство содержит общую высоту текста в пикселах;
- `textFieldWidth` — данное свойство определяет, какую ширину должен иметь текстовый блок, для того чтобы все текстовое содержимое, заданное параметром *текст*, было видно целиком.

  Если текст выводится в одну строку, то значение, возвращаемое данным свойством, соответствует нулевой ширине полей (величина которых задается свойствами `leftMargin` и `rightMargin`). При задании ширины полей, отличной от нуля, для получения реально требуемой ширины текстового блока необходимо прибавить к значению свойства `textFieldWidth` ширину левого и правого полей.

  Если для вывода текста используется автоматический перенос строк (то есть свойство `wordWrap` установлено в `true`), то значение свойства `textFieldWidth` будет совпадать со значением параметра *ширина*, передаваемого методу `getTextExtent()`;
- `textFieldHeight` — данное свойство определяет, какую высоту должен иметь текстовый блок, для того чтобы все текстовое содержимое, заданное параметром *текст*, было видно целиком;
- `ascent` — данное свойство содержит расстояние от верхней границы строки до ее базовой линии;
- `descent` — данное свойство содержит расстояние от нижней границы строки до ее базовой линии.

Примечание

Текстовый блок всегда имеет бордюр толщиной в 2 пиксела. В связи с этим текстовые символы не могут располагаться вплотную к границе текстового блока.

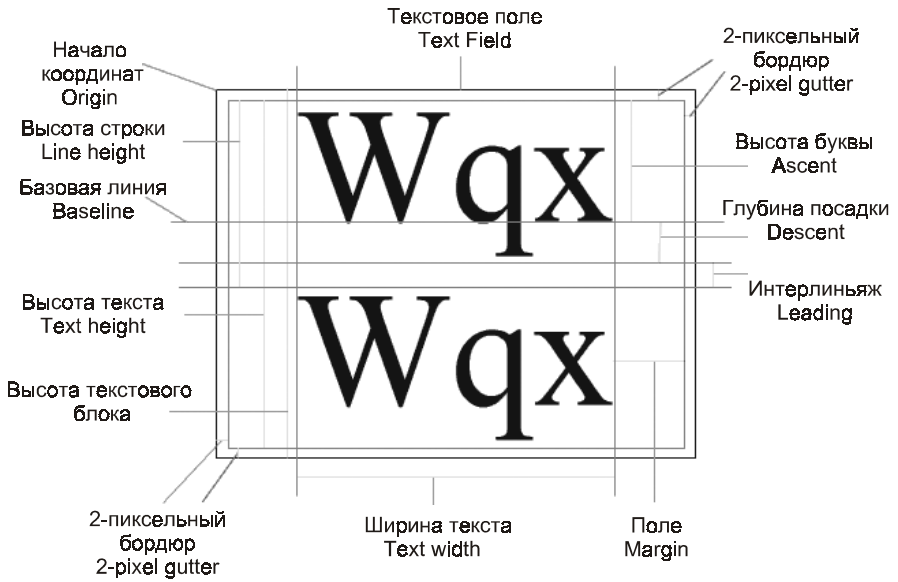


Рис. 14.6. Метрические характеристики текстового блока и шрифта

Приведенный далее код создает текстовый блок для вывода текста с применением автоматического переноса строк, устанавливая для него размеры таким образом, чтобы весь текст был виден.

```
var myText = "Для получения информации о метриках текста с учетом↵
применяемого к нему форматирования можно воспользоваться методом↵
getTextExtent().";
var textWidth = 300; // Ширина текстового блока
var myFormat = new TextFormat();
myFormat.font = "Times New Roman";
myFormat.size = 20;
myFormat.leading = 3;
myFormat.align = "left";
myFormat.leftMargin = myFormat.rightMargin = 10;
// Сохраняем объект, возвращаемый методом getTextExtent(), в переменной
var textMetrics = myFormat.getTextExtent(myText, textWidth);
/* Создаем текстовый блок, в качестве размеров используя значения свойств
объекта, возвращенного методом getTextExtent()*/
this.createTextField("txt", 1, 0, 0, textMetrics.textFieldWidth,↵
textMetrics.textFieldHeight);
txt.text = myText;
```

```
txt.setTextFormat(myFormat);
txt.wordWrap = true;
txt.border = true;
```

Для решения аналогичной задачи, но при выводе текста в одну строку, необходимо скорректировать ширину создаваемого текстового блока на ширину полей (`leftMargin`, `rightMargin`), величину отступа красной строки (`indent`) и отступа абзаца (`blockIndent`). Следующий сценарий иллюстрирует определение размеров текстового блока для вывода текста в одну строку с учетом заданного значения ширины полей.

```
var myText = "Однострочный текст";
var margin = 15;
var myFormat = new TextFormat();
myFormat.font = "Times New Roman";
myFormat.size = 20;
myFormat.align = "center";
myFormat.leftMargin = myFormat.rightMargin = margin;
var textFormatProps = myFormat.getTextExtent(myText);
this.createTextField("txt", 1, 0, 0, textFormatProps.textFieldWidth +↳
2*margin , textFormatProps.textFieldHeight);
txt.text = myText;
txt.border = true;
txt.setTextFormat(myFormat);
```

Проверка наличия у пользователя используемого шрифта

В классе `TextField` определен статический метод `getFontList()`. Этот метод вызывается не с помощью экземпляра текстового блока (как, например, метод `setTextFormat()`), а непосредственно из класса. Метод `getFontList()` возвращает массив, содержащий перечень названий всех шрифтов, установленных на компьютере, на котором выполняется воспроизведение Flash-фильма. Имена шрифтов в возвращаемом массиве представляют собой строки. Метод `getFontList()` не принимает параметров и имеет следующий синтаксис:

```
TextField.getFontList()
```

С помощью данного метода можно легко проверить факт наличия в системе пользователя требуемого шрифта. Приведенная в листинге 14.2 функция `detectFont()` возвращает значение `true`, если шрифт, название которого было передано ей в качестве параметра, установлен у пользователя, и значение `false`, если данный шрифт отсутствует.

Листинг 14.2. Проверка наличия у пользователя заданного шрифта

```
function detectFont(font:String):Boolean {
        var userFonts:Array = TextField.getFontList();
        for (var i = 0; i < userFonts.length; i++) {
                if (userFonts[i] == font) {
                        return true;
                }
        }
        return false;
}
```

Организация прокрутки текста

Динамический или пользовательский текстовый блок представляет собой своеобразное окно, в пределах которого находятся строки текста (если блок многострочный) или одна строка (если блок однострочный). Если размеры текста, выводимого в текстовый блок, превышают размеры этого блока, часть текста срезается его границами, то есть становится невидна. Избежать этого можно либо с помощью свойства `asutoSize`, либо установив точные размеры текстового блока на основании информации о размерах выводимого в нем текста. Однако подгонка размеров текстового блока под размеры выводимого в нем текста в ряде случаев оказывается неприемлемой или даже невозможной (если размеры текста достаточно велики, текстовый блок может выходить за пределы окна проигрывателя). При необходимости вывода в текстовый блок больших фрагментов текста разработчик сталкивается с необходимостью организации механизма, позволяющего пользователю прокручивать текст для того, чтобы увидеть его целиком.

Автоматическая прокрутка текста

В принципе стандартные возможности прокрутки текста реализуются для динамического или пользовательского текстового блока по умолчанию при использовании режима **Selectable** (Выделяемый текст) или при установке свойства `selectable` в значение `true`. К таким возможностям относятся следующие.

- Для выполнения прокрутки в вертикальном направлении можно поместить курсор в текст и либо, нажав левую кнопку мыши, протащить его в нужном направлении, либо воспользоваться клавишами перемещения курсора.
- Прокрутку в вертикальном направлении можно осуществлять с помощью колеса мыши.

Для управления возможностью применения колеса мыши при прокрутке текста можно использовать свойство `mouseWheelEnabled` класса `TextField`. Данное свойство принимает булево значение `true` или `false` и определяет, может ли пользователь выполнять прокрутку текста в соответствующем текстовом блоке с помощью колеса мыши (`true`) или нет (`false`). Следует иметь в виду, что автоматическая

прокрутка содержимого текстового блока с помощью колеса мыши осуществляется только при условии, что свойство `selectable` данного текстового блока установлено в `true`. Следующий код отключает возможность выполнения автоматической прокрутки текста с помощью колеса мыши в текстовом блоке с именем `txt`. При этом сохраняется возможность выделения содержимого `txt`.

```
txt.mouseWheelEnabled = false;
```

Свойства управления вертикальной прокруткой текста

Класс `TextField` обладает набором свойств, которые позволяют осуществлять прокрутку содержимого текстового блока программным образом. Эти свойства, в частности, используются при создании пользовательских элементов прокрутки текста, таких как полоса прокрутки и кнопки прокрутки. В данном разделе приводится описание свойств, контролирующих состояние прокрутки текста, после чего будет рассмотрен пример, описывающий создание пользовательской полосы прокрутки.

Свойство *scroll*

Свойство `scroll` определяет номер самой верхней видимой строки текстового блока. Нумерация строк в текстовом блоке выполняется сверху вниз. Самая верхняя строка текста имеет индекс 1. Свойство `scroll` используется как для чтения, так и для записи. Извлекая значение данного свойства, можно определить номер верхней видимой строки текста. Присваивая значение данному свойству, можно построчно прокручивать текст так, чтобы строка с заданным индексом оказывалась самой верхней видимой строкой. Увеличение значения свойства `scroll` смещает текст в текстовом блоке вверх (прокрутка вниз), а уменьшение — вниз (прокрутка вверх). В качестве значения свойство `scroll` может принимать целое положительное число в диапазоне от 1 до значения, возвращаемого свойством `maxscroll` (описание этого свойства приводится ниже). Если свойству `scroll` присваивается дробное числовое значение, оно автоматически округляется до меньшего целого. Если свойству `scroll` присваивается значение меньшее 1, оно автоматически становиться равным 1. Если свойству `scroll` присваивается значение большее, чем `maxscroll`, оно автоматически становиться равным `maxscroll`.

Следующий код прокручивает текст, выведенный в текстовом блоке `txt`, на одну строку вниз при каждом нажатии на кнопку `scrollDownBut`.

```
scrollDownBut.onPress = function(){
       txt.scroll++;
};
```

Примечание

Значение свойства `scroll` не зависит от параметров форматирования, примененных к отдельным символам в строке, равно как и от параметров форматирования всего текста. В частности, применение различных размеров символов в одной строке и изменение расстояния между строками (интерлиньяжа) не оказывают влияния на значение свойства `scroll`.

Свойство *maxscroll*

Данное свойство позволяет определить, сколько строк текстового блока может быть прокручено (вверх и/или вниз), и содержит максимальное значение, которое может принимать свойство `scroll` для указанного текстового блока. Значение свойства `maxscroll` всегда на единицу больше количества строк текста, выходящих за пределы границ текстового блока. Данное свойство используется только для чтения, то есть присвоить ему значение нельзя.

Если значение свойства `scroll` равно значению свойства `maxscroll`, это означает, что последняя строка текста видна в пределах текстового блока.

Несмотря на то, что свойство `scroll` не может принимать значение большее, чем значение свойства `maxscroll`, тем не менее, скорректируем предыдущий пример простейшей прокрутки, введя проверку условия достижения последней строки с тем, чтобы исключить возможность присваивания свойству `scroll` недопустимого значения. Если последняя строка уже видна, дальнейшего обращения к свойству `scroll` не происходит:

```
scrollDownBut.onPress = function() {
       if (txt.scroll != txt.maxscroll) txt.scroll++;
};
```

Свойство *bottomScroll*

Свойство `bottomScroll` определяет номер самой нижней видимой строки текстового блока. Таким образом, видимыми являются строки, начиная со строки с номером, соответствующим значению свойства `scroll`, и заканчивая строкой с номером, соответствующим значению свойства `bottomScroll`. Свойство `bottomScroll` используется только для чтения.

Следующий код выводит в панель **Output** общее число строк (видимых и выходящих за границы блока) в текстовом блоке `txt`:

```
trace(txt.maxscroll + txt.bottomScroll - txt.scroll);
```

Свойства управления горизонтальной прокруткой текста

В большинстве случаев при выводе текста на европейских языках возникает необходимость в обеспечении вертикальной прокрутки. Поскольку символы располагаются в строке, для комфортного чтения эта строка должна быть видна целиком. Срезание строки по горизонтали существенно снижает читабельность текста. Тем не менее в классе `TextField` существует два свойства, позволяющих организовывать горизонтальную прокрутку текста. Эти свойства называются `hscroll` и `maxhscroll`. Горизонтальная прокрутка может использоваться, например, для вывода информации бегущей строкой.

Свойство *hscroll*

Свойство `hscroll` определяет ширину текста, который срезается левой границей текстового блока. В отличие от свойства `scroll`, значение которого измеряется в строках, значение свойства `hscroll` измеряется в пикселах и представляет собой целое неотрицательное число. Диапазон изменения значений данного свойства лежит между 0 и значением свойства `maxhscroll`.

Свойство `hscroll` может использоваться как для чтения, так и для записи. Увеличение значения данного свойства смещает текст влево (прокрутка слева направо), уменьшение значения данного свойства смещает текст вправо (прокрутка справа налево).

Свойство *maxhscroll*

Свойство `maxhscroll` определяет, какая часть текста должна быть срезана левой границей текстового блока для того, чтобы в текстовом блоке стал виден правый конец строки. Значение данного свойства соответствует максимальному значению, которое может принимать свойство `hscroll`. Данное свойство используется только для чтения.

В листинге 14.3 приведен пример, иллюстрирующий возможность вывода текста бегущей строкой с помощью свойств `hscroll` и `maxhscroll`. В данном случае текст движется справа налево (прокрутка слева направо). Когда последний символ в строке попадает в пределы текстового блока, движение начинается сначала. Для того чтобы такое зацикливание происходило плавно, без резкого скачка, в конец строки, выводимой в текстовый блок `txt`, можно поместить несколько пробелов.

Листинг 14.3. Вывод текста бегущей строкой

```
this.createTextField("txt", this.getNextHighestDepth(), 0, 0, 150, 20);
txt.border = true;
txt.multiline = false;
txt.wordWrap = false;
txt.text = "Создайте систему, которой сможет пользоваться даже дурак,
и только дурак захочет ею пользоваться. ";
this.onEnterFrame = function() {
       if (txt.hscroll != txt.maxhscroll) {
              txt.hscroll += 3;
       }
       else {
              txt.hscroll = 0;
       }
};
```

Контроль прокрутки содержимого текстового блока

В классе TextField имеется обработчик события onScroller, позволяющий контролировать изменение значений свойств управления прокруткой текста. Событие onScroller генерируется всякий раз, когда происходит изменение значения свойств scroll, maxscroll, bottomScroll, hscroll или maxhscroll. Обработчик onScroller реагирует на изменение значений свойств управления прокруткой вне зависимости от того, было ли это изменение вызвано действиями пользователя (прокрутка текста, ввод текста, вырезание/вставка текста) или результатом действия сценария (прокрутка, вывод или удаление текста).

Для обработки данного события можно использовать метод обработчика экземпляра текстового блока или объекта-приемника. При этом используется следующий стандартный синтаксис:

```
текстовый блок.onScroller = function(ссылка){
...
};
объект-приемник.onScroller = function(ссылка){
...
};
```

В качестве параметра методу обработчика onScroller передается ссылка на экземпляр текстового блока, для которого определен данный метод или который является источником данного события. Эту ссылку можно использовать в теле функции, присваиваемой свойству onScroller для обращения к текстовому блоку, с помощью которого контролируется соответствующее событие. Следующий код выводит номер самой верхней видимой строки текстового блока txt на протяжении прокрутки его содержимого.

```
txt.onScroller = function(scrollTxt:TextField){
       trace(scrollTxt.scroll + "-я строка");
};
```

Этот же результат может быть получен в результате трансляции события onScroller объектом-источником TextField (txt) объекту-приемнику myListener, который осуществляет обработку данного события:

```
var myListener:Object = new Object();
txt.addListener(myListener);
myListener.onScroller = function(scrollTxt){
       trace(scrollTxt.scroll + "-я строка");
};
```

Пример, демонстрирующий применение данного метода, будет рассмотрен в листинге 14.5.

Для контроля выполнения прокрутки с помощью колеса мыши используется обработчик события onMouseWheel класса Mouse, описанный в *главе 17*.

Примеры реализации текстовой прокрутки

В данном разделе мы рассмотрим два подхода к реализации прокрутки. Первый из них заключается в создании двух кнопок для прокрутки текста вверх и вниз. При нажатии на кнопку прокрутка в соответствующем направлении выполняется непрерывно с заданной скоростью. Во втором примере будет предложен способ создания стандартной полосы прокрутки, снабженной бегунком, перемещая который можно выполнять прокрутку текста в вертикальном направлении.

В Flash MX 2004 имеются встроенные компоненты UIScrollBar и TextArea, которые обладают функциональностью, позволяющей выполнять прокрутку текстового содержимого. Для чего же нужно вновь изобретать велосипед, создавая пользовательские элементы прокрутки? Дело в том, что применение встроенного компонента увеличивает конечный объем документа на несколько десятков килобайт, в то время как во многих случаях постановка задачи требует использования лишь ограниченного набора возможностей данного компонента. Получается, что разработчик вынужден платить за то, чем он не воспользуется. Пользовательская полоса прокрутки, процесс создания которой будет описан в этом разделе, занимает чуть более 1 Кбайт. В связи с этим при выборе средств решения поставленной задачи необходимо исходить из конкретной ситуации. Если речь идет о создании элементарной прокрутки текста, от которой не требуется дополнительных возможностей и изощренной функциональности, может быть, имеет смысл затратить усилия на создание собственного механизма, ориентированного на конкретную задачу, который будет обладать только необходимыми возможностями и занимать минимальный объем. Если же целью является создание комплексного проекта, содержащего многочисленные текстовые поля, которые должны взаимодействовать друг с другом и с другими элементами интерфейса и обладать широкой функциональностью, можно прибегнуть к использованию готовых средств в ущерб объему конечного файла.

Организация непрерывной построчной прокрутки текста

Вначале рассмотрим более простой пример организации вертикальной прокрутки текста. Создадим динамический текстовый блок и присвоим ему имя `txt`. Текст в этот блок помещается программным образом. Для обеспечения возможности прокрутки выводимого в нем текста нам потребуется два клипа, исполняющих роли кнопок для прокрутки текста вверх и вниз. Присвоим этим клипам имена `scrollUpBut` и `scrollDownBut` соответственно. Задача заключается в том, чтобы прокрутка текста в блоке `txt` в определенном направлении (вверх или вниз) выполнялась непрерывно на протяжении всего времени, пока соответствующая кнопка остается нажатой.

Для решения данной задачи создадим вспомогательную функцию `scrollText()`, выполняющую прокрутку текста в блоке `txt` на одну строку в заданном направлении. Функция принимает в качестве параметра строку "`up`" или "`down`", указывающую направление прокрутки вверх или вниз. В принципе вместо строкового параметра можно было бы использовать флаг, принимающий булево значение, однако применение строкового параметра делает код более понятным и "говорящим".

Теперь для выполнения прокрутки в заданном направлении необходимо при нажатии соответствующей кнопки осуществлять периодический вызов функции `scroll-`

Text(), передавая ей параметр, определяющий направление прокрутки. Для этой цели воспользуемся глобальной функцией setInterval(). При нажатии на кнопку прокрутки будем осуществлять вызов функции setInterval(), передавая ей ссылку на функцию scrollText(), временной интервал, через который эта функция должна периодически вызываться, и строковый параметр, задающий направление прокрутки, который будет передаваться функции scrollText() при каждом ее вызове.

Для того чтобы при отпускании кнопки мыши прокрутка прекращалась, воспользуемся функцией clearInterval(), прекращая периодический вызов функции scrollText(), инициированный нажатием одной из кнопок прокрутки. Поскольку пользователь может уйти с кнопки с нажатой кнопкой мыши, имеет смысл также поместить фрагмент кода, останавливающий прокрутку при обнаружении события onDragOut.

В листинге 14.4 приводится программный код, реализующий поставленную задачу.

Листинг 14.4. Организация непрерывной построчной прокрутки текста

```
var speed = 50; // Скорость прокрутки
/* Вспомогательная функция, выполняющая прокрутку на одну строку в
заданном направлении */
function scrollText(scrollDirection:String) {
       if (scrollDirection == "down") {
              txt.scroll ++; // Прокрутка вниз
       }
       if (scrollDirection == "up") {
              txt.scroll --; // Прокрутка вверх
       }
       updateAfterEvent();
}
// При нажатии на кнопку, выполняющую прокрутку вниз...
scrollDownBut.onPress = function() {
       /* Выполняем периодический вызов функции, выполняющей прокрутку
       вниз на одну строку */
       intDown = setInterval(scrollText, speed, "down");
};
// При нажатии на кнопку, выполняющую прокрутку вверх...
scrollUpBut.onPress = function() {
       /* Выполняем периодический вызов функции, выполняющей прокрутку
       вверх на одну строку */
       intUp = setInterval(scrollText, speed, "up");
};
// При отпускании кнопки прекращаем прокрутку вниз
scrollDownBut.onRelease = function() {
```

```
        clearInterval(intDown);
};
/* При уходе курсора мыши за пределы кнопки прокрутки прекращаем
прокрутку вниз */
scrollDownBut.dragOut = scrollDownBut.onRelease;
// При отпускании кнопки прекращаем прокрутку вверх
scrollUpBut.onRelease = function() {
        clearInterval(intUp);
};
/* При уходе курсора мыши за пределы кнопки прокрутки прекращаем
прокрутку вверх */
scrollUpBut.dragOut = scrollUpBut.onRelease;
```

Создание вертикальной полосы прокрутки текста

Рассмотрим пример, иллюстрирующий создание стандартной вертикальной полосы прокрутки текста с помощью бегунка. Внешний вид такой полосы представлен на рис. 14.7. Для реализации данного примера нам понадобится текстовый блок с именем `txt`. Полоса прокрутки должна обладать функциональностью, позволяющей ей прокручивать не только динамический, но также пользовательский текст. При этом необходимо обеспечить автоматическое смещение бегунка при вводе текста с клавиатуры, при прокрутке с помощью клавиатуры и при выводе текста программным образом.

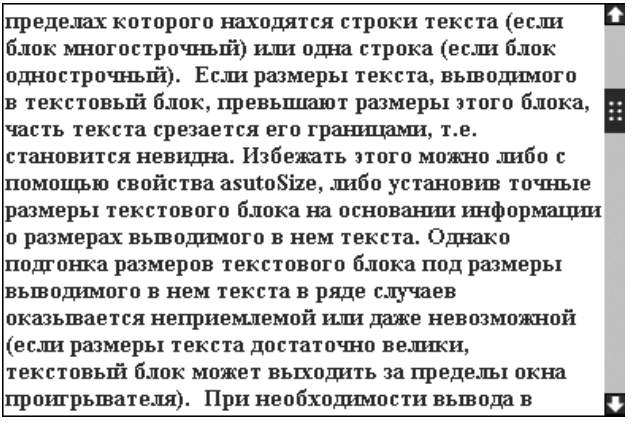


Рис. 14.7. Вертикальная полоса прокрутки

Полоса прокрутки будет включать следующие элементы.

- Символ `scrollBar`, представляющий саму полосу прокрутки, по которой будет перемещаться бегунок. В приведенном примере это просто прямоугольник. Вы-

сота и ширина этого символа не имеют значения, поскольку они будут установлены программным образом.

- Символ `slider`, представляющий бегунок, перемещая который, пользователь будет выполнять прокрутку текста.
- Символы `scrollUpBut` и `scrollDownBut`, представляющие кнопки, расположенные вверху и внизу полосы прокрутки соответственно. При нажатии одной из этих кнопок текст будет прокручиваться на одну строку в соответствующем направлении.

Все дальнейшие расчеты выполняются исходя из того, что точки регистрации всех описанных клипов находятся в их центрах.

Местоположение всех описанных элементов на сцене может быть произвольным, поскольку они будут программным образом позиционированы так, что полоса прокрутки будет расположена справа от текстового блока.

Следует иметь в виду, что дробные координаты и размеры объектов могут привести к накоплению погрешности, что, в свою очередь, может привести к некорректной работе. В связи с этим необходимо позиционировать текстовый блок так, чтобы его координаты представляли собой целочисленные значения. Это же относится к высоте текстового блока и высоте символов `slider`, `scrollUpBut` и `scrollDownBut`.

Ширина всех описанных элементов задается программным образом, так, чтобы значение свойства `_width` для каждого из них было одинаково и соответствовало значению переменной `width`, задающей ширину полосы прокрутки.

Позиционирование объектов выполняется в соответствии с рис. 14.7. Выполнение этой процедуры программным образом позволяет избежать проблем с выравниванием и установкой целочисленных значений координат в рабочей среде. Следует отметить, что по умолчанию бегунок должен быть расположен вверху полосы прокрутки, и первая строка текста должна быть видна.

Для выполнения прокрутки текста при смещении бегунка используется следующий алгоритм.

1. В переменной `initY` сохраняем исходную координату бегунка по *Y*. Это значение соответствует крайнему верхнему положению бегунка.
2. Определяем максимальное расстояние, на которое можно перемещать бегунок. Это расстояние определяется как промежуток между нижней границей бегунка и верхней границей кнопки `scrollDownBut`. Полученный результат сохраняем в переменной `scrollDist`.
3. Определяем, на какое расстояние необходимо сместить бегунок для прокрутки текста на одну строку. Для этого значение переменной делим на максимальное значение, которое может принимать свойство `scroll`. Результат сохраняем в переменной `scrollStep`:

   ```
   var scrollStep = scrollDist / (txt.maxscroll -1);
   ```

4. Организация перетаскивания бегунка реализуется с помощью метода `startDrag()`, причем, поскольку бегунок должен смещаться только по вертикали, задаем область перетаскивания таким образом, чтобы левая и правая ее границы совпадали с горизонтальной координатой клипа `slider`. Вертикальные

границы области перетаскивания задаются таким образом, чтобы клип slider не мог выйти за пределы кнопок scrollUpBut и scrollDownBut. Поскольку перетаскивание слайдера должно выполняться только при нажатии на него, вызов метода осуществляется только при обнаружении события onPress.

5. Для того чтобы связать состояние вертикальной прокрутки текста в текстовом блоке с вертикальной координатой клипа slider, используется следующий код:

```
this.onMouseMove = function() {
        txt.scroll = (slider._y - initY) / scrollStep + 1;
        updateAfterEvent();
        };
};
```

Этот код выполняется только при нажатии на клип slider; в связи с этим значение свойства scroll изменяется при перетаскивании этого клипа.

6. При отпускании клипа прекращаем перетаскивание с помощью метода stopDrag() и удаляем обработчик onMouseMove, определенный при его нажатии. В результате этих действий прокрутка прекращается.

Для выполнения прокрутки при нажатии на кнопку scrollUpBut или на кнопку scrollDownBut необходимо просто изменять значение свойства scroll на единицу. Единственный нюанс, который нужно предусмотреть, заключается в необходимости смещения бегунка при прокрутке текста с помощью кнопок. Однако и эта задача может быть достаточно просто решена. При нажатии на кнопку прокрутки необходимо смещать клип slider на расстояние, равное значению переменной scrollStep, в направлении, соответствующем направлению прокрутки. Для того чтобы бегунок не смог выйти за пределы полосы прокрутки, необходимо ввести проверочное условие, контролирующее состояние прокрутки текста. Если достигнута первая или последняя строка, бегунок должен быть помещен в крайнее верхнее или нижнее положение.

Необходимо учесть, что прокрутка может также выполняться с помощью клавиш управления курсором клавиатуры, а также при добавлении или удалении новых строк пользователем или внешним сценарием. Любая из этих операций должна приводить к смещению бегунка полосы прокрутки в соответствии с текущим значением свойства scroll.

Смещение бегунка вследствие прокрутки текста посредством клавиатуры и вывода текста программным образом реализуется с помощью обработчика события onScroller, которое генерируется при каждом изменении значения свойства scroll. В результате изменения количества строк текста, необходимо заново выполнить расчет значения scrollStep, поскольку значение свойства maxscroll могло измениться. Такая автоматическая корректировка положения бегунка должна выполняться только в том случае, если бегунок не перетаскивается вручную. Для того чтобы исключить влияние блока кода, помещенного в обработчик события onScroller, и блока кода, выполняющегося при перетаскивании бегунка, была введена переменная-флаг autoScrolling, принимающая значение false только в процессе выполнения прокрутки путем перемещения бегунка.

Программный код, реализующий описанный механизм прокрутки текста, представлен в листинге 14.5.

Листинг 14.5. Создание полосы прокрутки текста

```
var width = 28; // Ширина полосы прокрутки

// Установка равной ширины для всех элементов полосы прокрутки

scrollUpBut._width = scrollDownBut._width = slider._width =
scrollBar._width = width;

/* Позиционируем все элементы полосы прокрутки справа от текстового поля,
выравнивая их по горизонтали */

scrollUpBut._x = scrollDownBut._x = slider._x = scrollBar._x = txt._x +
txt._width + .5 * width;

// Позиционируем верхнюю кнопку полосы прокрутки

scrollUpBut._y = txt._y + .5 * scrollUpBut._height;

// Позиционируем нижнюю кнопку полосы прокрутки

scrollDownBut._y = txt._y + txt._height - .5 * scrollDownBut._height;

// Позиционируем бегунок вверху полосы прокрутки

slider._y = scrollUpBut._y + .5 * scrollUpBut._height + .5 *
slider._height;

// Сохраняем координату начального положения бегунка

var initY = slider._y;

// Максимальное расстояние в пикселах, на которое может смещаться
// бегунок.

var scrollDist = scrollDownBut._y - .5 * scrollDownBut._height -
scrollUpBut._y - .5 * scrollUpBut._height - slider._height;

/* Расстояние в пикселах, на которое нужно сместить бегунок для прокрутки
текста на одну строку */

var scrollStep = scrollDist / txt.maxscroll;

/* Флаг, используемый для определения того, выполняется прокрутка с
помощью полосы, или путем перемещения текстового курсора посредством
клавиатуры. Если флаг autoScrolling установлен в true - прокрутка
клавиатурой или вследствие изменения числа строк, если autoScrolling
установлен в false - прокрутка с помощью бегунка */

var autoScrolling = true;

/*----------ОРГАНИЗАЦИЯ ПРОКРУТКИ С ПОМОЩЬЮ БЕГУНКА-------------------*/

// Перетаскивание бегунка при нажатии на клип slider.

slider.onPress = function() {

/* Перемещение бегунка может осуществляться только в вертикальном
направлении. Область перетаскивания сверху и снизу ограничена кнопками
прокрутки scrollUpBut и scrollDownBut */
```

```
this.startDrag(false, this._x, scrollUpBut._y + .5 * ↳
scrollUpBut._height + .5 * this._height, this._x, scrollDownBut._y -.5 *↳
scrollDownBut._height - Math.round(.5 * this._height));
        // Переключаем флаг, так как прокрутка выполняется вручную
        autoScrolling = false;
        // Выполнение прокрутки текста при смещении бегунка
        this.onMouseMove = function() {
                txt.scroll = (slider._y - initY) / scrollStep;
                updateAfterEvent();
        };
};
// При отпускании кнопки мыши прекращаем перетаскивание и прокрутку
slider.onRelease = function() {
        this.stopDrag();
        // Переключаем флаг, так как прокрутка может выполняться
        // клавиатурой
        autoScrolling = true;
        // Прекращение прокрутки
        delete this.onMouseMove;
};
//----------------------ПРОКРУТКА С ПОМОЩЬЮ КНОПОК-----------------------
//Прокрутка вверх на одну строку при каждом нажатии на кнопку scrollUpBut
scrollUpBut.onPress = function() {
        txt.scroll--;
        // Смещение бегунка при выполнении прокрутки с помощью кнопок
        if (txt.scroll == 1) {
                // Если первая строка текста видна, останавливаем бегунок
                slider._y = initY;
        }
        else {
                // Если первая строка еще не видна, смещаем бегунок
                slider._y -= scrollStep;
        }
};
/* Аналогично выполняем прокрутку вниз на одну строку при каждом нажатии
на кнопку scrollDownBut */
scrollDownBut.onPress = function() {
        txt.scroll++;
        if (txt.scroll == txt.maxscroll) {
```

```
                // При достижении нижней строки останавливаем бегунок внизу
                полосы прокрутки.
                slider._y = scrollDownBut._y - .5 * scrollDownBut._height-↳
                .5 * slider._height;
        }
        else {
                slider._y += scrollStep;
        }
};
/* СМЕЩЕНИЕ БЕГУНКА ПРИ ВЫПОЛНЕНИИ ПРОКРУТКИ ПУТЕМ ПЕРЕМЕЩЕНИЯ КУРСОРА С
ПОМОЩЬЮ КЛАВИАТУРЫ И ВСЛЕДСТВИЕ ВЫВОДА ТЕКСТА ПРОГРАММНЫМ ОБРАЗОМ */
txt.onScroller = function(){
/* Бегунок должен смещаться при изменении значения свойства scroll
только в том случае, если прокрутка выполняется с помощью клавиатуры
вследствие ввода текста пользователем или вывода текста программным
образом. Если autoScrolling установлен в true, значит, бегунок не
перемещается вручную. */
if(autoScrolling){
/* Рассчитываем новое значение расстояния, на которое нужно сместить
бегунок для прокрутки текста на одну строку, исходя из того, что возможно
были добавлены/удалены строки и исходная высота текста изменилась */
scrollStep = scrollDist / (txt.maxscroll-1);
 if (txt.scroll == 1) {
        // Если первая строка текста видна, перемещаем бегунок вверх
        slider._y = initY;
}
else if (txt.scroll == txt.maxscroll && txt.maxscroll>1) {
        // При достижении нижней строки перемещаем бегунок в самый низ
        // полосы прокрутки.
        slider._y = scrollDownBut._y - .5 * scrollDownBut._height - .5 *↳
        slider._height;
        }
// Если ни первая, ни последняя строки не видны, корректируем положение
// бегунка в соответствии со значением свойства scroll.
else {
        slider._y = (this.scroll-1)*scrollStep + initY;
        }
};
```

В качестве упражнения читатель может самостоятельно объединить функциональность сценариев, приведенных в листингах 14.4 и 14.5 так, чтобы с помощью кнопок прокрутки `scrollUpBut` и `scrollDownBut` можно было выполнять непрерывную прокрутку текста.

Контроль выделения и передачи фокуса. Класс *Selection*

При работе с элементами пользовательского интерфейса и особенно с текстовыми полями может возникнуть необходимость в контроле и управлении выделением текста и передачей фокуса текстовому полю или иному объекту. Для контроля выделения текста используется встроенный класс `Selection`.

Под фокусом понимается активное состояние определенного элемента пользовательского интерфейса. Передача фокуса объекту может осуществляться пользователем или с помощью программного кода. ActionScript позволяет контролировать передачу фокуса текстовому блоку, кнопке, клипу или компоненту.

Класс `Selection` позволяет контролировать выделение текста, расположенного в текстовом блоке любого типа: статическом (только в выделяемом), динамическом и пользовательском. С его помощью также можно управлять выделением и передачей фокуса тексту.

Класс `Selection` не требует создания экземпляра, все его методы являются статическими, то есть вызываются непосредственно из класса. Это связано с тем, что в каждый момент времени в окне проигрывателя в фокусе может находиться только одно текстовое поле.

Передача фокуса

Для управления передачей фокуса можно воспользоваться методом `Selection.setFocus()`. С помощью этого метода можно осуществить передачу фокуса не только экземпляру текстового блока, но также кнопке или клипу. Метод `setFocus()` имеет следующий синтаксис:

`Selection.setFocus(`*`объект`*`)`

Параметр *`объект`* — это экземпляр текстового блока, кнопки или клипа, которому передается фокус. Данный параметр может быть задан строкой, содержащей путь к объекту, или выражением, возвращающим ссылку на объект (в документации к Flash ошибочно написано, что этот параметр задается только строковым литералом).

Если в качестве параметра методу `setFocus()` передается значение `null`, выполняется удаление текущего фокуса.

Метод `setFocus()` возвращает значение `true`, если фокус успешно передан требуемому объекту, и значение `false`, если этого не произошло. Следует иметь в виду, что для корректной работы данного метода перед его вызовом необходимо передать фокус окну проигрывателя Flash Player. Для этого достаточно просто щелкнуть в любой точке в пределах окна проигрывателя. До тех пор, пока окно проигрывателя не получит фокус, метод `setFocus()` будет возвращать значение `false`, что может привести к некорректной работе (или неработе) программы. Для того чтобы заставить пользователя передать фокус Flash-фильму, проще всего предложить ему нажать на кнопку, выполняющую переход к требуемому разделу фильма.

Так, например, требуется передать фокус текстовому блоку с именем `txt` и поместить в него текстовый курсор. Следующий код, помещенный в кадр, не даст желае-

мого результата, поскольку к моменту его выполнения фокус не передан окну проигрывателя.

```
Selection.setFocus(txt);
```

Если в процессе загрузки документа (например, в режиме **View>Simulate Download** среды тестирования) до выполнения этого кода успеть щелкнуть внутри окна проигрывателя, то требуемый результат будет достигнут. Однако более надежно было бы создание кнопки с именем `but` и применение следующего сценария:

```
but.onRelease = function(){
        Selection.setFocus(txt);
};
```

Передача фокуса клипу может оказаться необходимой для обеспечения возможности применения методов обработчиков событий `onKeyDown` и `onKeyUp`, определенных для данного клипа. Дело в том, что методы обработчиков данных событий функционируют только в том случае, если вызывающий их клип находится в фокусе. Следует иметь в виду, что передача фокуса клипу может быть выполнена либо при наличии у него хотя бы одного обработчика событий кнопки, либо если его свойство `focusEnabled` установлено в значение `true`.

Свойство `focusEnabled` принимает булево значение `true` или `false`. Если данное свойство клипа установлено в значение `false` или `undefined` (то есть не определено), клип может получать фокус только в том случае, если это клип-кнопка (то есть если для него определен обработчик хотя бы одного события кнопки). Если же свойство `focusEnabled` установлено в значение `true`, то клип может получать фокус даже, если он не является клипом-кнопкой.

Следующий код демонстрирует возможность применения метода обработчика события `onKeyUp`, определенного для клипа с именем `mc`.

```
mc.focusEnabled = true;
Selection.setFocus(mc)
mc.onKeyUp = function(){
        trace("Клавиша отпущена");
};
```

Следует иметь в виду, что фокус может быть очень легко утрачен, например, при перемещении курсора мыши.

В листинге 14.6 приведен сценарий, демонстрирующий возможность применения метода `setFocus()`. В данном примере имеется четыре пользовательских текстовых блока с именами `txt1`, `txt2`, `txt3` и `txt4`. При нажатии кнопки с именем `submit` выполняется проверка наличия незаполненных полей. При наличии незаполненных полей в динамический текстовый блок с именем `fillOutTxt` выводится сообщение с перечислением номеров незаполненных полей. При этом курсор автоматически перемещается в самое верхнее незаполненное поле.

Листинг 14.6. Проверка наличия незаполненных текстовых полей в форме

```
// Функция, выполняющая проверку того, все ли поля заполнены
function checkMissedFields() {
        // Локальный массив для хранения номеров незаполненных полей
        var missedFields:Array = new Array();
        // Переменная-флаг, показывающая, есть ли среди полей
        // незаполненные
        var missed:Boolean = false;
        // Проход всех полей
        for (i = 1; i < 5; i++) {
                // Если в каком-то поле нет текста...
                if (_root["txt" + i].length == 0) {
                        // Помещаем его номер в массив missedFields
                        missedFields.push(i);
                        // Переключаем флаг
                        missed = true;
                }
        }
        // Если есть незаполненные поля...
        if(missed){
        // Выводим их номера
        fillOutTxt.text = "Необходимо заполнить поля: " +
        missedFields.join(", ");
        // Перемещаем фокус на первое сверху незаполненное поле
        Selection.setFocus(_root["txt" + missedFields[0]]);
        }
        // Если все поля заполнены, выводим соответствующее сообщение
        else {
        fillOutTxt.text = "Спасибо! Все поля заполнены.";
        }
}
// Выполняем проверку при отпускании кнопки
submit.onRelease = checkMissedFields;
```

Обнаружение носителя фокуса

Для того чтобы определить, какой объект в данный момент находится в фокусе, можно воспользоваться методом `getFocus()`.

Этот метод не принимает параметров и имеет следующий синтаксис:

```
Selection.getFocus()
```

Метод `getFocus()` возвращает строку, содержащую путь к экземпляру текстового блока, кнопки или клипа, который находится в фокусе в данный момент.

Если ни один из перечисленных объектов не находится в фокусе, метод `getFocus()` возвращает значение `null`.

Контроль передачи фокуса

Класс `Selection` предоставляет возможность обрабатывать передачу фокуса различным объектам. Всякий раз при изменении фокуса объект `Selection` генерирует событие `onSetFocus`. Поскольку в каждый момент времени в фокусе может находиться только один объект, событие `onSetFocus` генерируется всякий раз при переходе фокуса с одного объекта на другой (текстовый блок, кнопка или клип) или в случае утраты фокуса. Для обработки данного события необходимо создать объект-приемник (listener) и зарегистрировать его на получение сообщений об обнаружении события `onSetFocus` от объекта `Selection` с помощью метода `addListener()`. Синтаксис обработки этого события приведен ниже.

```
var myListener:Object = new Object();
Selection.addListener(myListener);
myListener.onSetFocus = function (oldObj, newObj) {
...
};
```

Методу обработчика события `onSetFocus` в качестве параметров передаются две ссылки: `oldObj` — ссылка на предыдущий объект, удерживающий фокус, и `newObj` — ссылка на объект, получивший фокус. Эти ссылки могут быть использованы в теле метода обработчика для обращения к соответствующим объектам. Если перед передачей фокуса объекту ни один объект не находился в фокусе, параметр `oldObj` принимает значение `null`. Если в результате утраты фокуса объектом фокус не передается другому объекту, параметр `newObj` принимает значение `null`.

Для прекращения трансляции события `onSetFocus` объекту-приемнику можно воспользоваться методом `removeListener()` класса `Selection`.

В листинге 14.7 приведен пример использования обработчика события `onSetFocus`. Представленный в листинге код выделяет любой текстовый блок, принимающий фокус, красной прямоугольной рамкой. При утрате фокуса рамка отключается.

Листинг 14.7. Выделение текстового поля с фокусом прямоугольной рамкой

```
// Установка параметров текстовых блоков
for (i in this) {
       if (this[i] instanceof TextField) {
              this[i].background = true;
```

```
                this[i].backgroundColor = 0xE0E0F0;
                this[i].borderColor = 0xFF0000;
        }
}
var myListener:Object = new Object();
Selection.addListener(myListener);
myListener.onSetFocus = function(oldTxt, newTxt) {
        /* Если ранее в фокусе был текстовый блок, отключаем отображение
        его рамки */
        if (oldTxt instanceof TextField) {
                oldTxt.border = false;
        }
        /* Если фокус передан текстовому блоку, включаем отображение его
        рамки */
        if (newTxt instanceof TextField) {
                newTxt.border = true;
        }
};
```

Определение границ выделения и положения курсора

Для определения границ диапазона текстового выделения можно воспользоваться методами `getBeginIndex()` и `getEndIndex()`.

Метод `getBeginIndex()` возвращает индекс первого символа, входящего в диапазон выделения. Нумерация символов в текстовом блоке осуществляется, начиная с нуля. Если в момент вызова данного метода ни один текстовый блок не находится в фокусе, метод возвращает значение −1. Метод `getBeginIndex()` не принимает параметров и имеет следующий синтаксис:

```
Selection.getBeginIndex()
```

Метод `getEndIndex()` возвращает индекс последнего символа, входящего в диапазон выделения. Если в момент вызова данного метода ни один текстовый блок не находится в фокусе, метод возвращает значение −1. Метод `getEndIndex()` не принимает параметров и имеет следующий синтаксис:

```
Selection.getEndIndex()
```

Индексация символов в текстовом блоке всегда выполняется слева направо и сверху вниз. В связи с этим значение, возвращаемое методом `getBeginIndex()`, не может превышать значение, возвращаемое методом `getEndIndex()`. Это означает, что вне зависимости от того, в каком направлении пользователь выполняет выделение (слева направо или справа налево), метод `getBeginIndex()` всегда указывает индекс

символа, расположенного в диапазоне выделения левее и выше, а метод `getEndIndex()` всегда указывает индекс символа, расположенного в диапазоне выделения правее и ниже.

Если текстовый курсор помещен между символами (то есть текстовый блок — в фокусе), но диапазон выделения отсутствует, то значения, возвращаемые методами `getBeginIndex()` и `getEndIndex()`, будут равны.

Для определения текущего положения текстового курсора можно воспользоваться методом `getCaretIndex()`. Данный метод возвращает индекс символа, перед которым находится текстовый курсор (мигающая вертикальная линия). Если текстовый курсор отсутствует, данный метод возвращает значение −1. Формально значение, возвращаемое методом `getCaretIndex()`, может быть получено из условия равенства значений, возвращаемых методами `getBeginIndex()` и `getEndIndex()`. Метод `getCaretIndex()` не принимает параметров и имеет следующий синтаксис:

```
Selection.getCaretIndex()
```

В листинге 14.8 приведен пример, демонстрирующий возможности использования описанных методов. В рассматриваемом примере в специально созданные динамические поля выводится информация о границах диапазона выделения и текущем положении курсора. Для этих целей используются следующие текстовые блоки:

- `startTxt` — в этот блок выводится начальный индекс диапазона выделения;
- `endTxt` — в этот блок выводится конечный индекс диапазона выделения;
- `totalSelectedTxt` — в этот блок выводится общее количество выделенных символов;
- `caretTxt` — в этот блок выводится индекс текущего положения курсора.

Информация о границах и размере диапазона выделения выводится непосредственно в процессе выделения, а также в случае, если в текстовом блоке имеется выделенный диапазон символов. Текущее положение курсора выводится при его позиционировании с помощью мыши и при вводе символов или перемещении с помощью клавиатуры. Во время выделения текущее положение курсора не указывается. Если ни один текстовый блок не находится в фокусе, все текстовые поля очищаются. Приведенный сценарий действует для любого текстового блока, который находится в кадре.

Листинг 14.8. Определение границ выделения и положения курсора

```
// Создаем объект-приемник для обработки событий мыши
var myListener:Object = new Object();
// Регистрируем myListener на получение событий от объекта Mouse
Mouse.addListener(myListener);
// Регистрируем myListener на получение событий от объекта Key
Key.addListener(myListener);
// При перемещении курсора мыши с нажатой левой кнопкой...
myListener.onMouseDown = function() {
```

```
        myListener.onMouseMove = function() {
                // Если имеется диапазон выделения...
                if (Selection.getBeginIndex() < Selection.getEndIndex()) {
                        // Выводим индекс начала и конца диапазона выделения
                        startTxt.text = Selection.getBeginIndex();
                        endTxt.text = Selection.getEndIndex();
                        // Выводим количество выделенных символов
                        totalSelectedTxt.text = Selection.getEndIndex() -
                        Selection.getBeginIndex();
                        /* Очищаем поле с текущим положением курсора во
                        время выделения */
                        caretTxt.text = "";
                        updateAfterEvent();
                }
        };
};
// При отпускании кнопки мыши...
myListener.onMouseUp = function() {
        // Прекращаем вывод индексов диапазона выделения
        delete myListener.onMouseMove;
        /* Если выделение снято, очищаем текстовые поля. В противном
        случае индексы диапазона выделения остаются в текстовых полях */
        if (Selection.getBeginIndex() == Selection.getEndIndex()) {
                startTxt.text = "";
                endTxt.text = "";
                totalSelectedTxt.text = "";
                // Вывод текущего положения курсора в поле caretTxt
                if (Selection.getCaretIndex() == -1) {
                        // Если текстовое поле не в фокусе, очищаем поле
                        caretTxt.text = "";
                }
                else {
                        /* Если текстовое поле в фокусе, выводим индекс
                        положения курсора */
                        caretTxt.text = Selection.getCaretIndex();
                }
        }
};
```

```
/* Вывод текущего положения курсора в поле caretTxt при нажатии
клавиатурной клавиши */
myListener.onKeyUp = function() {
       if (Selection.getCaretIndex() == -1) {
              caretTxt.text = "";
       }
       else {
              caretTxt.text = Selection.getCaretIndex();
       }
};
```

Управление выделением

Для выполнения выделения диапазона текста программным образом можно воспользоваться методом `setSelection()`. Данный метод имеет следующий синтаксис:

`Selection.setSelection(`*`начальный индекс`*`, `*`конечный индекс`*`)`

Параметр *`начальный индекс`* задает индекс первого символа, входящего в диапазон выделения.

Параметр *`конечный индекс`* задает индекс последнего символа, входящего в диапазон выделения.

Оба параметра задаются целыми неотрицательными числами, которые могут передаваться как числовые литералы или выражения, возвращающие числа.

В результате применения данного метода в текстовом блоке, который в момент его вызова находился в фокусе, выделяется диапазон текста, заключенный между символами, индексы которых заданы параметрами *`начальный индекс`* и *`конечный индекс`*. Если в момент вызова метода `setSelection()` ни один текстовый блок не находится в фокусе, метод не дает никакого результата. Поэтому перед вызовом метода `setSelection()` следует удостовериться в том, что текстовый блок, в котором необходимо выполнить выделение, находится в фокусе, или принудительно передать ему фокус с помощью метода `setFocus()`.

Следующий код создает текстовый блок и выделяет в нем слово "должна".

```
this.createTextField("txt", this.getNextHighestDepth(), 0, 0, 1, 1);
txt.autoSize = true;
txt.text = "Машина должна работать, человек - думать";
Selection.setFocus(txt);
Selection.setSelection(7, 13);
```

Управление порядком переноса фокуса с помощью клавиши табуляции

Перенос фокуса с одного объекта на другой может выполняться с помощью нажатия клавиши <Tab>. При этом важно определить, какие элементы могут получать фокус и в какой последовательности он будет передаваться.

Включение объектов в автоматический порядок переноса фокуса

При использовании автоматического порядка переноса фокуса он передается слева направо и сверху вниз в соответствии с местоположением объектов на сцене. При этом иерархия их вложенности не оказывает влияния на порядок переноса фокуса.

Для включения объекта в автоматический порядок переноса фокуса можно воспользоваться свойством `tabEnabled`. Свойство `tabEnabled` имеется у объектов `TextField`, `Button` и `MovieClip`.

Данное свойство принимает булево значение `true` или `false`. Если его значение равно `true` или не определено (то есть принимает значение `undefined`), объект включен в автоматический порядок переноса фокуса. При этом следует иметь в виду, что если значение свойства `tabEnabled` клипа равно `undefined`, этот клип включается в автоматический порядок переноса фокуса только в случае, если для него определен хотя бы один обработчик события кнопки (например, `onPress`, `onRelease` и т. д.).

Если значение свойства `tabEnabled` установлено в `false`, объект не включается ни в автоматический, ни в пользовательский порядок переноса фокуса с помощью клавиши <Tab>.

Существует возможность определить, будут ли объекты, расположенные на монтажной линейке клипа, включены в автоматический порядок переноса фокуса. Для этих целей используется свойство `tabChildren`, определенное для объекта `MovieClip`. Если данное свойство принимает значение `true` или `undefined` (не определено), объекты, расположенные на монтажной линейке соответствующего клипа, принимают участие в автоматическом порядке переноса фокуса. Если значение данного свойства равно `false`, содержимое клипа не участвует в автоматическом порядке переноса фокуса.

Визуальное выделение фокуса

При передаче фокуса пользовательскому текстовому блоку происходит автоматическое выделение его содержимого. Если текстовый блок не содержит текста, то в нем появляется мигающий текстовый курсор. Для того чтобы визуально отразить передачу фокуса кнопке или клипу, можно воспользоваться свойством `_focusrect`. Данное свойство определено для объектов `Button` и `MovieClip`, его также можно использовать как глобальное свойство. Применение данного свойства позволяет отображать желтую рамку вокруг клипа или кнопки, которой был передан фокус. Свойство `_focusrect` принимает значение `true` или `false`.

Если значение данного свойства установлено в `true`, то при передаче фокуса соответствующему клипу или кнопке этот объект выделяется желтой рамкой. Если свойство `_focusrect` кнопки или клипа принимает значение `false`, при переносе фокуса на соответствующий объект рамка не появляется.

Следующая строка кода отключает рамку фокуса для клипа с именем `mc`.

```
mc._focusrect = false;
```

Глобальное свойство действует аналогичным образом, однако его действие распространяется сразу на все клипы и кнопки, используемые в фильме, в то время как обращение к одноименному свойству экземпляра оказывает воздействие только на данный экземпляр. Свойства `MovieClip._focusrect` и `Button._focusrect` имеют приоритет над глобальным свойством `_focusrect`.

Управление порядком передачи фокуса

Как уже было сказано ранее, автоматический порядок переноса фокуса с помощью клавиши <Tab> определяется, исходя из расположения объектов на сцене. Такой порядок не всегда соответствует поставленным задачам, поэтому разработчику предоставляется возможность определить пользовательский порядок переноса фокуса по табуляции. Для этих целей используется свойство `tabIndex`, определенное для объектов `TextField`, `Button` и `MovieClip`.

Свойство `tabIndex` позволяет задать индекс, определяющий порядок переноса фокуса по табуляции. Каждому экземпляру текстового блока, кнопки или клипа может быть сопоставлен числовой индекс. Перенос фокуса выполняется в соответствии со значениями данного индекса. При этом иерархическая вложенность объектов не учитывается. Вне зависимости от того, что экземпляры разных объектов могут находиться на разных монтажных линейках, перенос фокуса от одного объекта к другому выполняется, исходя из значения заданного для него индекса. Индекс задается целым положительным числом, для каждого объекта его значение должно быть уникальным. Перенос фокуса выполняется в восходящем порядке, то есть вначале фокус переходит на объект с индексом 1, затем на объект с индексом 2 и т. д.

В случае применения пользовательского порядка переноса фокуса автоматический перенос фокуса с помощью клавиши <Tab> не выполняется. Свойство `tabChildren` при этом также не оказывает никакого действия.

Приведенный ниже код задает порядок передачи фокуса для клипов-кнопок с именами `mc1`, `mc2`, ..., `mc7`. Первым фокус получает клип `mc7`, затем `mc6` и т. д.

```
maxNum = 7; // Количество клипов
for (i = 0; i <= maxNum; i++) {
        this["mc" + i].onPress = function() {
                trace(this._name);
        };
        this["mc" + i].tabIndex = maxNum - i;
}
```

Обработка событий текстового блока

Класс TextField позволяет контролировать четыре события, связанных с текстовыми блоками. Для обработки данных событий можно использовать соответствующие методы обработчиков, определенные для конкретного экземпляра текстового блока, или объекты-приемники (listeners), получающие сообщения о наступлении событий от объекта TextField. Для регистрации объекта-приемника на получение уведомления о наступлении события, обнаруживаемого объектом TextField, используется метод addListener(), определенный в классе TextField. Удалить объект-приемник из списка "слушающих" объектов объекта TextField можно с помощью метода removeListener().

Контроль изменения содержимого текстового блока

Для обнаружения изменения содержимого пользовательского текстового блока в результате действий пользователя применяется метод обработчика события onChanged. Событие onChanged генерируется объектом TextField в следующих случаях:

- при вводе пользователем текста с помощью клавиатуры;
- при удалении выделенного текста из пользовательского текстового блока с помощью команды контекстного меню **Delete** или клавиши <Delete>;
- при вырезании/вставке текста из/в пользовательский текстовый блок.

Следует иметь в виду, что событие onChanged генерируется только при изменении содержимого текстового блока, выполняемом в результате действий пользователя. Изменение текстового содержимого, выполняемое программным образом, не приводит к генерированию данного события.

Для обработки данного события можно использовать метод обработчика экземпляра текстового блока или объекта-приемника. При этом используется следующий стандартный синтаксис:

```
текстовый блок.onChanged = function(ссылка){
...
};
объект-приемник.onChanged = function(ссылка){
...
};
```

В качестве параметра методу обработчика onChanged передается ссылка на экземпляр текстового блока, для которого определен данный метод или который является источником данного события. Эту ссылку можно использовать в теле функции, присваиваемой свойству onChanged для обращения к текстовому блоку, с помощью которого контролируется соответствующее событие.

Следующий код выводит в панель **Output** сообщение при каждом изменении содержимого текстового блока с именем txt:

```
txt.onChanged = function (txtField:TextField){
       trace("Содержимое блока " + txtField + " изменено");
};
```

Тот же результат можно получить, используя текстовый блок txt в качестве источника события onChanged, обработка которого выполняется объектом-приемником с именем myListener.

```
var myListener:Object = new Object();
txt.addListener(myListener);
myListener.onChanged = function(txtField:TextField) {
        trace("Содержимое блока " + txtField + " изменено");
};
```

Контроль передачи и удаления фокуса

Для контроля передачи фокуса экземпляру текстового блока можно воспользоваться методом обработчика события onSetFocus, принадлежащего классу TextField. Событие onSetFocus генерируется экземпляром текстового блока в момент передачи ему фокуса. Методу обработчика onSetFocus в качестве параметра передается ссылка на объект, удерживающий фокус до его передачи текстовому блоку. Эту ссылку можно использовать в теле функции, присваиваемой обработчику onSetFocus. Если перед передачей фокуса текстовому блоку ни один объект не находился в фокусе, данный параметр принимает значение null. Следующий код при передаче фокуса текстовому блоку с именем txt выводит в панель **Output** соответствующее сообщение и ссылку на предыдущий объект, удерживающий фокус:

```
txt.onSetFocus = function(oldObj:Object) {
        trace("Фокус с " + oldObj + " перешел на txt");
};
```

Данное событие автоматически не транслируется объектом TextField. Для трансляции этого события другим объектам-приемникам необходимо использовать класс AsBroadcaster, описанный в *главе 11*.

Для контроля перехода фокуса с экземпляра текстового блока на другой объект (или просто утраты фокуса) используется метод обработчика события onKillFocus, принадлежащего классу TextField. Событие onKillFocus генерируется экземпляром текстового блока при утрате им фокуса. Методу обработчика onKillFocus в качестве параметра передается ссылка на объект, принимающий фокус после его утраты текстовым блоком. Эту ссылку можно использовать в теле функции, присваиваемой обработчику onKillFocus. Если после утраты фокуса текстовым блоком его не получает другой объект, данный параметр принимает значение null. Следующий код при переходе фокуса с текстового блока txt выводит в панель **Output** соответствующее сообщение и ссылку на текущий объект, удерживающий фокус:

```
txt.onKillFocus = function(newObj:Object) {
        trace("Фокус с txt перешел на " + newObj);
};
```

Данное событие автоматически не транслируется объектом TextField. Для трансляции этого события другим объектам-приемникам необходимо использовать класс AsBroadcaster, описанный в *главе 11*.

Примечание

Обработчики `onSetFocus` и `onKillFocus` также имеются у объектов `Button` и `MovieClip`. С помощью этих обработчиков можно осуществлять контроль получения и удаления фокуса экземплярами кнопок и клипов. Их назначение и применение аналогичны применению и назначению одноименных обработчиков объекта `TextField`.

Пример, демонстрирующий применение описанных в этом разделе обработчиков событий, приведен в листинге 14.9. В данном примере показана функция `addInputTextField()`, создающая пользовательское текстовое поле, в которое автоматически выводится текстовая подсказка, объясняющая его назначение. Если пользователь передает фокус данному полю, подсказка исчезает, предоставляя возможность ввести требуемые данные (это похоже на использование поля **Instance Name** в панели **Properties**). Если после утраты фокуса поле остается пустым, в нем вновь выводится подсказка. Цвет подсказки должен отличаться от цвета текста, вводимого пользователем, а все остальные параметры форматирования должны совпадать. Наша функция принимает следующие параметры:

- `txt` — строка, представляющая собой текст подсказки;
- `x`, `y` — координаты текстового поля;
- `w`, `h` — размеры текстового поля;
- `textFormat` — объект `TextFormat`, содержащий все параметры форматирования для текста, вводимого пользователем и для выводимой автоматически подсказки. Значение свойства `color` данного объекта действует только на текст, вводимый пользователем;
- `hintColor` — цвет текста подсказки.

Листинг 14.9. Создание текстового поля для ввода данных с автоматической подсказкой

```
function addInputTextField(txt, x, y, w, h, textFormat, hintColor) {
        /* Переменная для хранения номера ближайшего верхнего свободного
        уровня */
        var depth = this.getNextHighestDepth();
        // Переменная для хранения исходного значения цвета текста
        var originalColor = textFormat.color;
        /* Создаем пользовательский текстовый блок и выводим в него текст
        с подсказкой */
        this.createTextField("txt" + depth, depth, x, y, w, h);
        this["txt" + depth].type = "input";
        this["txt" + depth].text = "<" + txt + ">";
        /* Изменяем исходное значение свойства color объекта textFormat в
        соответствии с указанным цветом отображения подсказки */
        textFormat.color = hintColor;
```

```
// Применяем к тексту подсказки форматирование
this["txt" + depth].setTextFormat(textFormat);
/* Устанавливаем свойство color объекта textFormat в исходное
значение для отображения текста, вводимого пользователем */
textFormat.color = originalColor;
// Задаем параметры форматирования для вновь вводимого текста
this["txt" + depth].setNewTextFormat(textFormat);
// При передаче фокуса текстовому блоку...
this["txt" + depth].onSetFocus = function() {
        // Если в нем отображается подсказка...
        if (!this.changed) {
                // Удаляем текст подсказки
                this.text = "";
                // Помещаем курсор в текстовое поле
                Selection.setFocus(this);
        }
};
// При изменении содержимого текстового блока...
this["txt" + depth].onChanged = function() {
/* Устанавливаем флаг changed в состояние true. Переменная changed
принимает значение false или undefined, если в текстовом блоке
отображается текст подсказки, и значение false, если в текстовом
блоке содержится текст, введенный пользователем. Этот флаг должен
быть создан для каждого текстового блока, создаваемого с помощью
функции addInputTextField(). Поэтому переменная changed создается
как свойство экземпляра текстового блока (то есть свойство объекта
TextField), для чего и используется идентификатор this. */
        this.changed = true;
};
// При утрате фокуса...
this["txt" + depth].onKillFocus = function() {
        // Проверяем, если пользователь не ввел текст...
        if (this.length == 0) {
                // Выводим в текстовый блок подсказку
                this.text = "<" + txt + ">";
                // Устанавливаем цвет для подсказки
                textFormat.color = hintColor;
                this.setTextFormat(textFormat);
                /* Устанавливаем цвет для текста, который будет
                вводиться пользователем */
                textFormat.color = originalColor;
```

```
                    this.setNewTextFormat(textFormat);
                    /* Переключаем флаг в состояние, соответствующее
                    отсутствию текста, введенного пользователем */
                    this.changed = false;
                }
        };
}
```

Теперь применим созданную нами функцию для создания трех текстовых полей стандартной e-mail-формы с подсказками. Результат представлен на рис. 14.8.

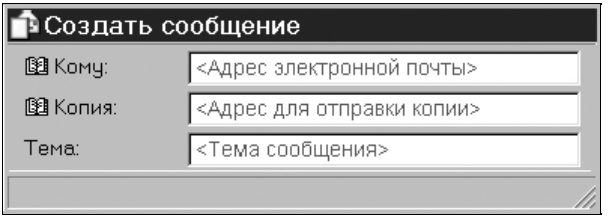


Рис 14.8. Текстовые поля с подсказками, созданные в результате применения функции `addInputTextField()`

```
//----------------------------ПРИМЕНЕНИЕ-----------------------
var hcol = 0x999999; // Цвет текста подсказки
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Arial";
myFormat.size = 14;
myFormat.color = 0x000000; // Цвет пользовательского текста
addInputTextField("Адрес электронной почты", 127, 33, 255, 20, myFormat,↳
hcol);
addInputTextField("Адрес для отправки копии", 127, 60, 255,20, myFormat,↳
hcol);
addInputTextField("Тема сообщения", 127, 87, 255, 20, myFormat, hcol);
```

Замена содержимого текстового блока

Класс `TextField` располагает двумя методами, позволяющими осуществлять замену содержимого текстового блока программным образом. Эти методы позволяют заменять выделенные символы или просто заданный диапазон символов.

Метод *replaceSel()*

Метод `replaceSel()` используется для замены выделенного диапазона символов указанным текстом. Данный метод использует следующий синтаксис:

текстовый блок`.replaceSel(`*текст*`)`

Здесь *текстовый блок* — это экземпляр текстового блока, в который вставляется текст.

Параметр *текст* — это строковый литерал или выражение, возвращающее строку, которая будет вставлена на место выделенного диапазона.

Вставка текста выполняется, начиная с первого символа диапазона выделения. К вставленному тексту применяется форматирование, используемое в выделенном диапазоне текста.

В листинге 14.10 приведен сценарий, позволяющий изменять регистр (прописные/строчные) выделенного диапазона символов текста в результате нажатия соответствующих кнопок с именами `upperCaseBut` и `lowerCaseBut`. Текст выводится в текстовый блок с именем `txt`.

Единственной трудностью в решении поставленной задачи является тот факт, что при нажатии на кнопку текстовое поле с выделенным текстом утрачивает фокус и, соответственно, текстовое выделение пропадает. В связи с этим необходимо непосредственно в процессе выделения курсором мыши сохранять индексы начала и конца выделения в переменных с тем, чтобы эту информацию можно было использовать при нажатии кнопки. Таким образом, пользователь вначале выполняет выделение текста, при этом границы выделения сохраняются в переменных `startTxt` и `endTxt` (здесь используется тот же механизм, что и в листинге 14.8). Затем пользователь щелкает по соответствующей кнопке для преобразования регистра символов, при этом выделение пропадает. В нашу задачу входит передача фокуса текстовому блоку и восстановление там выделения, выполненного пользователем на основании значений переменных `startTxt` и `endTxt`. Вслед за этим нужно применить метод `replaceSel()` для того, чтобы заменить выделенный текст теми же символами, но в требуемом регистре.

Листинг 14.10. Изменение регистра выделенного диапазона текста с помощью кнопок

```
var startTxt = 0; //Начальная позиция выделения
var endTxt = 0; // Конечная позиция выделения
// Создаем объект-приемник для обработки событий мыши
var myListener:Object = new Object();
// Регистрируем myListener на получение событий от объекта Mouse
Mouse.addListener(myListener);
// При перемещении курсора мыши с нажатой левой кнопкой...
myListener.onMouseDown = function() {
        myListener.onMouseMove = function() {
```

```
                // При выделении с помощью мыши ...
                if (Selection.getBeginIndex() < Selection.getEndIndex()) {
                // Сохраняем индексы начала и конца диапазона выделения
                        startTxt = Selection.getBeginIndex();
                        endTxt = Selection.getEndIndex();
                        updateAfterEvent();
                }
        };
};
// При отпускании кнопки мыши...
myListener.onMouseUp = function() {
        /* Прекращаем сохранение индексов диапазона выделения при
        перемещении курсора мыши */
        delete myListener.onMouseMove;
};
// Перевод выделенных символов в верхний регистр
upperCaseBut.onRelease = function() {
        /* Передаем фокус текстовому блоку txt (при этом все его
        содержимое выделяется) */
        Selection.setFocus(txt);
        /* Устанавливаем выделение в соответствии со значениями индексов
        пользовательского выделения, сохраненными в переменных startTxt
        и endTxt.*/
        Selection.setSelection(startTxt, endTxt);
        /* Заменяем выделенный фрагмент текста теми же символами, но в
        верхнем регистре */
        txt.replaceSel(txt.text.substring(startTxt,endTxt).toUpperCase());
        // Обнуляем значения индексов выделения (так как выделение
        // пропадает)
        startTxt = endTxt = 0;
};
// Перевод выделенных символов в нижний регистр выполняется аналогично
lowerCaseBut.onRelease = function() {
        Selection.setFocus(txt);
        Selection.setSelection(startTxt, endTxt);
        txt.replaceSel(txt.text.substring(startTxt,
endTxt).toLowerCase())
        startTxt = endTxt = 0;
};
```

Метод *replaceText()*

Метод `replaceText()` позволяет осуществить замену заданного диапазона текста текстом, переданным в качестве параметра. Этот метод имеет следующий синтаксис:

текстовый блок`.replaceText(`*начальный индекс*`, `*конечный индекс*`, `*текст*`)`

Здесь *текстовый блок* — это экземпляр текстового блока, в который осуществляется вставка текста.

Параметры *начальный индекс* и *конечный индекс* задают соответственно индекс первого и последнего символа удаляемого диапазона текста, на место которого будет вставлен новый текст. Индексация символов в текстовом блоке начинается с нуля.

Параметр *текст* — это строковый литерал или выражение, возвращающее строку, которая будет вставлена в текстовый блок вместо заданного диапазона исходного текста.

В результате вызова метода `replaceText()` диапазон текста, заданный индексами *начальный индекс* и *конечный индекс*, удаляется, а на его место вставляется текст, заданный параметром *текст*. Если значение параметра *начальный индекс* равно значению параметра *конечный индекс*, исходный текст не удаляется, а текст, заданный параметром *текст*, вставляется перед символом с индексом, равным значению параметра *начальный индекс*.

Следующий пример демонстрирует принцип использования метода `replaceText()`, превращая фразу "Все, что хорошо начинается, кончается плохо" во фразу "Все, что начинается плохо, кончается еще хуже" путем последовательного применения данного метода.

```
this.createTextField("txt", 1, 0, 0, 1, 1);
txt.autoSize = true;
txt.text = "Все, что хорошо начинается, кончается плохо"
txt.replaceText(9, 26, "начинается плохо");
trace(txt.text);// Все, что начинается плохо, кончается плохо
txt.replaceText(27, txt.length, "кончается еще хуже");
trace(txt.text); // Все, что начинается плохо, кончается еще хуже
trace(txt.length);
```

В качестве упражнения читатель может самостоятельно модифицировать пример, представленный в листинге 14.10, внеся соответствующие изменения с тем, чтобы заменить метод `replaceSel()` методом `replaceText()`.

Удаление текстового блока

Для удаления динамически созданного текстового блока используется метод `removeTextField()` класса `TextField`. Данный метод по отношению к текстовым блокам действует аналогично методу `removeMovieClip()`, используемому для удаления динамически созданных клипов. Следует иметь в виду, что метод `removeTextField()` позволяет удалять только текстовые блоки, созданные с помощью метода `cre-`

ateTextField(). Метод removeTextField() не принимает параметров и имеет следующий синтаксис:

текстовый блок.removeTextField()

Здесь *текстовый блок* — это удаляемый экземпляр текстового блока.

В листинге 14.11 приведен сценарий, который выводит координаты курсора мыши в динамически созданный текстовый блок при нажатии левой кнопки мыши. Смещение курсора мыши или отпускание ее левой кнопки приводит к удалению данного текстового блока.

Листинг 14.11. Вывод координат мыши в текстовое поле

```
var myListener = new Object();
Mouse.addListener(myListener);
myListener.onMouseDown = function() {
        _root.createTextField("mouseTxt", _root.getNextHighestDepth(),↳
        _xmouse + 10, _ymouse + 10, 1, 1);
        mouseTxt.autoSize = true;
        mouseTxt.border = true;
        mouseTxt.text = "x:" + _xmouse + " y:" + _ymouse;
        myListener.onMouseMove = function() {
                mouseTxt.removeTextField();
        };
};
myListener.onMouseUp = function() {
        mouseTxt.removeTextField();
        delete this.onMouseMove;
};
```

В качестве упражнения читатель может самостоятельно модифицировать пример, представленный в листинге 14.1 так, чтобы всплывающие подсказки появлялись и удалялись не за счет изменения значения свойства _visible, а в результате создания и удаления текстовых блоков.

Форматирование текста с помощью HTML

В проигрыватель Flash Player включена поддержка некоторых базовых тегов HTML. Это позволяет осуществлять форматирование содержимого текстового блока, используя теги разметки подобно тому, как это делается при работе с HTML. При необходимости использования различного форматирования в пределах одного текстового блока применение возможностей, предоставляемых классом TextFormat, может оказаться достаточно неудобным и трудоемким, поскольку требует указания диапазона символов, к которому применяется форматирование. Использование HTML-

форматирования позволяет достаточно просто и эффективно применять различные параметры форматирования к различным фрагментам текста внутри одного текстового блока. Кроме того, с помощью HTML можно реализовать некоторые возможности, недоступные для форматирования обычными средствами.

Активизация поддержки HTML и вывод текста с применением HTML-форматирования

Для того чтобы к текстовому блоку можно было применить HTML-форматирование, необходимо для данного текстового блока активизировать режим поддержки HTML. Этот может быть сделано как для текстового блока, созданного в рабочей среде, так и для текстового блока, созданного с помощью метода `createTextField()`.

Для того чтобы активизировать поддержку HTML для текстового блока, созданного в рабочей среде, необходимо воспользоваться кнопкой **Render text as HTML** , расположенной в панели **Properties**.

Примечание

При активизации поддержки HTML для текстового блока, созданного в рабочей среде, появляется возможность задания различных параметров форматирования для его содержимого, помещенного в текстовый блок непосредственно в рабочей среде. Также становится доступной возможность назначения гиперссылки.

Для того чтобы активизировать поддержку HTML для текстового блока, созданного с помощью метода `createTextField()`, необходимо для данного экземпляра текстового блока установить свойство `html` в значение `true`. Свойство `html` может быть определено для любого объекта `TextField`. Данное свойство принимает значение булева типа и определяет, активизирован ли режим поддержки HTML для соответствующего текстового блока.

Для вывода текста с применением HTML-форматирования в текстовый блок можно использовать переменную или свойство `htmlText`.

При работе с переменной достаточно присвоить ей в качестве значения строку, содержащую текст, и теги разметки точно так же, как и при выводе обычного текста. Если режим поддержки HTML активизирован, текст будет выводиться с применением соответствующего форматирования.

Свойство `htmlText` представляет собой аналог свойства `text`, используемый для вывода текста, включающего теги разметки, которые определяют его форматирование. Данное свойство содержит текст, находящийся в текстовом блоке, в его HTML-представлении. Как и свойство `text`, свойство `htmlText` может использоваться для чтения и для записи. Если режим поддержки отключен, свойство `htmlText` действует аналогично свойству `text`.

В следующем примере создается динамический текстовый блок и в него выводится строка текста, содержащего HTML-форматирование, выполняющее подчеркивание.

```
this.createTextField("txt", 1, 100, 200, 1, 1);
txt.autoSize = true;
```

```
txt.border = true;
txt.html = true;
txt.htmlText = "<u>Непредусмотрительность - вот точная наука</u>";
```

При работе с HTML-текстом имеется возможность удаления всех дополнительных промежутков между символами и строками (пробелы, переводы строк и т. д.). Для этих целей используется свойство `condenseWhite`, определенное в классе `TextField`.

Свойство `condenseWhite` принимает булево значение `true` или `false` и оказывает воздействие на содержимое текстового блока только при активизации поддержки HTML-форматирования. Если свойство `condenseWhite` принимает значение `true`, из текстового блока удаляются все последовательно расположенные пробелы, за исключением начальных. Так, если исходная строка текста содержит три последовательных пробела, то два их них будут удалены. Из текста также удаляются все переносы строк (выполненные с помощью Escape-последовательностей `"\n"` или `"\r"`). Для выполнения переноса необходимо использовать тег `<BR>` или `<P>`. По умолчанию свойство `condenseWhite` принимает значение `false`, при котором автоматического удаления промежутков в тексте не происходит.

Теги HTML, используемые для форматирования текста

В Flash используется стандарт XHTML, в связи с чем названия тегов и их атрибутов обычно указываются с применением только строчных букв. Это требование не является обязательным. Применение традиционно используемого в HTML верхнего регистра не приводит к некорректной работе, однако здесь и в дальнейшем мы будем придерживаться правил, используемых в XHTML. Значения атрибутов тегов должны быть заключены в кавычки вне зависимости от того, являются они строками или числами. Поскольку строковый литерал, присваиваемый свойству `htmlText`, заключается в кавычки, для использования кавычек внутри строки применяются либо одиночные кавычки, либо Escape-последовательность (`\"` или `\'`). Список тегов, поддерживаемых проигрывателем Flash Player, приведен в табл. 14.1.

Таблица 14.1. *Теги HTML-форматирования, поддерживаемые проигрывателем Flash Player*

Теги HTML	Описание
`<a>`	Используется для создания текстовой гиперссылки
`<b>`	Устанавливает полужирное начертание шрифта
` `	Выполняет перенос строки
`<font>`	Задает гарнитуру, размер и цвет шрифта
`<i>`	Устанавливает курсивное начертание шрифта
`<img>`	Внедряет в текстовый блок растровое изображение, SWF-документ или символ из библиотеки фильма

Таблица 14.1 *(окончание)*

Теги HTML	Описание
	Применяется для создания маркированного списка
<p>	Применяется для создания нового абзаца
	Позволяет использовать класс CSS для форматирования соответствующего текстового фрагмента
<textformat>	Позволяет выполнять форматирование текста, используя набор параметров, соответствующих свойствам класса TextFormat, которые указываются как атрибуты данного тега
<u>	Отображает текст с подчеркиванием

Тег *<font>*

Тег <font> задает гарнитуру, размер и цвет шрифта. Поддерживаются следующие атрибуты данного тега:

- face — задает шрифт, используемый для вывода текста. Если используемый шрифт отсутствует в системе пользователя и не встроен в документ, вместо него будет применяться соответствующий машинно-независимый шрифт. При использовании встроенного шрифта значением данного атрибута является идентификатор связи соответствующего шрифтового символа;
- size — задает размер шрифта в пунктах. Допускается применение относительных размеров. Относительному значению предшествует знак + или –. Относительные значения вычисляются на основании размера шрифта, используемого по умолчанию. Так, если по умолчанию размер шрифта составляет 12 пунктов, то использование значения +6 установит кегль 18;
- color — задает цвет шрифта. Значением данного параметра является строка, содержащая шестнадцатеричное представление цвета, которое начинается с префикса #. Строковые константы, содержащие название цвета типа "marine blue" или "cyan", не поддерживаются.

В следующем примере в текстовый блок выводится текст синего цвета с кеглем 18 пунктов и с применением гарнитуры Pragmatica.

```
txt.htmlText = "<font face = 'Pragmatica' size = '18' color = '#0000FF'>
Непредусмотрительность - вот точная наука</font>";
```

Тег *<b>*

Тег <b> устанавливает полужирное начертание шрифта.

```
txt.htmlText = "<b>Ставьте задачи, по которым решение есть только у вас
</b>";
```

Тег <i>

Тег <i> устанавливает курсивное начертание шрифта.

```
txt.htmlText = "<i>Новые системы плодят новые проблемы</i>";
```

Тег <u>

Тег <u> отображает шрифт с подчеркиванием.

```
txt.htmlText = "<u>Если вы попали впросак, напускайте туману</u>";
```

Тег <a>

Тег <a> используется для создания текстовой гиперссылки. В Flash поддерживаются следующие атрибуты данного тега:

- href — задает абсолютный или относительный URL, из которого осуществляется загрузка требуемой страницы. Относительный путь указывается относительно местоположения HTML-страницы, содержащей SWF-документ с гиперссылкой;
- target — указывает целевое окно или фрейм, в который будет загружена страница. Данный параметр может принимать одно из стандартных значений _self, _blank, _parent, _top или пользовательское имя окна, в которое будет осуществлена загрузка.

Следующий код выводит текстовую гиперссылку с применением подчеркивания:

```
txt.htmlText = "<a href = 'http://www.macromedia.com/support' target =
'_blank'> <u>Служба технической поддержки компании macromedia</u></a>";
```

Аналогично можно создать ссылку, открывающую почтовую программу для отправки сообщения по заданному адресу:

```
txt.htmlText = "<a href = 'mailto:book_flash@mail.ru'> <u>Написать письмо
автору</u></a>";
```

Примечание

Используя встроенную поддержку CSS, для тега <a> можно определить стили a:link, a:hover и a:active, позволяющие реализовать стандартную функциональность HTML-гиперссылки. Эти возможности будут рассмотрены в настоящей главе.

Помимо стандартных протоколов, таких как HTTP, HTTPS или FTP, атрибут href позволяет использовать особый (специфичный для Flash) протокол asfunction, позволяющий вызвать функцию ActionScript при нажатии на текст гиперссылки. При этом вызываемой функции может быть передан один параметр. Протокол asfunction имеет следующий синтаксис:

```
asfunction:функция,параметр
```

Здесь функция — это имя вызываемой функции. Следует обратить внимание, что здесь указывается только идентификатор функции, оператор вызова функции () не используется.

Параметр, передаваемый функции, представляет собой строку. Следует иметь в виду, что при вызове функции в качестве строки передается все, что указывается после запятой. Так, например, пробел, помещенный после запятой, будет интерпретироваться как первый символ строки, передаваемой в качестве параметра, что может привести к неожиданным результатам или некорректной работе кода.

Следующий код перемещает воспроизводящую головку в кадр с меткой `start` и начинает воспроизведение при нажатии на текст гиперссылки, которая выводится с применением HTML-форматирования.

```
stop();
this.createTextField("txt", 1, 100, 200, 100, 400);
txt.autoSize = true;
txt.border = true;
txt.html = true;
txt.htmlText = "<a href = 'asfunction:gotoAndPlay,start'><u>Начать
воспроизведение </u></a>";
```

При необходимости передачи вызываемой функции более одного параметра можно воспользоваться каким-либо разделителем и затем, при вызове функции непосредственно, в ее теле разделить параметры, например, с помощью метода `split()`.

Тег *
*

Тег `<br>` выполняет перенос строки. Оказываемое им действие равносильно применению Escape-последовательности `\n`. Различие состоит в том, что использование свойства `condenseWhite` не приводит к удалению переносов строк, выполненных с помощью тега `<br>`, в то время как переносы, выполненные с помощью Escape-последовательности `\n`, удаляются.

Следующий код выводит текст в три строки:

```
txt.htmlText "Никогда не рисуйте то, что можно скопировать <br>Никогда
не копируйте то, что можно обвести</br> <br>Никогда не обводите то, что
можно вырезать и наклеить</br>"
```

Применение закрывающего тега `</br>` не является обязательным, однако использование закрывающего тега является желательным в целях лучшего соответствия правилам синтаксиса XHTML.

Тег *<p>*

Тег `<p>` применяется для создания нового абзаца. Результат действия данного тега несколько отличается от его применения в HTML. Отличия выражаются в следующем:

- для создания нового абзаца необходимо наличие закрывающего тега `</p>`;
- тег `<p>` в Flash вызывает перевод текста только на одну строку, в то время как в HTML применение данного тега переводит текст на две строки;

Следующий код выводит текст в два абзаца:

```
txt.htmlText = "<p>То, что вы храните достаточно долго, можно выбросить
</p>Как только вы что-то выбросите, оно вам понадобится"
```

В Flash поддерживаются два атрибута тега <p>:

- align — определяет выравнивание текста в пределах абзаца. Допустимыми значениями являются следующие строки:
  - "center"— выравнивание по центру. Данное значение используется по умолчанию;
  - "right" — выравнивание по правому краю;
  - "left" — выравнивание по левому краю;
- class — данный атрибут применяется для связывания с классом CSS, используемым для форматирования содержимого абзаца. Эта возможность будет рассмотрена далее в настоящей главе.

Следует иметь в виду, что при изменении значения свойства htmlText с применением составного оператора присваивания (+=) конкатенируемые строки автоматически размечаются тегом <p>, что приводит к переносу строки и созданию нового абзаца. Так, следующий код выведет текст в два абзаца:

```
txt.htmlText = "Привет, ";
txt.htmlText+= "Flash";
```

Результат будет выглядеть так:

```
Привет,
Flash
```

Эквивалентом приведенного кода является следующий:

```
txt.htmlText = "<p>Привет,</p><p> Flash </p> ";
```

Тег *<li>*

Тег <li> применяется для создания маркированного списка. Текст, заключенный в открывающий и закрывающий теги <li> и , выводится с новой строки, перед первым символом которой помещается стандартный маркер списка.

Следующий пример иллюстрирует использование маркированного списка.

```
this.createTextField("txt", 1, 100, 200, 300, 65);
txt.multiline = true;
txt.border = true;
txt.html = true;
txt.htmlText = "<p align = 'center'> <b> Краткий определитель
современных наук</b></p><li>Если оно зеленое или дергается – это
биология.</li><li>Если дурно пахнет - химия.</li><li>Если не работает -
физика.</li>"
```

Приведенный код выводит следующий результат:

Краткий определитель современных наук

- Если оно зеленое или дергается — это биология.
- Если дурно пахнет — химия.
- Если не работает — физика.

Тег *<textformat>*

Тег `<textformat>` позволяет выполнять форматирование текста, используя набор параметров, соответствующих свойствам класса `TextFormat`, которые указываются как атрибуты данного тега. Фактически данный тег позволяет применять к HTML-тексту форматирование, для применения которого к обычному тексту используются свойства класса `TextFormat`. Тег `<textformat>` имеет следующие атрибуты:

- `blockindent` — задает отступ всех строк текста от левой границы текстового блока в пунктах. Данный атрибут соответствует свойству `blockindent` класса `TextFormat`;
- `indent` — задает отступ красной строки (то есть первой строки абзаца) в пунктах. Данный атрибут соответствует свойству `indent` класса `TextFormat`;
- `leading` — задает интерлиньяж, то есть межстрочное расстояние. Данный атрибут соответствует свойству `leading` класса `TextFormat`;
- `leftmargin` — задает ширину левого поля в пунктах. Данный атрибут соответствует свойству `leftMargin` класса `TextFormat`;
- `rightmargin` — задает ширину правого поля в пунктах. Данный атрибут соответствует свойству `rightMargin` класса `TextFormat`;
- `tabstops` — позволяет задать положение позиций табуляции, используемых в текстовом блоке. Данный атрибут соответствует свойству `tabStops` класса `TextFormat`, его значение задается как литерал массива с использованием оператора квадратных скобок `[]`.

Следующий код иллюстрирует применение атрибута `tabstops` для создания простой таблицы.

```
this.createTextField("txt", 1, 100, 200, 300, 65);
txt.multiline = true;
txt.autoSize = true;
txt.border = true;
txt.html = true;
// Заголовок
var header = "Исполнитель\tКомпозиция\tДлительность";
// Строки
var row1 = "Oscar Peterson\tMack The Knife\t7,25";
```

```
var row2 = "Andy Goodrich\tYou Must Believe in Spring\t8,30";
var row3 = "The Charlie Byrd Quintet\tBesame Mucho\t4,16";
// Вывод текста
txt.htmlText = "<textformat tabStops = '150, 300'>";
txt.htmlText += "<p align = 'center'><b>"+header+"</b></p>";
txt.htmlText += row1;
txt.htmlText += row2;
txt.htmlText += row3;
txt.htmlText += "<textformat>";
```

Результат действия этого кода приведен далее.

Исполнитель	Композиция	Длительность
Oscar Peterson	Mack The Knife	7,25
Andy Goodrich	You Must Believe in Spring	8,30
The Charlie Byrd Quintet	Besame Mucho	4,16

Тег **

Тег `<span>` позволяет использовать класс CSS для форматирования соответствующего текстового фрагмента. Этот тег имеет единственный атрибут `class`, позволяющий установить связь с классом CSS, задающим параметры форматирования. Подробно применение данного тега будет описано далее в этой главе.

Тег *<img>*

Тег `<img>` позволяет загружать в текстовый блок экземпляры символов из библиотеки фильма, растровые изображения в формате JPEG, а также внешние SWF-документы. Загружаемые изображения и фильмы встраиваются непосредственно в текст, при этом автоматически выполняется обтекание текстом. Для того чтобы загрузка и позиционирование изображений в текстовом блоке выполнялись корректно, необходимо установить свойство `wordWrap` в значение `true` (или включить режим `multiline` для текстового блока, созданного в рабочей среде). Использование тега `<img>` приводит к созданию нового клипа, который служит контейнером для загружаемого изображения или фильма. В свою очередь сам клип-контейнер является дочерним объектом экземпляра текстового блока, осуществляющего загрузку. Автоматически создаваемый клип-контейнер помещается на нижний уровень стека объекта, содержащего текстовый блок, в который внедряется изображение. Если в текстовый блок внедряется несколько изображений, то изображение, которое находиться в тексте ниже, располагается в клипе, который помещается на самый нижний уровень стека. Тег `<img>` не требует применения закрывающего тега. Внедряемое в текстовый блок изображение (или ролик) помещается в строку, следующую за тегом `<img>`. Исключением является ситуация, когда текст начинается с тега `<img>`,

в этом случае изображение помещается на первую строку. Для того чтобы внедренные изображения не перекрывали друг друга, необходимо разместить между ними достаточное количество текста. В противном случае изображение, расположенное в тексте ниже, будет перекрывать изображение, расположенное выше.

Тег `<img>` имеет следующие атрибуты:

- `src` — позволяет задать абсолютный или относительный путь к загружаемому документу. Данный атрибут является обязательным. Для загрузки растровых изображений или внешних SWF-документов значение данного атрибута представляет собой путь к файлу с расширением jpg (jpeg) или swf соответственно. Относительные ссылки разрешаются относительно местоположения HTML-страницы, содержащей SWF-документ, в котором выполняется загрузка. Следующий код встраивает изображение в текстовый блок с именем `txt`.

  ```
  txt.htmlText = "<img src = 'picture.jpg'> Продолжительность брака
  обратно пропорциональна расходам на свадьбу"
  ```

  Необходимо помнить, что Flash Player не поддерживает динамическую загрузку изображений, использующих режим **Progressive**. Изображение, клип или внешний фильм отображаются только после полного окончания своей загрузки.

  Для того чтобы внедрить в текстовый блок экземпляр символа, расположенного в библиотеке документа, необходимо снабдить этот символ идентификатором связи (Linkage Identifier). Это делается так же, как и при использовании метода `attachMovie()`. Атрибуту `src` при этом необходимо присвоить строку, содержащую идентификатор связи символа, который должен быть встроен в текстовый блок;

- `id` — данный атрибут позволяет задать идентификатор или имя экземпляра клипа-контейнера, создаваемого проигрывателем, в который будет загружено изображение или ролик. Применение данного атрибута оказывается полезным, если загруженное изображение должно контролироваться с помощью ActionScript. Следующий код встраивает в текстовый блок внешний SWF-документ и присваивает ему идентификатор `mov`.

  ```
  txt.htmlText = "<img src = 'movie.swf' id = 'mov'> Того, что вам
  нравится, всегда нет в продаже"
  ```

  Приведенная запись означает, что загруженный в текстовый блок `txt` объект помещается в автоматически созданный дочерний клип-контейнер с именем `mov`. Имя клипа сгенерировано на основании значения атрибута `id`.

  Необходимо помнить, что автоматически создаваемый клип-контейнер является дочерним объектом текстового блока, поэтому обращение к нему осуществляется как к свойству экземпляра текстового блока. Так, для того чтобы задать внедренному в текстовый блок `txt` объекту с идентификатором `mov` 50%-ную прозрачность, нужно использовать следующий код:

  ```
  txt.mov._alpha = 50;
  ```

- атрибуты `width` и `height` — задают ширину и высоту внедряемого изображения, фильма или клипа. Данные значения определяют, какую область внутри текстового блока будут занимать изображения после внедрения. На основании значений этих атрибутов в текстовом блоке резервируется место для внедряемого объ-

екта и выполняется обтекание зарезервированных участков текстом еще до окончания их загрузки. После того как элемент полностью загрузился, он масштабируется в соответствии со значениями атрибутов `width` и `height` и отображается в предназначенной для него области.

Если данные атрибуты не указываются, то место для внедряемых изображений не резервируется, и на протяжении их загрузки текст заполняет все пространство в пределах текстового блока. После окончания загрузки изображения или ролика он внедряется в текстовый блок, используя свои исходные размеры, и становится видимым. При этом автоматически выполняется обтекание текстом, в результате чего исходный порядок расположения строк может существенно измениться.

Примечание

Следует иметь в виду, что при внедрении в текстовый блок экземпляра символа из библиотеки документа его точка регистрации должна находиться в левом верхнем углу.

Несмотря на то, что атрибуты `width` и `height` не являются обязательными, тем не менее, рекомендуется применять их всегда с тем, чтобы сделать процесс загрузки более естественным и обеспечить корректное обтекание внедряемых объектов текстом;

- `align` — данный атрибут определяет выравнивание внедренного изображения относительно полей текстового блока. Допустимыми значениями данного атрибута являются `"left"` и `"right"`. По умолчанию используется значение `"left"`. Для того чтобы без перекрытия расположить два изображения на одной и той же строке, нужно задать им разные значения атрибута `align`. Тогда при условии, что их суммарная ширина не превышает расстояния между полями текстового блока, изображения могут располагаться на одном уровне по горизонтали;
- атрибуты `hspace` и `vspace` определяют расстояние от границ внедряемого изображения до текста по горизонтали и по вертикали соответственно. По умолчанию значения данных атрибутов составляют 8 пикселов. Атрибуты `hspace` и `vspace` могут принимать как положительные, так и отрицательные значения. Применение отрицательных значений приводит к тому, что изображение перекрывает текст.

Приведенный ниже сценарий демонстрирует возможности применения тега `<img>` и его атрибутов. В данном сценарии выполняется загрузка в текстовый блок внешнего SWF-документа и клипа с идентификатором связи `runner`, расположенного в библиотеке. Результат действия этого сценария представлен на рис. 14.9.

```
this.createTextField("txt", 1, 0, 0, 320, 400);
txt.border = true;
txt.borderColor = 0x333333;
txt.autoSize = false;
txt.wordWrap = true;
txt.html = true;
txt.htmlText = "<font face = 'Arial Cyr' color = '#DDDDDD'><b>Символы
Flash\n</b>Символ типа муви-клип может<img src = 'swf/engine.swf' id =
```

```
'eng' width = '110' height = '140' hspace = '10' vspace = '10'>содержать
на своей монтажной линейке любые графические объекты, анимацию,
экземпляры любых других символов (включая муви-клип), звук и сценарии
ActionScript. Замечательной особенностью муви-клипа является независимость
его монтажной линейки от монтажной линейки, на которую помещен его
экземпляр. Вследствие этого вся функциональность муви-клипа будет
выполняться до тех пор, пока его экземпляр присутствует на сцене.\n<img
src ='runner' id = 'runner' width = 120 height = 105 align = 'right'
hspace = '10' vspace = '10'> В символ типа Graphic могут быть помещены
экземпляры любых других символов (включая графический), но кадры его
монтажной линейки не могут содержать звук и сценарии ActionScript.
Монтажная линейка графического символа зависит от родительской монтажной
линейки (т. е. линейки, в кадр которой помещен его экземпляр).</font>";
```



Рис. 14.9. Загрузка в текстовый блок внешнего SWF-документа и клипа из библиотеки

Для того чтобы снабдить внедренное изображение или клип функциональностью гиперссылки, можно вложить тег `<img>` в тег `<a>`. Так, следующий код внедряет в текстовый блок `txt` растровое изображение и превращает его в гиперссылку, осуществляющую вызов почтовой программы и передачу ей адреса электронной почты:

```
txt.htmlText = "<a href = 'mailto:book_flash@mail.ru'><img src =↳
'myPicture.jpg' width = '90' height = '90'</a>";
```

Применение такого подхода приводит к тому, что внедренный объект перестает реагировать на события мыши и клавиатуры. Поэтому для того чтобы снабдить внедренный элемент интерактивностью, необходимо отказаться от заключения его в тег `<a>` и реализовать соответствующие функции с помощью ActionScript.

При необходимости программного управления внедренными объектами можно столкнуться с проблемой, обусловленной ходом выполнения загрузки. Как уже отмечалось, внедренный в текстовый блок ролик или изображение отображаются только после полного окончания своей загрузки. При внедрении символа из библиотеки документа с применением идентификатора связи загрузка осуществляется так же, как и при динамическом создании экземпляра с помощью метода `attachMovie()`. Если в параметрах связи (окно **Linkage Properties**) для загружаемого экземпляра установлен флажок **Export in first frame**, то соответствующий символ будет загружен перед загрузкой первого кадра основной монтажной линейки фильма. В этом случае при вхождении воспроизводящей головки в кадр со сценарием, в котором выполняется внедрение экземпляра в текстовый блок, экземпляр сразу отобразится в тексте. Если флажок снят, то для того чтобы экземпляр символа мог быть загружен в текстовый блок, на монтажной линейке должен присутствовать другой экземпляр этого символа, созданный вручную.

Загрузка внешнего файла (фильма или растрового изображения) выполняется уже после загрузки основного документа. Поэтому внедренные в текстовый блок экземпляры символа отображаются раньше, чем внешние файлы.

Важно отметить, что создание клипа-контейнера, в который помещается внедренный в текстовый блок экземпляр или изображение, происходит после выполнения сценария кадра основной монтажной линейки фильма. Это означает, что в одном и том же кадре нельзя разместить код, реализующий внедрение объекта, и непосредственное обращение к клипу-контейнеру, содержащему этот объект, поскольку создание контейнера происходит после выполнения сценария кадра.

Одним из вариантов преодоления этой проблемы является размещение в кадре, в котором выполняется внедрение изображения, сценария, осуществляющего периодическую проверку, произошло ли создание клипа-контейнера. Для этого можно использовать событие `onEnterFrame`. Если клип-контейнер создан, можно выполнить с ним необходимые действия. Подобный алгоритм реализуется в листинге 14.12. Здесь для двух клипов-контейнеров с именами `eng` и `runner`, в которых находятся внедренные в текстовый блок `txt` внешний фильм и экземпляр из библиотеки, определяются методы обработчиков события `onPress`. Последовательное нажатие на загруженный файл с идентификатором `eng` изменяет степень его прозрачности, а последовательное нажатие на экземпляр `runner` останавливает и снова запускает его анимацию. Проблема заключается в том, что на момент выполнения сценария кадра клипы-контейнеры еще не созданы, поэтому код должен определять методы обработчиков до тех пор, пока соответствующие клипы не будут созданы. Механизм работы данного сценария заключается в том, что он периодически (с частотой, равной скорости воспроизведения фильма) пытается определить (создать) метод обработчика событий для клипов-контейнеров. До тех пор, пока эти клипы не будут созданы, определить их метод нельзя (свойство `onPress`, хранящее ссылку на этот метод, принимает значение `undefined`), поэтому попытки возобновляются. Как только клип-контейнер создан, определяется метод обработчика события `onPress`. Когда оба метода оказываются определены, происходит удаление объекта, выполняющего периодическую проверку, поскольку он уже исполнил свою роль. Код, реализующий внедрение фильма и клипа в текстовый блок `txt`, был приведен выше. В настоящем сценарии объект `MovieClip` используется в качестве источника события `onEnter-`

Frame, которое транслируется объекту-приемнику objListener. Эта возможность была описана в *разд. "Применение класса AsBroadcaster" главы 11.*

Листинг 14.12. Обработка событий объектов, внедренных в текстовый блок

```
AsBroadcaster.initialize(MovieClip);
var objListener:Object = new Object();
MovieClip.addListener(objListener);
this.createEmptyMovieClip("broadcaster", this.getNextHighestDepth());
broadcaster.onEnterFrame = function() {
       MovieClip.broadcastMessage("onEnterFrame");
};
objListener.onEnterFrame = function() {
       /* Если методы-обработчики для обоих внедренных объектов
       определены, удаляем объект-приемник и клип, транслирующий событие
       onEnterFrame, так как они больше не нужны */
       if (txt.eng.onPress != undefined && txt.runner.onPress !=↳
       undefined) {
              delete objListener;
              broadcaster.removeMovieClip();
       }
       /* Определяем метод-обработчика (если он еще не определен) для
       клипа-контейнера, в который помещается внешний файл */
       if (txt.eng.onPress == undefined) {
              txt.eng.onPress = function() {
                     // Стандартная кнопка-флаг
                     if(!j){
                     this._alpha = 50;
                     j=true;
                     }
                     else{
                     this._alpha = 100;
                     j=false;
                     }
              };
       }
       /* Определяем метод-обработчика (если он еще не определен) для
       клипа-контейнера, в который помещается экземпляр из библиотеки */
       if (txt.runner.onPress == undefined) {
              txt.runner.onPress = function() {
```

```
                // Стандартная кнопка-флаг
                        if(!i){
                        this.stop();
                        i=true;
                        }
                        else{
                        this.play()
                        i=false;
                        }
                };
        }
};
```

Поддержка и использование CSS. Класс *StyleSheet*

CSS (Cascading Style Sheets) — каскадные таблицы стилей применяются для определения правил форматирования содержимого в HTML- или XML-документах. Применение таблиц стилей позволяет компактно и изолированно хранить всю информацию, описывающую параметры форматирования различных элементов. Это дает возможность обеспечить единообразное стилевое оформление для любого количества документов и существенно облегчает процедуру изменения форматирования элементов.

Проигрыватель Flash Player 7 поддерживает лишь малую часть возможностей, предоставляемых спецификацией CSS1, однако даже эти весьма ограниченные возможности позволяют существенно оптимизировать процедуру форматирования и, что особенно важно, значительно упростить обновление и изменение параметров форматирования содержимого текстовых блоков.

В настоящем разделе будут рассмотрены возможности применения CSS в Flash. Как вскоре будет видно, форматирование текста с помощью CSS в Flash — это простая задача, для решения которой достаточно иметь лишь самое общее представление о технологии CSS. Читатели, незнакомые с этим языком, смогут получить всю необходимую информацию по работе с CSS в Flash далее в этой главе, однако для более глубокого изучения вопросов создания и применения каскадных таблиц необходимо обратиться к соответствующей литературе или к многочисленным ресурсам Интернета, посвященным этой теме.

Применение CSS в Flash включает четыре основных этапа.

1. Создание объекта `StyleSheet` для хранения таблицы стилей.
2. Определение правил форматирования, то есть собственно стилей. Для объявления стилей можно использовать ActionScript или выполнять их загрузку из внешнего файла CSS.

3. Добавление стилей в таблицу стилей, представленную объектом `StyleSheet`.
4. Применение таблицы стилей, хранящейся в объекте `StyleSheet`, для форматирования содержимого текстового блока.

Создание таблицы стилей

Таблицей стилей (Style sheet) называется набор правил, которые описывают форматирование содержимого HTML- или XML-документа. В дальнейшем под таблицей стилей мы будем подразумевать набор правил, которые определяют параметры форматирования содержимого динамического или пользовательского текстового блока.

Создание таблицы стилей и выполнение различных манипуляций с ее содержимым осуществляется с помощью встроенного класса `StyleSheet`. Экземпляр класса `StyleSheet` представляет таблицу стилей и хранит информацию о наборе правил форматирования, определяющих различные параметры форматирования текста, такие как, например, шрифт, его размер, цвет, начертание и др. В классе `StyleSheet` определен ряд методов, с помощью которых можно добавлять новые стили в таблицу, получать информацию о хранящихся там стилях и очищать ее, удаляя стили.

Поскольку класс `StyleSheet` находится в пакете `TextField`, создание его экземпляра с помощью конструктора выполняется с применением следующего синтаксиса *(понятие пакета классов будет рассмотрено в главе 25)*:

```
var таблица_стилей:TextField.StyleSheet = new TextField.StyleSheet()
```

Здесь *`таблица_стилей`* — это имя переменной, в которой сохраняется ссылка на создаваемый экземпляр класса `StyleSheet`.

В результате создания экземпляра класса `StyleSheet` создается объект, предназначенный для хранения таблицы стилей, однако изначально этот объект пуст — он не содержит свойств, описывающих правила форматирования. Для того чтобы создать таблицу стилей, необходимо добавить стили в объект `StyleSheet`. Выполнение этой процедуры может быть реализовано одним из трех способов.

- Создание стиля с помощью объекта общего назначения `Object`. Параметры форматирования, выполняемого стилем, задаются как свойства этого объекта. Созданный таким образом объект, представляющий стиль, передается в качестве параметра методу `setStyle()` класса `StyleSheet`, который добавляет указанный стиль в таблицу стилей, хранящуюся в объекте `StyleSheet`.
- Описание стиля в виде строки, содержащей объявление правил форматирования, выполненное с применением стандартного синтаксиса CSS. Сформированная таким образом строка передается методу `parseCSS()` класса `StyleSheet`, который выполняет синтаксический разбор этой строки и, если разбор выполнен успешно, добавляет переданные стили в таблицу стилей, хранящуюся в объекте `StyleSheet`.
- Создание стиля во внешнем CSS-файле (с расширением css). Добавление стилей, объявление которых содержится во внешнем файле, выполняется с помощью метода `load()` класса `StyleSheet`, осуществляющего загрузку внешней таблицы стилей в объект `StyleSheet`.

Как видно из приведенного краткого обзора, определение стиля может выполняться либо с помощью ActionScript, либо с применением стандартного синтаксиса CSS. Перед тем как перейти к подробному описанию возможностей определения стилей, приведем перечень свойств CSS, поддерживаемых проигрывателем Flash Player, и соответствующих им свойств ActionScript. Эта информация представлена в табл. 14.2. Анализ данной таблицы показывает, что названия свойств ActionScript получаются из названий соответствующих свойств CSS в результате удаления дефиса и использования прописных букв в составных именах.

Таблица 14.2. *Поддерживаемые в Flash свойства CSS и соответствующие им свойства ActionScript*

Свойство CSS	Свойство ActionScript	Назначение свойства и принимаемые им значения
`text-align`	`textAlign`	Задает выравнивание текста в абзаце. Допустимыми значениями являются: `left`, `center` и `right`
`font-size`	`fontSize`	Задает размер шрифта в пикселах. Допустимым значением является целое неотрицательное число. Единицы измерения (px или pt) не указываются
`text-decoration`	`textDecoration`	Определяет, применяется ли при выводе текста подчеркивание. Допустимыми значениями являются `none` и `underline`
`margin-left`	`marginLeft`	Задает ширину левого поля. Допустимым значением является целое неотрицательное число. Единицы измерения (px или pt) не указываются
`margin-right`	`marginRight`	Задает ширину правого поля. Допустимым значением является целое неотрицательное число. Единицы измерения (px или pt) не указываются
`font-weight`	`fontWeight`	Определяет, используется ли при выводе текста полужирное начертание. Допустимыми значениями являются `normal` и `bold`
`font-style`	`fontStyle`	Определяет, используется ли при выводе текста курсивное начертание. Допустимыми значениями являются `normal` и `italic`
`text-indent`	`textIndent`	Задает отступ красной строки. Допустимым значением является целое неотрицательное число. Единицы измерения (px или pt) не указываются.

Таблица 14.2 *(окончание)*

Свойство CSS	Свойство ActionScript	Назначение свойства и принимаемые им значения
`font-family`	`fontFamily`	Задает перечень шрифтов, используемых для вывода текста. Значением данного свойства является набор произвольных названий шрифтов, разделенных запятыми. Шрифты указываются в порядке убывания желательности их использования. Если у пользователя отсутствует шрифт, указанный первым, для вывода текста используется шрифт, указанный вторым. Если отсутствует первый и второй, используется третий и т. д. Допускается задание имен типовых шрифтов, которые автоматически преобразуются в соответствующие машинно-независимые шрифты. Поддерживаются следующие типовые имена шрифтов: `mono` преобразуется в `_typewriter`, `sans-serif` преобразуется в `_sans`, и `serif` преобразуется в `_serif`
`color`	`color`	Задает цвет текста. Допустимым значением является шестнадцатеричное число, перед которым указывается префикс `#` (решетка). Цветовые константы (такие как gold, pink или gray) не поддерживаются
`display`	`display`	Определяет, как будет отображаться элемент. Допустимыми значениями являются: `block`, `inline` и `none`. Значение `block` разбивает строку, выделяя элемент в отдельный структурный блок, последующий текст выводится с новой строки. Значение `inline` не приводит к разбиению строки, то есть в одной строке может находиться несколько элементов. Значение `none` отключает отображение элемента

Для связывания таблицы стилей, представленной объектом `StyleSheet`, с экземпляром текстового блока применяется свойство `styleSheet` объекта `TextField`. Если данному свойству текстового блока в качестве значения присваивается ссылка на объект `StyleSheet`, содержимое этого текстового блока будет форматироваться в соответствии с правилами, описанными в таблице стилей.

При связывании таблицы стилей с текстовым блоком имеют место следующие особенности.

- Форматирование с применением стилей применяется вне зависимости от того, используется ли для текстового блока режим `html` или нет (то есть свойство `html` может быть не определено).

- Различия между свойствами `text` и `htmlText` стираются. Оба этих свойства так же, как и переменная, сопоставленная текстовому блоку, могут быть использованы для вывода текста в текстовое поле.
- Методы `setTextFormat()` и `replaceSel()` более не могут быть использованы для изменения форматирования и содержимого текстового блока. Тем не менее цвет текста можно изменить с помощью свойства текстового блока `textColor`. Изменения содержимого текстового блока могут быть выполнены только путем изменения значения свойств `text` или `htmlText` текстового блока или путем замены значения переменной, сопоставленной текстовому блоку. При этом к тексту, который вновь выводится в текстовое поле, будет применяться форматирование, определенное в таблице стилей.
- Текст становится недоступным для редактирования пользователем. Таким образом, текстовое поле может быть использовано только для чтения.

Определение стилей

В данном разделе будут рассмотрены возможности определения стилей с применением синтаксиса CSS. Для определения стиля можно воспользоваться одним из трех способов.

- Создание стилевого класса или псевдокласса (только для тега `<a>`).
- Переопределение встроенных тегов.
- Объявление новых тегов.

В самом общем виде объявление стиля имеет следующий синтаксис:

```
имя стиля {
свойство1 : значение;
свойство2 : значение;
...
свойствоN : значение
}
```

При наличии точки с запятой после задания значения свойства пары "свойство — значение" можно располагать в одну строку, однако приведенный формат синтаксиса делает определение стиля более читабельным и понятным.

Следует отметить, что свойства CSS при использовании в Flash регистрозависимы (то есть `text-align` и `Text-Align` — это не одно и то же). Значения свойств указываются без кавычек, вне зависимости от того, являются они числами или нет.

Создание стилевого класса

При создании стилевого класса используется следующий синтаксис:

```
.имя класса{
свойство1 : значение;
```

```
свойство2 : значение;
...
свойствоN : значение
}
```

Примечание

Обратите внимание, что перед именем класса в объявлении обязательно помещается точка!

Стилевой класс может быть использован для форматирования текста только в сочетании с тегами <p> и/или . Оба данных тега поддерживают атрибут class, значением которого является имя стилевого класса, применяемого для форматирования соответствующего фрагмента текста. Таким образом, для того чтобы применить форматирование, описываемое в стилевом классе, к текстовому фрагменту, необходимо разметить этот фрагмент тегом <p> или и присвоить имя класса атрибуту class данного тега. Рассмотрим небольшой пример:

```
.emphasized{
font-family:Pragmatica, Arial, _sans;
font-size:20;
color:#0000FF;
font-weight:bold
}
```

Стилевой класс emphasized устанавливает правила форматирования, в соответствии с которыми текст выводится с применением шрифтов Pragmatica, Arial или _sans, кеглем 20 красного цвета с полужирным начертанием. Для применения этого форматирования можно воспользоваться тегом .

```
<span class = 'emphasized'>Знания не могут заменить ум.</span>
```

Обратите внимание, что при задании значения атрибуту class точка перед именем класса не используется.

Тег сам по себе не применяет форматирования к своему содержимому. Только при наличии атрибута class размечаемый этим тегом элемент будет отформатирован в соответствии с правилами, заданными в классе. Таким образом, с помощью тега можно отформатировать произвольный фрагмент текста. В отличие от него тег <p> сам по себе задает форматирование (создает новый абзац). Таким образом, при использовании стилевого класса в сочетании с тегом <p> форматирование применяется ко всему абзацу:

```
<p class = 'emphasized'>Смотрят все, а видят немногие</p>
```

Наряду с созданием стилевых классов в Flash поддерживаются псевдоклассы для элемента <a>. Эти псевдоклассы позволяют изменять форматирование элемента гиперссылки в зависимости от состояния курсора.

В Flash допускается применение трех (из четырех имеющихся в "настоящем" CSS) псевдоклассов элемента <a>:

- link — определяет форматирование неактивной гиперссылки (то есть гиперссылки в отсутствие взаимодействия);
- over — определяет форматирование гиперссылки при наведении на нее курсора мыши;
- active — определяет форматирование гиперссылки в момент нажатия на нее.

Применение псевдокласса гиперссылки предполагает использование следующего синтаксиса:

```
a:псевдокласс{
свойство1 : значение;
свойство2 : значение;
...
свойствоN : значение
}
```

Рассмотрим простой пример:

```
a:link{
       font-family:sans;
       text-decoration:none;
       color:#000099
}
a:hover{
       font-family:sans;
       text-decoration:underline;
       color:#0000FF
}
a:active{
       font-family:sans;
       text-decoration:none;
       color:#800080
}
```

В соответствии с объявленными правилами форматирования гиперссылка будет выводиться синим цветом, при наведении цвет будет изменяться и текст гиперссылки будет выводиться с подчеркиванием, а при нажатии подчеркивание будет исчезать. Эти правила будут автоматически применяться при использовании тега <a>:

```
<a href = "http://www.avalon.ru">Факультет Переподготовки
Специалистов</a>
```

Примечание

Flash Player не поддерживает четвертый стандартный псевдокласс элемента <a>, используемый в CSS, который называется visited. С помощью этого псевдокласса можно определять параметры форматирования, используемые для отображения посещенных ссылок.

Переопределение встроенных тегов

Используя возможности CSS, можно устанавливать параметры форматирования выполняемого стандартными тегами HTML. Это называется переопределением тега. В Flash переопределение тегов можно применять только к достаточно ограниченному набору элементов. К их числу относятся:

- тег <a>;
- тег <p>;
- тег <li>.

При объявлении стиля, выполняющего переопределение встроенного тега, применяется следующий синтаксис:

```
элемент{
свойство1 : значение;
свойство2 : значение;
...
свойствоN : значение
}
```

Здесь элемент — это название стиля, совпадающее с именем переопределяемого тега.

В качестве примера создадим два стиля, определяющих параметры форматирования, для тегов <p> и <li>.

```
p{
        font-family: Arial, sans;
        font-size: 15;
        font-weight: bold;
        color: #660000
}
li{
        font-family: Times New Roman, serif;
        font-size: 10;
        font-weight: normal;
        color: #FF8800
}
```

Правила форматирования, определенные стилями `p` и `li`, будут автоматически применяться к тексту, размеченному соответствующими тегами:

```
<p>Типы текста в Flash</p><li>Static</li><li>Dynamic</li><li>Input</li>
```

Отформатированный таким образом текст будет иметь следующий вид:

Типы текста в Flash

- Static
- Dynamic
- Input

Объявление новых тегов

Наряду с применением стилевых классов и переопределением некоторых встроенных тегов, с помощью CSS можно создавать собственные теги и определять правила выполняемого ими форматирования. Так, например, можно создать тег `header` для разметки заголовков, тег `subtitle` для разметки подзаголовков, тег `emphasized` для текста, который должен быть визуально выделен, и т. д. Эта возможность оказывается чрезвычайно полезной при работе с XML-документами, состоящими из набора тегов, определенных пользователем. Форматирование, выполняемое такими тегами, определяется правилами, заданными в таблице стилей.

Синтаксически объявление собственных тегов выполняется так же, как и переопределение встроенных. В следующем примере создаются стили `header`, `subtitle` и `body`.

```
header{
       font-family: Arial, sans;
       font-size: 25;
       font-weight: bold;
       color: #660000
}
subtitle{
       font-family: Times New Roman, serif;
       font-size: 20;
       text-decoration: underline;
       color: #005050
}
body{
       font-family: Times New Roman, serif;
       font-size: 15;
       color: #005050
}
```

Для применения форматирования, определяемого этими стилями, нужно использовать теги, имена которых совпадают с именами соответствующих стилей:

```
<header>Заголовок</header>
<subtitle>Подзаголовок</subtitle>
<body>Основной текст</body>
```

Следует иметь в виду, что при вложении тегов форматирование выполняется на основе принципа каскадности. Это означает, что вложенный тег наследует все параметры форматирования, определенные для родительского (то есть содержащего его) тега. Если для родительского и для вложенного тегов определены одни и те же свойства форматирования, то приоритет имеет вложенный тег.

Создадим два стиля:

```
body{
       font-family: Arial, sans;
       font-size: 10;
       color: #FF0000
}
emphasized{
       font-weight: bold
}
```

Стиль `emphasized` задает только жирное начертание для содержимого, размеченного тегом `<emphasized>`. Однако если тег `<emphasized>` вложен в тег `<body>`, то заключенный в него текст выводится с помощью шрифта Arial, десятым кеглем, синим цветом и с применением полужирного начертания:

```
<body>С пола упасть <emphasized>нельзя</emphasized></body><emphasized>
```

Результат будет выглядеть следующим образом:

С пола упасть **нельзя**

Добавление стилей в таблицу

Как уже упоминалось, создание стилей и их добавление в таблицу стилей, представляемую объектом `StyleSheet`, может выполняться различными способами. В предыдущем разделе были рассмотрены общие возможности определения стилей, поддерживаемые Flash. Здесь будут описаны конкретные способы их создания, добавления в таблицу стилей и использования. Еще раз кратко перечислим эти способы.

- Создание стиля с помощью объекта общего назначения.
- Описание стиля в виде строки.
- Загрузка стиля из внешнего CSS-файла.

Создание стиля с помощью объекта общего назначения

Данный способ предполагает не использование CSS как такового, а моделирование его применения с помощью ActionScript. Для создания стиля, определяющего правила форматирования, используется объект `Object`. Правила форматирования задаются путем добавления этому объекту набора свойств, перечисленных в табл. 14.2. Таким образом, стиль представляет собой объект с набором допустимых свойств, имена которых являются производными от имен соответствующих свойств CSS (из имени CSS-свойства удаляется дефис, и каждое слово в имени, начиная со второго, выделяется прописной буквой). При необходимости использования нескольких стилей для каждого создается отдельный объект. Рассмотрим пример создания стиля. Вначале создадим объект `StyleSheet` для хранения таблицы стилей:

```
var myStyle = new TextField.StyleSheet();
```

Затем создадим объект, представляющий стиль заголовка:

```
var header:Object = {
       fontFamily:'Pragmatica, Arial Cyr, _sans',
       fontSize:'20',
       color:'#FF0000',
       fontWeight:'bold',
       display:'block'
       };
```

В приведенной выше записи намеренно выполнена разбивка на строки для лучшей читабельности кода и для внешней схожести с объявлением стандартного CSS-стиля. Обратите внимание на необходимость применения кавычек при установке значений свойств. Все свойства ActionScript, соответствующие свойствам CSS, принимают только строковые значения.

Для добавления стиля, представленного объектом `Object`, в таблицу стилей, представленную объектом `StyleSheet`, используется метод `setStyle()` класса `StyleSheet`. Этот метод имеет следующий синтаксис:

таблица стилей.setStyle(*имя стиля*, *объект*)

Здесь *таблица стилей* — это объект `StyleSheet`, представляющий таблицу стилей.

Параметр *имя стиля* — это строка, задающая имя стиля.

Параметр *объект* — это объект, представляющий стиль с набором свойств, определяющих правило форматирования. Следует отметить, что данный параметр может быть задан не только идентификатором, содержащим ссылку на объект, но также и непосредственно литералом объекта.

В результате вызова данного метода и передачи ему требуемых параметров в объекте *таблица стилей* создаются два свойства `_css` и `_styles`.

Свойство `_css` — это объект `Object`, в который в качестве свойства копируется объект *объект*, переданный методу `setStyle()`. При каждом вызове метода `setStyle()` в объекте `_css` создается новое свойство, значением которого является копия объекта, переданного методу `setStyle()`. Таким образом, объект `_css`, буду-

чи свойством объекта StyleSheet, в свою очередь, содержит набор свойств, представляющих собой объекты. Эти объекты являются копиями объектов, содержащих определения стилей, которые передаются методу setStyle() в качестве параметров при каждом его вызове.

Свойство _styles — это также объект Object, свойствами которого являются объекты TextFormat. Значения свойств этих объектов устанавливаются на основании параметров форматирования, переданных методу setStyle() с помощью параметра *объект*.

Итак, добавим стиль, представленный созданным выше объектом header, в таблицу стилей, хранящуюся в объекте myStyle, присвоив ему имя title.

```
myStyle.setStyle("title", header);
```

Для того чтобы увидеть содержимое объекта после добавления в него стиля, можно в среде тестирования воспользоваться командой главного меню **Debug>List Variables**. В результате выполнения этой команды в панель **Output** будет выведено следующее (для краткости здесь показаны только те значения свойств объекта _styles, которые отличны от null):

```
_level0.myStyle = [object #1, class 'StyleSheet'] {
  _css:[object #2, class 'Object'] {
   title:[object #3, class 'Object'] {
    fontFamily:"Pragmatica, Arial Cyr, _sans",
    fontSize:"20",
    color:"#FF0000",
    fontWeight:"bold",
    display:"block"
   }
  },
  _styles:[object #4, class 'Object'] {
   title:[object #5, class 'TextFormat'] {
    font:[getter/setter] "Pragmatica,Arial Cyr,_sans",
    size:[getter/setter] 20,
    color:[getter/setter] 0xFF0000,
        bold:[getter/setter] true,
    display:[getter/setter] "block"
   }
  }
 }
```

Поскольку объекты, представляющие стили, при вызове метода setStyle() копируются в объект _css, их можно удалить для того, чтобы освободить ресурсы.

Для этого воспользуемся оператором delete:

```
delete header;
```

Если при вызове в качестве параметра метода setStyle() использовать не ссылку на объект, а литерал объекта, это избавляет от необходимости его удаления вручную.

Для того чтобы применить таблицу стилей, хранящуюся в объекте StyleSheet, для форматирования содержимого текстового блока, необходимо присвоить ссылку на этот объект свойству styleSheet текстового блока. Свойство styleSheet — это свойство объекта TextField, которое применяется для установки связи между текстовым блоком и таблицей стилей, в которой определены правила форматирования его содержимого. В качестве значения данное свойство принимает ссылку на объект StyleSheet, хранящий определения стилей. Как только свойству styleSheet текстового блока присваивается ссылка на объект StyleSheet, форматирование в этом блоке выполняется с применением указанной таблицы стилей. Для того чтобы разорвать связь между текстовым блоком и таблицей стилей, нужно присвоить свойству styleSheet значение null или undefined. В результате выполнения этой процедуры существующее форматирование текста не удаляется, однако изменение содержимого таблицы стилей уже не будет оказывать на текст никакого воздействия. После разрыва связи между текстовым блоком и таблицей стилей форматирование содержимого текстового блока можно будет вновь выполнять с применением методов setTextFormat() и setNewTextForamt().

Итак, создадим текстовый блок, установим для него связь с таблицей стилей myStyle и выведем в него текст:

```
this.createTextField("txt", 1, 0, 0, 0, 0);
txt.autoSize = true;
txt.styleSheet = myStyle;
txt.htmlText = "<title>Это заголовок</title>"
```

В листинге 14.13 приведен пример, иллюстрирующий всю процедуру создания и применения стилей с применением данного подхода. Результат действия этого сценария представлен на рис. 14.10.

Листинг 14.13. Применение стилей, определенных с помощью объектов Object

```
// Создание текстового блока и настройка его параметров
this.createTextField("txt", 1, 75, 0, 400, 400);
txt.border = true;
txt.autoSize = true;
txt.wordWrap = true;
txt.multiline = true;
txt.borderColor = 0x333333;
txt.html = true;
// Создание объекта для хранения таблицы стилей
var myStyle = new TextField.StyleSheet();
// Создание объектов, представляющих стили
```

```
var main:Object = {
       marginLeft:'10',
       marginRight:'10'
       };
var header:Object = {
       fontFamily:'Pragmatica, Arial Cyr, _sans',
       fontSize:'20',
       color:"#FF0000",
       fontWeight:'bold',
       display:'block'
       };
var subtitle:Object = {
       fontFamily:'Arial Cyr, _sans',
       fontSize:'12',
       fontWeight:'bold',
       color:'#990000',
       display:'block'
       }
var mainText:Object = {
       fontFamily:'Times New Roman, Arial Cyr, _sans',
       fontSize:'13',
       color:'#000000',
       textIndent:'35'
       };
// Добавление стилей в таблицу стилей, представляемую объектом myStyle
myStyle.setStyle('main', main);
myStyle.setStyle('title', header);
myStyle.setStyle('subtitle', subtitle);
myStyle.setStyle('body', mainText);
/* // Этот код нужно раскомментировать для вывода информации о параметрах
добавляемых в таблицу стилей
for(i in myStyle._css){
       trace("--------------"+"стиль "+i+"---------------");
       for(j in myStyle._css[i]){
              trace(j + ": " + myStyle._css[i][j]);
       }
}
*/
/* Удаляем объекты, содержащие описание стилей, так как они уже
скопированы в объект myStyle */
```

```
delete main;
delete header;
delete subtitle;
delete mainText;
// Установка связи между текстовым блоком txt и таблицей стилей myStyle
txt.styleSheet = myStyle;
/* Вывод текста с применением HTML-форматирования, реализуемого с помощью
таблицы стилей */
txt.htmlText = "<main><title>Разработка персона-
жа</title><subtitle>принципы формирования психологического
портрета</subtitle><body><br/><p>Герои, как правило, не похожи на
реальных существ, встречающихся в жизни. В этом и состоит прелесть
анимационного кино. Анимационный персонаж всегда содержит некоторую
условность, однако в то же время он должен быть совершенно убедительным.
<img src = 'pers' align = 'right'>Внешний вид персонажа должен полностью
соответствовать его роли в фильме. Таким образом, его психологический
портрет должен быть явно выражен графическими средствами.</p><p>. Для
начала нужно постараться сформировать персонаж из нескольких простых
геометрических форм, исходя из отведенной ему роли. Любая геометрическая
форма имеет свой характер, и в нашем восприятии формирует представление
о том или ином качестве. Так, формы, близкие к кругу, указывают на
некоторую мягкость, иногда комичность, квадратные — на мощность,
основательность, прямоту. Наличие острых углов свидетельствует о
направленной силе, жесткости, возможно, агрессии. Таким образом, общие
очертания персонажа уже дают о нем некоторое представление. Для того
чтобы сделать персонаж более читаемым и выделить его на общем фоне, как
правило, используется резкий контрастный контур.</p></body></main>";
```



Рис. 14.10. Результат выполнения сценария листинга 14.13

Описание стиля в виде строки

Для добавления стиля в таблицу стилей можно воспользоваться методом `parseCSS()` класса `StyleSheet`. Этот подход предполагает формирование строки, содержащей объявление стилей, выполненное с применением стандартного синтаксиса CSS. Сформированная таким образом строка передается в качестве параметра методу `parseCSS()`, который выполняет синтаксический разбор строки и добавляет объявленные стили в объект `StyleSheet`, представляющий таблицу стилей. Метод `parseCSS()` имеет следующий синтаксис:

таблица стилей.parseCSS(*строка стилей*)

Здесь *таблица стилей* — это объект `StyleSheet`, представляющий таблицу стилей, в которую добавляются новые стили.

Параметр *строка стилей* — это строка, содержащая объявление стилей, выполненное в соответствии с синтаксисом CSS *(см. разд. "Определение стилей" данной главы)*. Данный параметр может быть задан строковым литералом, идентификатором, содержащим строку, или выражением, возвращающим строку.

При добавлении стиля, который уже имеется в таблице, добавляются только новые свойства или свойства, значения которых изменяются.

Метод `parseCSS()` возвращает значение `true`, если синтаксический разбор переданной ему строки выполнен успешно, и значение `false`, если при разборе строки обнаружена синтаксическая ошибка. Следует иметь в виду, что если синтаксическая ошибка заключается в неверном указании имени свойства или в неверном указании его значения, то это не приводит к прекращению действия метода. В этом случае в свойство `_css` объекта `StyleSheet` по-прежнему будет добавлено переданное свойство (как если бы оно было задано корректно), однако это не приведет к каким-либо изменениям в свойстве `_styles` (то есть не отразится на свойствах объекта `TextFormat`, принадлежащего объекту `StyleSheet`). Это означает, что в результате анализа строки, переданной методу `parseCSS()`, в таблицу стилей добавляются только поддерживаемые свойства с допустимыми значениями. Остальные не вызывают ошибки, но и не дают видимого результата. Метод `parseCSS()` возвращает значение `false` только в случае невозможности анализа строки, при наличии в ней существенных синтаксических ошибок, например, при отсутствии фигурных скобок.

Для того чтобы проконтролировать, успешно ли выполнен разбор строки, метод `parseCSS()` обычно вызывается в проверочном выражении условного предложения. Если метод `parseCSS()` возвращает значение `true`, выполняется связывание текстового блока с таблицей стилей, в противном случае можно вывести сообщение о невозможности применения стиля в связи с наличием синтаксических ошибок.

Следующий пример демонстрирует принцип использования данного подхода.

```
// Создание текстового блока
this.createTextField("txt", 1, 175, 100, 200, 1);
txt.autoSize = true;
txt.html = true;
// Создание пустой таблицы стилей
var myStyle:TextField.StyleSheet = new TextField.StyleSheet();
```

```
// Сохранение строки с объявлением стиля header в переменной
var s:String = "header{fontdd-face:Arial, _sans; font-size:25;↳
color:#0000FF}";
// Добавление стиля в таблицу в случае успешного разбора строки
if (myStyle.parseCSS(s)) {
        txt.styleSheet = myStyle;
}
else {
        trace("Ошибка в объявлении стиля header");
}
// Вывод текста с применением стиля header
txt.htmlText = "<header>Заголовок</header>";
```

В листинге 14.14 приведен сценарий, создающий список гиперссылок. Для этих целей применяется стиль `li` (маркированный список) и псевдоклассы гиперссылки (`link`, `hover`, `active`):

Листинг 14.14. Применение стилей, определенных с помощью строки

```
// Создаем текстовый блок и задаем его параметры
this.createTextField("txt", 1, 175, 100, 200, 1);
txt.border = true;
txt.autoSize = true;
txt.multiline = true;
txt.wordWrap = true;
txt.borderColor = 0x333333;
txt.html = true;
// Создаем объект StyleSheet для хранения таблицы стилей
var myStyle:TextField.StyleSheet = new TextField.StyleSheet();
/* Функция, добавляющая новый стиль, который передается ей в виде строки,
в таблицу стилей myStyle. Если переданная строка не является синтаксически
корректной, выводится соответствующее сообщение и добавления стиля не
происходит */
function parse(CSS:String) {
        if (myStyle.parseCSS(CSS)) {
                txt.styleSheet = myStyle;
        }
        else {
                trace("Ошибка при анализе строки CSS: " + CSS);
        }
}
```

```
/* Переменная для хранения строки со стилем для разметки маркированного
списка */
var bulletedList:String = "li{font-family:Arial, _sans; font-size:14}";
// Переменная для хранения строки со стилем для разметки гиперссылки
var hyperlink:String = "a:link{color:#000088;text-decoration:Underline}
a:hover{color:#0000FF;text-decoration:underline}a:active{color: #0000FF;
text-decoration:none}";
// Добавляем стили в таблицу стилей myStyle
parse(bulletedList);
parse(hyperlink);
/* Переменные, содержащие фрагменты HTML-текста, предназначенного для
вывода */
var header = "<p align = 'center'><font face = '_sans' size = '20' color
= '#CC3333'><b>Flash-ресурсы</b></font></p>";
var link1 = "<a href= 'http://www.actionscript.org' target =
'_blank'>www.actionscript.org </a>";
var link2 = "<a href= 'http://www.ultrashock.com' target =
'_blank'>www.ultrashock.com </a>";
var link3 = "<a href= 'http://www.flash-ripper.com' target =
'_blank'>www.flash-ripper.com</a>";
var link4 = "<a href= 'http://www.multikov.net' target =
'_blank'>www.multikov.net</a>";
var link5 = "<a href= 'http://www.flasher.ru' target =
'_blank'>www.flasher.ru</a>";
// Вывод отформатированного текста в поле txt
txt.htmlText = header; // Заголовок
// Создание списка ссылок с помощью конкатенации строк
for (var i = 1; i < 6; i++) {
       txt.htmlText += "<li>" + this["link" + i]+ "</li>";
}
```

Результат выполнения данного сценария представлен на рис. 14.11.



Рис. 14.11. Создание списка ссылок

Загрузка стиля из внешнего CSS-файла

Наряду с возможностью объявления стилей непосредственно внутри фильма Flash предоставляет возможность применения внешних таблиц стилей, представляющих собой документы CSS. Эта возможность на практике, пожалуй, является наиболее важной и полезной, так как хранение информации о форматировании в отдельном документе позволяет быстро и легко выполнять изменение форматирования. Зачастую одна (или несколько) таблица стилей используется несколькими документами, поэтому при необходимости внесения изменений в параметры форматирования достаточно просто изменить соответствующий файл CSS, не прибегая при этом к редактированию содержимого SWF-документов (что требует значительно больших затрат времени).

Для загрузки внешней таблицы стилей используется метод `load()` класса `StyleSheet`. Данный метод имеет следующий синтаксис:

```
таблица стилей.load(путь)
```

Здесь *таблица стилей* — это объект `StyleSheet`, представляющий таблицу стилей, в которую добавляются стили из внешнего документа.

Параметр *путь* задает путь к файлу CSS, содержащему объявление стилей. Этот параметр задается строковым литералом или выражением, возвращающим строку.

Следует отметить, что загрузка внешней таблицы стилей выполняется асинхронно. Это означает, что вывод текста в текстовое поле по времени никак не увязывается с окончанием загрузки таблицы стилей, используемой для форматирования этого текста. Для того чтобы отследить факт окончания загрузки внешней таблицы стилей, используется обработчик `onLoad`. Событие `onLoad` генерируется непосредственно после выполнения операции, инициированной вызовом метода `load()`, то есть после полного окончания загрузки таблицы стилей или в случае прекращения загрузки вследствие ее недоступности или возникшей ошибки. Метод обработчика `onLoad` может быть определен для любого объекта `StyleSheet`. При этом используется следующий синтаксис:

```
таблица стилей.onLoad = function(статус){
блок кода;
...
};
```

Параметр *статус*, передаваемый функции, которая присваивается свойству `onLoad`, принимает значение `true` или `false`. Если внешняя таблица стилей успешно загрузилась, параметр *статус* принимает значение `true`. Если таблица не может быть найдена по указанному адресу или если при получении данных с сервера возникла ошибка, параметр *статус* принимает значение `false`.

Для того чтобы избежать вывода текста до окончания загрузки таблицы стилей, определяющей правила его форматирования, имеет смысл определить метод обработ-

чика onLoad, выводящий текст в текстовое поле только в случае, если параметр *статус* принимает значение true.

Такой подход демонстрируется в листинге 14.15. Таблица стилей, используемая для форматирования текста, представлена далее (исходный файл расположен на сопровождающем книгу CD по адресу Глава 14/CSS/styles1.css):

```
.header{font-family:Pragmatica,Arial Cyr, _sans;
font-size:20;
color:#FE9090;
font-weight:bold;
text-indent:10;
display:block
}
.main{font-family:Times New Roman, _serif;
font-size:14;
color:#333333;
display:block;
margin-left:10
}
.emphasized{
color:#FF0000;
font-weight:bold;
display:inline
}
.caption{
color:#666666;
font-family:Arial Cyr, _sans;
font-size:12;
font-weight:bold;
text-decoration:underline;
text-align:left;
display:block
}
```

Файл с таблицей стилей сохранен с именем styles1.css в каталоге CSS, расположенном в том же каталоге, что и страница, содержащая результирующий SWF-фильм.

Наряду с текстом в текстовый блок внедряются два внешних SWF-документа с именами splash.swf и tick.swf, расположенные в каталоге SWF.

Листинг 14.15. Применение внешней таблицы стилей

```
// Создаем текстовый блок и задаем его параметры
this.createTextField("txt", 1, 0, 0, 575, 400);
txt.border = true;
txt.autoSize = true;
txt.multiline = true;
txt.wordWrap = true;
txt.borderColor = 0x333333;
txt.html = true;
// Создаем объект StyleSheet для хранения таблицы стилей
var myStyle = new TextField.StyleSheet();
/* Определение обработчика onLoad. По окончании успешной загрузки
связываем текстовый блок с таблицей стилей и выводим в него
отформатированный текст. Если загрузка не выполнилась, выводим
соответствующее сообщение" */
myStyle.onLoad = function(success){
       if(success){
       txt.styleSheet = myStyle;
       txt.htmlText = "<p class = 'header'>Анонс</p><img src =
       'swf/splash.swf' width = '350' height = '125' hspace = '10' vspace
       = '0'><p class = 'main'><span class = 'emphasized'>Flash</span> -
       это чрезвычайно мощное и простое в использовании средство создания
       мультимедийных проектов со встроенной поддержкой интерактивности.
       <span class = 'emphasized'>Простота</span> использования и
       широчайшие <span class = 'emphasized'>возможности</span>
       этого пакета являются причинами его огромной <span class =
       'emphasized'>популярности</span>. Flash объединяет в себе три
       компонента, предоставляющие возможность создания векторной
       графики, анимационных фильмов, снабженных звуком, и
       интерактивности, позволяющей пользователю взаимодействовать
       с проектом.<br/><img src = 'swf/tick.swf' width = '10' height =
       '15' align = 'right' vspace = '0'><p class = 'caption'><a href =
       'http://www.avalon.ru'>Краткосрочные компьютерные курсы</a></p>";
       }
else {
txt.htmlText = "Ошибка при загрузке таблицы стилей";
       }
};
// Загрузка внешней таблицы стилей
myStyle.load("CSS/styles1.css");
```

Результат действия этого сценария представлен на рис. 14.12.



Рис. 14.12. Применение внешней таблицы стилей

Получение информации об используемых стилях

В классе `StyleSheet` определены два метода, позволяющие получать информацию о стилях, хранящихся в объекте `StyleSheet`, представляющем таблицу стилей. К их числу относятся метод `getStyle()` и метод `getStyleNames()`.

Метод `getStyle()` возвращает объект, содержащий набор свойств, соответствующих свойствам форматирования, заданных в указанном стиле. Фактически значение, возвращаемое данным методом, представляет собой копию свойства объекта `_css`, который (как было показано) содержит информацию о стилях, помещенных в таблицу стилей, представленную объектом `StyleSheet`. Метод `getStyle()` имеет следующий синтаксис:

таблица стилей`.getStyle(`*стиль*`)`

Здесь *таблица стилей* — это объект `StyleSheet`, представляющий таблицу стилей.

Параметр *стиль* — это строка с именем стиля, копию которого в виде объекта требуется получить. Данный параметр может быть задан с помощью строкового литерала или выражения, возвращающего строку.

Так, если имеется некий объект `StyleSheet` с именем `myStyle`, содержащий стилевой класс `.header`, то для получения копии объекта, представляющего данный стиль, можно использовать следующий код:

```
var headerStyle:Object = myStyle.getStyle(".header");
```

Для вывода перечня свойств форматирования, определенных в этом стиле, удобно воспользоваться циклом `for in`:

```
for (var i in headerStyle){
trace (i + " = "+ headerStyle[i]);
}
```

Метод `getStyleNames()` возвращает массив, содержащий имена стилей (в виде строк), помещенных в таблицу стилей, представленную объектом `StyleSheet`. Этот метод имеет следующий синтаксис:

```
таблица_стилей.getStyleNames()
```

Так, для того чтобы получить перечень стилей, содержащихся в объекте `myStyle` из листинга 14.15, можно воспользоваться следующим кодом:

```
var stylesList:Array = myStyle.getStyleNames();
trace(stylesList.join("\n"));
```

В листинге 14.16 приведен сценарий, позволяющий вывести перечень всех стилей и определенных в них правил форматирования для таблицы стилей, представленной объектом `myStyle`. Следует обратить внимание на то, что при использовании внешней таблицы стилей получить данную информацию можно только после окончания ее загрузки. Поэтому в данном случае код, выполняющий вывод этой информации, помещен в метод обработчика события `onLoad`.

Листинг 14.16. Вывод информации о стилях, помещенных в таблицу стилей

```
// Создаем объект StyleSheet для хранения таблицы стилей
var myStyle = new TextField.StyleSheet();
//Вывод информации о загруженных стилях
myStyle.onLoad = function(success) {
       if (success) {
              // Создаем массив с именами стилей
              var styles:Array = myStyle.getStyleNames();
              // Для каждого стиля из массива styles...
              for (var i = 0; i < styles.length; i++) {
              trace("-------Стиль " + styles[i] + "-------");
                     // ...выводим список свойств со значениями
                     for (var j in myStyle.getStyle(styles[i])) {
                     trace(j + " = " + myStyle.getStyle(styles[i])[j]);
                     }
              }
       }
       else {
              trace("Ошибка при загрузке таблицы стилей");
       }
};
```

Очистка таблицы стилей

Для того чтобы удалить все стили, добавленные в таблицу, представленную объектом StyleSheet, можно воспользоваться методом clear(). Метод clear() имеет следующий синтаксис:

таблица стилей.clear()

Здесь *таблица стилей* — это объект StyleSheet, из которого требуется удалить стили.

Действие метода clear() заключается в том, что он удаляет все свойства из объектов _css и _styles, очищая таким образом таблицу стилей, хранящуюся в объекте StyleSheet. При этом сами объекты _css и _styles остаются пустыми, но не удаляются.

В результате очистки стилей все форматирование, используемое в текстовом блоке, удаляется, и текст выводится с применением параметров форматирования, используемых по умолчанию. Однако вызов метода clear() не приводит к утрате связи между таблицей стилей и текстовым блоком. Поэтому применение методов setTextFormat() и setNewTextForamt() не будет оказывать никакого результата. Для того чтобы разорвать связь между таблицей стилей и текстовым блоком, необходимо присвоить свойству styleSheet текстового блока значение null или undefined. После этого для форматирования текстового блока можно будет вновь использовать возможности класса TextFormat.

Загрузка текстовых данных из XML-документа

Одной из важнейших особенностей работы с текстом в Flash является возможность загрузки внешнего XML-документа, отформатированного с применением внешней таблицы стилей. Такой подход позволяет хранить отдельно от SWF-документа его текстовое наполнение, правила форматирования, используемые при выводе текста, а также изображения и ролики, предназначенные для встраивания в текстовое поле. Это дает возможность обновления содержимого текстовых блоков без необходимости редактирования самого Flash-документа.

В этой книге мы не будем останавливаться на описании языка XML и правил создания XML-документов. Это чрезвычайно объемная тема, и читатель, желающий ознакомиться с этим вопросом, может обратиться к специальной литературе. Для понимания механизма загрузки отформатированного содержимого из XML-документа достаточно лишь самых общих знаний.

Документ XML представляет собой дерево, состоящее из иерархически вложенных друг в друга элементов — узлов. Каждый элемент документа ограничен парой тегов, так же как и в HTML. Если элемент не имеет содержимого, открывающий и закрывающий теги объединяются в один, например:

```
<br/>
```

В отличие от HTML, в котором все имена тегов являются зарезервированными, в XML имена могут быть произвольными. При этом в именах можно применять не только символы, относящиеся к кодировке ASCII, поскольку XML поддерживает Unicode. Это означает, что имена могут задаваться, в частности, на русском языке.

В документах, подлежащих загрузке в Flash-фильм, нужно использовать стандарт кодирования UTF-8.

Имена тегов не должны начинаться с цифры, суффикса `xml` (вне зависимости от регистра) и точки с дефисом. Также в именах не рекомендуется использовать двоеточие. Наряду с именами пользовательских тегов XML-документ может содержать имена стандартных тегов HTML. При загрузке в Flash элементы, размеченные стандартными HTML-тегами, которые поддерживаются проигрывателем Flash Player, будут выводиться с применением соответствующего форматирования. Элементы, размеченные пользовательскими тегами, будут выводиться в соответствии с правилами форматирования, определенными для этих тегов в таблице стилей, связанной с текстовым блоком, в который выводится содержимое XML-документа.

Таким образом, в общих чертах процесс загрузки данных из XML-документа предполагает следующую последовательность действий.

1. Создание таблицы стилей, определяющей правила форматирования содержимого XML-документа.
2. Создание XML-документа, содержащего данные, предназначенные для вывода в текстовый блок.
3. Загрузка таблицы стилей в Flash-документ и связывание ее с текстовым блоком, в который будет выводиться загруженная информация.
4. Загрузка XML-документа в Flash-документ и вывод его отформатированного содержимого в соответствующий текстовый блок.

Для загрузки XML-документа используется следующий подход.

Вначале создается объект `XML` как экземпляр встроенного класса `XML`. Для этого используется следующий синтаксис:

```
var объект XML:XML = объект XML new XML();
```

Здесь *объект XML* — это переменная, в которой сохраняется ссылка на созданный объект.

Загрузка внешнего XML-документа выполняется с помощью встроенного метода `load()` класса `XML`. В результате вызова метода `load()` данные из внешнего XML-документа помещаются в объект `XML`, вызвавший этот метод. Метод `load()` имеет следующий синтаксис:

```
объект XML.load(путь);
```

Параметр *путь* — это строка, содержащая путь к загружаемому XML-документу. Этот параметр может быть задан с помощью строкового литерала или выражения, возвращающего строку. Следует иметь в виду, что загружаемый XML-документ должен находиться в том же домене, что и HTML-страница, на которой находится Flash-фильм, осуществляющий загрузку. Это связано с политикой безопасности, используемой в седьмой версии проигрывателя Flash Player. Подробно информация об этом будет рассмотрена в *главе 20*.

Загрузка внешнего XML-документа так же, как и загрузка внешней таблицы стилей, выполняется асинхронно. Для того чтобы отследить момент окончания загрузки или обнаружить факт наличия ошибки, препятствующей загрузке, можно воспользовать-

ся обработчиком события onLoad, существующем в классе XML. Синтаксис обработчика события onLoad аналогичен синтаксису одноименного обработчика класса StyleSheet.

```
объект XML.onLoad = function(статус){
блок кода;
...
};
```

Параметр *статус*, передаваемый функции, которая присваивается свойству onLoad, принимает значение true или false. Если внешний XML-документ успешно загрузился, параметр *статус* принимает значение true. Если внешний XML-документ не может быть найден по указанному адресу или если при получении данных с сервера возникла ошибка, параметр *статус* принимает значение false.

Для того чтобы избежать вывода информации в текстовое поле до окончания загрузки внешнего XML-документа, необходимо определить метод обработчика onLoad, выводящий текст в текстовое поле только в случае, если параметр *статус* принимает значение true (точно так же, как и при загрузке внешней таблицы стилей).

Итак, рассмотрим процесс загрузки внешнего XML-документа на конкретном примере. Создадим модель каталога живописных работ. Каталог представляет собой текстовое поле, снабженное вертикальной полосой прокрутки. Все содержимое, которое будет выводиться в текстовое поле, описывается во внешнем файле XML. Изображения в формате JPEG будут храниться во внешних файлах и внедряться в текстовый блок с помощью тега <img>. Форматирование текста осуществляется с помощью таблицы стилей, хранящейся во внешнем CSS-документе. Таким образом, сам Flash-документ будет представлять собой оболочку, содержащую текстовый блок и полосу прокрутки. Вся информация, выводящаяся в этом текстовом блоке, хранится отдельно во внешних документах. Это обеспечивает очень удобные возможности обновления и форматирования содержимого. Полностью готовый пример оформлен как Flash-проект (Project) и сохранен на сопровождающем книгу CD под именем 14_17.flp. Внешний вид проекта представлен на рис. 14.13.

Создадим таблицу стилей и сохраним ее в файле с именем styles2.css (исходный файл расположен на сопровождающем CD по адресу Глава 14/CSS/styles2.css).

```
title{
text-align:center;
font-family:Arial Cyr, Helvetica, sans-serif;
color:#993333;
font-size:18;
font-weight:bold;
display: block;
}
subtitle{
text-align:center;
```

```
font-style:italic;
font-family:Times New Roman, Times, serif;
color:#666666;
font-size:11;
font-weight:normal;
display:block
}
body{
margin-left:150;
text-align:left;
font-family:Arial, _sans;
color:#666666;
display:block
}
a:link{
font-family: Arial, _sans;
font-size:12;
color: #555555;
text-decoration: underline;
font-weight: normal
}
a:hover{
font-family: Arial, _sans;
font-size:12;
color: #0000FF;
text-decoration: underline;
font-weight: normal;
}
a:active{
font-family: Arial, _sans;
color: #0000FF;
font-size: 12;
text-decoration: none;
font-weight: normal;
}
caption{
font-family:Times New Roman, Times, serif;
text-align:center;
```

```
font-size: 12;
color:#666666;
display:block;
}
```

Теперь создадим XML-документ, описывающий данные, которые должны выводиться в текстовый блок. Этот документ сохраним в каталоге xml под именем content.xml (исходный файл расположен на сопровождающем CD по адресу Глава 14/xml/content.xml).

```
<content>
<title>ДМИТРИЙ АЛЬБЕРТ<br/>ЖИВОПИСЬ</title>
<subtitle>(масло, пастель)</subtitle>
<br/>
<body>
<img src="images/Image1.jpg" width="93" height="122"
hspace="30"vspace="10"/><br/><br/><br/><br/><li><b>Композиция #2.
</b>30x40, б., п. 2002.</li><br/><br/><br/><br/><br/>
<img src="images/Image2.jpg" width="120" height="93" hspace="18"
vspace="-20"/><li><b>Утро. Открытие мира.</b> 80x90, х.м.,
2003.</li><br/><br/><br/><br/>
<img src="images/Image3.jpg" width="135" height="93" align="left"
hspace="10" vspace="-20"/><li><b>Настройка оркестра.</b> 30x40, б., п.
2003.</li><br/><br/>
<img src="images/Image4.jpg" width="90" height="122"
hspace="30"vspace="10"/><br/><br/><br/><li><b>Муринский ручей. Июль
</b>30x40, орг., м. 2001.</li><br/><br/>
</body>
<caption>
Отзывы и комментарии направляйте по адресу<br/><a
href="mailto:book_flash.mail.ru">book_flash.mail.ru</a>
</caption>
</content>
```

Поскольку XML-документ представляет собой дерево элементов, все теги, расположенные на одном уровне, должны быть заключены в родительский тег (узел). В нашем примере роль такого корневого тега выполняет тег, который мы назвали <content>. В принципе, его наличие не дает какого-то конкретного результата, однако для того чтобы не отступать от правил, мы включили его в код разметки. Наличие большого количества тегов переноса строки
 вызвано необходимостью равномерного расположения изображений в текстовом блоке. Растровые изображения, которые внедряются в текстовое поле, располагаются в каталоге images.

В рабочей среде Flash создадим слой, на котором разместим элементы полосы прокрутки. Процедура ее создания была описана в *разд. "Создание вертикальной полосы прокрутки текста" данной главы* (программный код приведен в листинге 14.5). В тот

же кадр, но на другом слое, поместим сценарий, приведенный в листинге 14.17. Результат действия этого сценария приведен на рис. 14.13.

Листинг 14.17. Вывод содержимого XML-документа, отформатированного с применением внешней таблицы стилей

```
// Создание текстового блока и настройка его параметров
this.createTextField("txt", 1, 0, 0, 355, 400);
txt.wordWrap = true;
txt.border = true;
txt.multiline = true;
txt.html = true;
// Создаем объект для хранения таблицы стилей
var myStyleSheet = new TextField.StyleSheet();
// Переменная для хранения пути к файлу с таблицей стилей
var stylesURL:String = "css/styles2.css";
// Проверка окончания загрузки таблицы стилей
myStyleSheet.onLoad = function(success) {
       if (success) {
              txt.styleSheet = myStyleSheet;
       } else {
              txt.htmlText = "<b>Ошибка!</b> Не удалось загрузить CSS";
       }
};
// Загрузка стилей
myStyleSheet.load(stylesURL);
// Создаем объект XML
var myContent:XML = new XML();
// Переменная для хранения пути к файлу xml
var contentURL:String = "xml/content.xml";
// Проверка окончания загрузки xml
myContent.onLoad = function(success) {
       if (success) { // Если документ загружен...
       // Выводим текст в поле txt
              txt.htmlText = myContent;
       } else {
       // В противном случае выводим сообщение об ошибке
              txt.htmlText = "<b>Ошибка!</b> Не удалось загрузить XML";
       }
};
```

```
// Загружаем документ xml
myContent.load(contentURL);
```

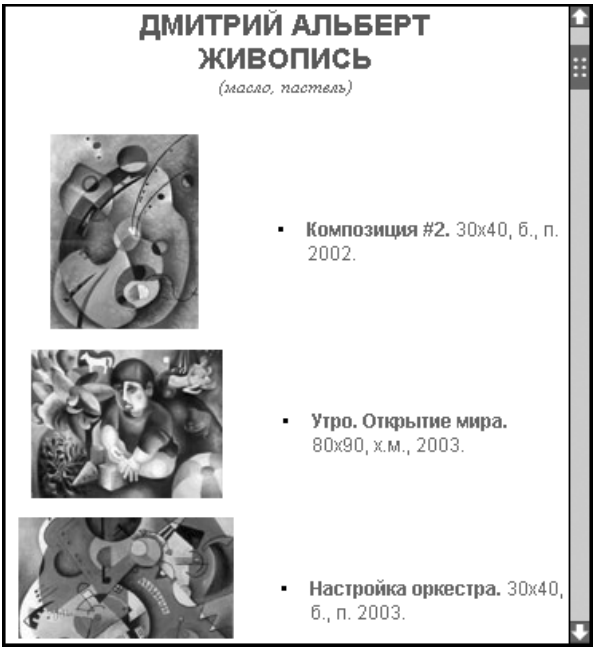


Рис. 14.13. Создание галереи с динамической загрузкой данных в форматах XML, CSS и JPEG

Контроль статического текста. Класс *TextSnapshot*

В этой главе до сих пор речь шла о работе с динамическим и пользовательским текстом. Характерной особенностью этих типов текста является возможность реализации программного контроля всех параметров текста, включая содержание, форматирование и обработку событий. Статический текст (Static text) в отличие от пользовательского и динамического создается исключительно в рабочей среде и не может подвергаться никаким изменениям в ходе воспроизведения фильма. Тем не менее в седьмой версии Flash (Flash MX 2004) появились новые возможности, позволяющие контролировать статический текст, расположенный на монтажной линейке клипа, не изменяя при этом содержание текста и параметры его форматирования. Эти возможности реализуются с помощью встроенного класса `TextSnapshot`.

Создание объекта *TextSnapshot*

Для того чтобы получить возможность обработки статических текстовых полей в клипе, необходимо создать объект `TextSnapshot` как экземпляр класса `TextSnapshot`. Объект `TextSnapshot` представляет собой "снимок" всего статического текста, расположенного на монтажной линейке клипа. Он отражает содержимое всех статических текстовых полей в клипе.

Создание объекта `TextSnapshot` осуществляется с помощью метода `getTextSnapShot()` класса `MovieClip` или с помощью конструктора и оператора `new`. Ссылку на созданный объект необходимо сохранить в переменной.

Создание объекта `TextSnapshot` с помощью метода `getTextSnapShot()` имеет следующий синтаксис:

```
var снимок:TextSnapshot = клип.getTextSnapshot();
```

Здесь *снимок* — это идентификатор переменной, в которой сохраняется ссылка на объект `TextSnapshot`.

Объект *клип* — это клип, содержащий статические текстовые поля, содержимое которых будет отражено в объекте `TextSnapshot`.

Создание объекта `TextSnapshot` с помощью конструктора выполняется следующим образом:

```
var снимок:TextSnapshot = new getTextSnapshot(клип);
```

При создании объекта `TextSnapshot` имеет место несколько характерных особенностей.

- Объект `TextSnapshot` включает текст из всех статичеких текстовых полей, расположенных на монтажной линейке клипа. Объект `TextSnapshot` не отражает текстового содержимого, расположенного во вложенных клипах.
- Содержимое статических текстовых блоков клипа конкатенируется в единую строку и помещается в объект `TextSnapshot` в порядке, соответствующем расположению текстовых блоков в стеке. Блоки, расположенные в стеке ниже, попадают в начало строки, блоки, расположенные выше, попадают в конец. Строка не содержит никаких символов, указывающих на конец предыдущего блока и начало следующего.
- Объект `TextSnapshot` предоставляет информацию о состоянии статического текста в клипе на момент своего создания. Если после создания объекта `TextSnapshot` статический текст в клипе подвергся изменениям (например, вследствие смены кадров и появления нового текстового блока или удаления старого), эти изменения не фиксируются в объекте `TextSnapshot` автоматически.
- Объект `TextSnapshot` предоставляет информацию только о статическом тексте, расположенном на монтажной линейке клипа. В связи с этим с его помощью можно контролировать только невыделяемый текст (текст с отключенным режимом **Selectable**). Другим важным условием является то, что шрифт должен быть встроен в конечный документ. Это означает, что статический текст, использую-

щий машинно-независимый или системный шрифт, не будет отражен в объекте TextSnapshot.

- Все возможности, предоставляемые классом TextSnapshot, предполагают доступ к статическому тексту в режиме "только для чтения", то есть не позволяют изменять его содержание и настройки.

Чтение и поиск в статическом тексте

Описываемые в этом разделе возможности позволяют извлекать текст из статических текстовых блоков.

Количество символов в текстовых полях

Для получения информации о количестве символов в объекте TextSnapshot используется метод getCount(). Он имеет следующий синтаксис:

снимок.getCount()

Здесь *снимок* — это объект TextSnapshot, содержащий текст из статических текстовых полей клипа.

Метод возвращает целое число, указывающее количество символов в строке, представляющей статический текст, которая хранится в объекте TextSnapshot.

Пусть в клипе mc имеется три статических текстовых блока. В первом находится слово "Flash" во втором — "MX", а в третьем — "2004". Для того чтобы вывести информацию о суммарном количестве символов в текстовых полях, используем следующий код:

```
var snapshot:TextSnapshot = mc.getTextSnapshot();
trace(snapshot.getCount()); // Выводит 11
```

Чтение текста

Для чтения текста, помещенного в статические текстовые блоки клипа, используется метод getText(). Метод getText() позволяет извлечь из объекта TextSnapshot диапазон символов на основании индексов начального и конечного символов диапазона и возвращает его в виде строки. Данный метод имеет следующий синтаксис:

снимок.getText(*начальный индекс*, *конечный индекс*, *разрывы строк*)

Параметр *начальный индекс* — задает индекс начального символа, входящего в извлекаемый диапазон. Индексация символов в строке начинается с нуля. Допустимыми значениями являются целые числа в диапазоне от 0 до getCount()−1. Если значение этого параметра отрицательно, используется значение 0.

Параметр *конечный индекс* — задает индекс символа, следующего за последним символом извлекаемого диапазона. Символ с индексом, заданным этим параметром, не входит в извлекаемую подстроку. Если этот параметр не указан, используется значение, возвращаемое методом getCount(). Если значение этого параметра равно значению параметра *начальный индекс*, в качестве его значения используется *начальный индекс*.+1.

Параметр *разрывы строк* — принимает булево значение `true` или `false`, определяющее будут ли в возвращаемую данным методом строку добавляться символы переноса строки в местах, соответствующих концу одного текстового блока и началу другого. Если значением этого параметра является `true`, длина возвращаемой строки оказывается больше, чем длина исходного текста (за счет добавления символов разрыва строк). Этот параметр является необязательным. Если при вызове метода `getText()` данный параметр опущен, возвращаемая строка имеет такую же длину, как и исходный текст.

Проиллюстрируем действие этого метода на предыдущем примере с клипом `mc` и тремя статическими текстовыми полями, расположенными на его монтажной линейке.

```
var snapshot:TextSnapshot = mc.getTextSnapshot();
trace(snapshot.getText(0, snapshot.getCount(), false));
```

В результате выполнения этого кода в панель **Output** будет выведено следующее:

```
FlashMX2004
```

Как видно, содержимое всех текстовых полей конкатенировалось в одну строку без всяких разделителей.

Теперь попробуем передать методу `getText()` значение `true` в качестве последнего параметра.

```
trace(snapshot.getText(0, snapshot.getCount(), true));
```

В результате выполнения этого кода в панель **Output** будет выведен новый результат:

```
Flash
MX
2004
```

Поиск текстового фрагмента

Для выполнения поиска первого вхождения заданного набора символов в строку, хранящуюся в объекте `TextSnapshot`, используется метод `findText()`. Принцип действия этого метода очень похож на принцип работы метода `indexOf()` класса `String`, выполняющего поиск в заданной строке.

снимок.findText(*начальный индекс*, *текст*, *регистр*)

Параметр *начальный индекс* — задает индекс символа, с которого будет начат поиск.

Параметр *текст* — задает набор символов, который требуется найти. Этот параметр задается строковым литералом или выражением, возвращающим строку.

Параметр *регистр* принимает булево значение `true` или `false`, определяющее, должен ли поиск выполняться с учетом регистра (`true`) или нет (`false`).

Метод `findText()` — возвращает положительное целое число, соответствующе индексу первого вхождения искомого набора символов, или значение −1, если искомая подстрока не найдена.

Выделение статического текста цветом

Одной из наиболее важных возможностей, предоставляемых классом `TextSnapshot`, является возможность выделения статического текста заданным цветом. Такое выделение не имеет ничего общего с обычным выделением текста. Выделение цветом реализуется за счет заполнения ограничивающей рамки символа указанным цветом и, по сути, является просто окрашиванием его фона. При этом цвет самих выделенных символов не меняется (в отличие от обычного выделения).

Следует иметь в виду, что для реализации возможности выделения текста цветом и контроля этого выделения необходимо, чтобы в конечный документ была включена информация о метриках шрифтов, используемых в статических текстовых блоках, к которым должно применяться выделение. По умолчанию при работе со статическим текстом эта информация не включается в конечный документ. Документация к Flash рекомендует следующий выход из этой ситуации. Для внедрения в документ информации о метриках шрифтов необходимо создать в этом документе пустое динамическое поле, использующее тот же шрифт, что и статическое поле. Для этого динамического поля необходимо внедрить очертания хотя бы одного символа из используемого шрифта. В результате описание метрик шрифта будет включено в конечный документ. Причем для включения в документ этой информации не имеет значения, очертания какого именно символа будут внедрены. Таким символом может быть даже пробельный символ!

Примечание

Напомним, что каждое отдельное начертание шрифта требует встраивания метрик, не говоря уж об использовании в текстовых блоках разных шрифтов. Для обеспечения возможности цветового выделения статического текста необходимо встроить метрики каждого используемого шрифта. Это означает, что для каждого шрифта необходимо создать пустое динамическое текстовое поле и встроить хотя бы один символ этого шрифта.

Выделение цветом можно использовать в художественных целях (например, для создания визуальных акцентов в тексте), а также для организации выделения, действующего подобно стандартному текстовому выделению. Целесообразность организации такого псевдовыделения заключается в следующем.

- Текст может быть отформатирован с применением настроек, присущих только статическому невыделяемому тексту. К числу таких настроек относятся, например, трекинг (межбуквенный промежуток) и регистр (верхний и нижний). Динамический и пользовательский типы текста не поддерживают эти настройки. Поскольку выделяемый статический текст по сути является динамическим, использование этих настроек для него также исключается. Тем не менее при необходимости выделения текста, содержащего форматирование с применением этих настроек, можно воспользоваться возможностью организации выделения цветом.
- Можно устанавливать цвет выделения. Эта особенность позволяет добиться более целостного цветового решения.

Установка и снятие выделения

Для выполнения установки и снятия выделения статического (невыделяемого) текста используется метод `setSelected()`. Этот метод имеет следующий синтаксис:

снимок`.setSelected(`*начальный индекс*`, `*конечный индекс*`, `*установка/снятие*`)`

Здесь *снимок* — это объект `TextSnapshot`, содержащий текст из статических текстовых полей клипа.

Параметр *начальный индекс* — задает индекс начального символа, входящего в диапазон, который требуется выделить или с которого нужно снять выделение. Индексация символов в строке начинается с нуля. Допустимыми значениями являются целые числа в диапазоне от `0` до `getCount()-1`. Если значение этого параметра отрицательно, используется значение 0.

Параметр *конечный индекс* — задает индекс символа, следующего за последним символом диапазона, который требуется выделить или с которого нужно снять выделение. Если этот параметр не указан, используется значение, возвращаемое методом `getCount()`. Если значение этого параметра равно значению параметра *начальный индекс*, в качестве его значения используется *начальный индекс*`.+1`.

Параметр *установка/снятие* принимает булево значение `true` или `false` и определяет необходимость выполнения установки выделения (`true`) или снятия выделения (`false`).

Выделение цветом можно установить сразу для нескольких диапазонов текста, причем эти диапазоны не должны располагаться последовательно друг за другом.

Пусть на монтажной линейке клипа `mc` имеется статический текстовый блок, в котором находится текст "ActionScript". Следующий код выделяет сочетание "Action":

```
var snapshot:TextSnapshot = mc.getTextSnapshot();
snapshot.setSelected(0, 6, true);
```

Для того чтобы снять это выделение, можно воспользоваться следующим кодом:

```
snapshot.setSelected(0, 6, false);
```

Для того чтобы выделить все содержимое статического текстового блока, напишем:

```
snapshot.setSelected(0, snapshot.getCount(), true);
```

Установка цвета выделения

По умолчанию выделение цветом выполняется с применением желтого цвета. Задать цвет выделения можно с помощью метода `setSelectColor()`. Этот метод имеет следующий синтаксис:

снимок`.setSelectColor(`*цвет*`)`

Здесь *снимок* — это объект `TextSnapshot`, содержащий текст из статических текстовых полей клипа. Следует отметить, что цвет выделения задается сразу для всех статичных текстовых полей в клипе, то есть для всего содержимого объекта `TextSnapshot` используется одинаковый цвет выделения.

Параметр *цвет* — это шестнадцатеричное число, задающее цвет выделения. Цвет выделения не может содержать прозрачность.

Управление выделением

Как уже говорилось, выделение цветом функционирует совершенно не так, как обычное выделение. В связи с этим для реализации функциональности, позволяющей пользователю выделять статический текст с помощью мыши, необходимо контролировать выделение и управлять его установкой и снятием. Для этих целей в классе `TextSnapshot` имеется несколько встроенных методов.

Для определния того, имеется ли в заданном диапазоне хотя бы один выделенный символ, используется метод `getSelected()`. Метод имеет следующий синтаксис:

снимок.getSelected(*начальный индекс*, *конечный индекс*)

Параметр *начальный индекс* — это индекс первого символа диапазона, в котором выполняется поиск выделенных символов. Допустимыми значениями являются целые числа в диапазоне от 0 до `getCount()-1`. Если значение этого параметра отрицательно, используется значение 0.

Параметр *конечный индекс* — это индекс символа, следующего за последним символом диапазона, в котором выполняется поиск выделенных символов. Если этот параметр не указан, используется значение, возвращаемое методом `getCount()`. Если значение этого параметра меньше или равно значению параметра *начальный индекс*, в качестве его значения используется *начальный индекс*.+1.

Метод `getSelected()` возвращает булево значение `true`, если в заданном диапазоне имеется хотя бы один выделенный символ, и значение `false`, если в заданном диапазоне нет выделенных цветом символов.

Для извлечения выделенного текста используется метод `getSelectedText()`. Этот метод имеет следующий синтаксис:

снимок.getSelectedText(*включить переносы строк*)

Параметр *включить переносы строк* принимает булево значение и определяет, нужно ли включить в возвращаемую строку символы переноса строк в соответствии с разбиеним текста на строки в исходном текстовом блоке. Если данный параметр принимает значение `true`, строка, возвращаемая методом `getSelected()`, будет включать символы переноса строки. Если параметр *включить переносы строк* принимает значение `false`, из строки, возвращаемой методом `getSelected()`, будут удалены символы переноса строки. Этот параметр является необязательным, по умолчанию он принимает значение `false`.

Для реализации функциональности, позволяющей выполнять выделение текста с помощью мыши, необходимо контролировать местоположение курсора относительно символа текста. Для решения этой задачи используется метод `hitTestTextNearPos()`. Этот метод возвращает индекс ближайшего к заданной точке символа текста. Метод `hitTestTextNearPos()` имеет следующий синтаксис:

снимок.hitTestTextNearPos(*x*, *y*, *максимальное расстояние*)

Параметр *x* — это горизонтальная координата точки, заданная в системе координат клипа, содержащего статический текст.

Параметр *y* — это вертикальная координата точки, заданная в системе координат клипа, содержащего статический текст.

Параметр *максимальное растояние* — это положительное число, задающее максимальное расстояние, в пределах которого осуществляется поиск ближайшего к точке (*x*; *y*) текстового символа. Это расстояние измеряется от центра ограничивающей рамки текстового символа до точки с заданными координатами. Параметр *максимальное растояние* является необязательным. По умолчанию его значение составляет 0.

Метод возвращает индекс текстового символа, ближайшего к точке с координатами, заданными первыми двумя параметрами этого метода. Если в пределах заданного растояния символ обнаружить не удается, метод возвращает значение −1. Для того чтобы определить, над каким символом в данный момент находится курсор мыши, нужно в качестве первых двух параметров передать методу `hitTestTextNearPos()` значения `this._xmouse` и `this._ymouse`.

В качестве примера, иллюстрирующего возможности работы с классом `TextSnapshot`, рассмотрим организацию пользовательского выделения статического текста с помощью мыши. На монтажной линейке клипа `mc` создадим статический невыделяемый текстовый блок. Задача заключается в том, чтобы реализовать возможность выделения текста, как если бы режим **Selectable** был активен. Выделение должно выполняться при протаскивании курсора с нажатой кнопкой мыши над текстом. Причем, если курсор проходит над уже выделенным текстом, выделение должно сниматься. Щелчок мышью также должен снимать выделение с текста. Для большей наглядности реализуем возможность выделения нескольких несвязанных фрагментов текста при нажатии клавиши <Ctrl>. Сценарий, реализующий эту функциональность, снабжен подробными комментариями и представлен в листинге 14.18.

Листинг 14.18. Организация выделения и копирования статического текста с помощью мыши

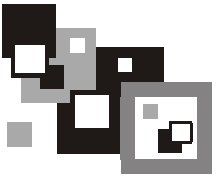
```
// Создаем объект TextSnapshot
var snapshot = mc.getTextSnapshot();
// Устанавливаем цвет выделения
snapshot.setSelectColor(0xD0D0D0);
// При нажатии левой кнопки мыши...
mc.onMouseDown = function() {
       /* Сохраняем индекс ближайшего к курсору символа в переменной
       currentIndex */
       currentIndex = snapshot.hitTestTextNearPos(this._xmouse,
this._ymouse, 1000);
       // Если клавиша <CTRL> не нажата...
       if (!Key.isDown(Key.CONTROL)) {
```

```
            // Снимаем выделение с текста
            snapshot.setSelected(0, snapshot.getCount(), false);
        }
        /* Выделение текста мышью вполняется только в том случае, если
        курсор находится достаточно близко к символу. Это проверяется
        следующим условием: */
        if (currentIndex != -1) {
            // При перемещении курсора мыши (с нажатой левой кнопкой)
            this.onMouseMove = function() {
            /* В переменной prevIndex будем сохранять индекс предыдущего
            символа, над которым находился курсор. Поскольку при
            очередном нажатии мыши необходимо начать выделение нового
            диапазона, в качестве этого значения поместим в переменную
            значение currentIndex, а в дальнейшем при перемещении мыши
            вдоль диапазона символов будем сохранять в ней индекс
            предыдущего выделенного символа. Для этого используем
            условный оператор, присваивающий переменной prevIndex
            значение currentIndex, если ее значение не определено, и
            присваивающий ей в дальнейшем значение индекса предыдущего
            символа, над которым находился курсор (hitIndex) */
        prevIndex=(prevIndex==undefined ? currentIndex : hitIndex);
            /* Сохраняем индекс текущего символа, над которым находится
            курсор */
            hitIndex = snapshot.hitTestTextNearPos(this._xmouse,↳
            this._ymouse, 1000);
            /* Выделяем диапазон символов между символом, над которым
            была нажата кнопка мыши, и символом, над которым в данный
            момент находится курсор. Поскольку выделение может
            выполняться в любую сторону от начального символа
            (currentIndex), слева направо и справа налево, начальный
            и конечный индексы задаются в соответствии с направлением
            перемещения курсора */
            snapshot.setSelected(Math.min(currentIndex, hitIndex),
            Math.max(currentIndex, hitIndex), true);
            /* Если курсор с нажатой кнопкой мыши перемещается по уже
            выделенным символам, выделение с них снимается. */
            if (snapshot.getSelected(hitIndex, hitIndex+1)) {
                snapshot.setSelected(Math.min(prevIndex,hitIndex),↳
                Math.max(prevIndex, hitIndex), false);
                }
            };
        }
};
/* При отпускании левой кнопки мыши присваиваем переменной prevIndex
значение undefined с тем, чтобы при следующем нажатии мыши можно было
начать выделение нового диапазона символов. Прекращаем выделение */
```

```
mc.onMouseUp = function() {
        prevIndex = undefined;
        delete this.onMouseMove;
};
// -----СОЗДАНИЕ ЭЛЕМЕНТА КОНТЕКСТНОГО МЕНЮ ДЛЯ КОПИРОВАНИЯ В БУФЕР-----
my_CM = new ContextMenu();
my_CM.hideBuiltInItems();
var copy_CM = new ContextMenuItem("Копировать в буфер", copy);
my_CM.customItems.push(copy_CM);
function copy() {
        // Передаем строку, содержащую выделенные символы методу
        // setClipboard(), помещающего эту строку в буфер обмена.
        System.setClipboard(snapshot.getSelectedText(true));
}
mc.menu = my_CM;
```

Для того чтобы выделенный текст можно было скопировать в буфер, в приведенном сценарии реализуется создание пользовательской команды контекстного меню, которая появляется при щелчке правой кнопкой мыши на клипе `mc`. Подробная информация о настройке и добавлении новых команд в контекстное меню содержится в *главе 22*.

Копирование выделенного текста в буфер обмена выполняется с помощью статического метода `setClipboard()` класса `System`. Данный метод помещает переданную ему в качетве параметра строку текста в буфер обмена, замещая хранящееся там содержимое. Текст помещается в буфер как неформатированный текст. Метод `setClipboard()` возвращает значение `true`, если копирование в буфер выполнено успешно, и значение `false`, если разместить данные в буфере не удалось.



Глава 15

Класс *Math*. Выполнение математических вычислений

Работая над решением задачи, всегда полезно знать ответ.

Правило точности

Класс `Math` предоставляет набор математических констант и функций, которые используются при выполнении математических расчетов и вычислений. Все свойства и методы этого класса являются статическими, это означает, что для их использования не требуется создания экземпляра с применением конструктора. Обращение к встроенным константам и вызов методов класса `Math` осуществляются непосредственно с помощью объекта `Math`. При этом используется следующий синтаксис:

```
Math.свойство // Обращение к математической константе
Math.метод() // Вызов функции
```

Необходимо иметь в виду, что вычисление многих математических функций (в частности, тригонометрических) выполняется с определенной степенью точности. Это может привести к неожиданным результатам. Так, например, $\cos(\pi/2) = 0$, однако результат вычисления этой функции, полученный с помощью метода `Math.cos()`, будет являться лишь аппроксимацией значения функции косинуса в точке $\pi/2$ (в частности и из-за того, что $\pi/2$ — это иррациональное число). Вместо значения 0 будет получено значение 6.12303176911189e-17. В связи с этим для получения корректных результатов часто значение, возвращаемое методом объекта `Math`, необходимо округлить до нужного знака.

Встроенные свойства. Математические константы

Свойства класса `Math` хранят приближенные значения наиболее часто используемых математических констант. Все свойства доступны только для чтения. Полный перечень математических констант, хранящихся в классе `Math`, приведен в табл. 15.1.

Таблица 15.1. *Свойства класса* `Math`*. Математические константы*

Свойство	Описание
`Math.E`	Число *e*, постоянная Эйлера, основание натуральных логарифмов (приблизительно 2,718)
`Math.LN2`	Натуральный логарифм 2 (приблизительно 0,693)
`Math.LN10`	Натуральный логарифм 10 (приблизительно 2,302)
`Math.LOG10E`	Десятичный логарифм числа *e* (приблизительно 0,434)
`Math.LOG2E`	Логарифм числа *e* по основанию 2 (приблизительно 1,442)
`Math.PI`	Число π, отношение длины окружности к ее диаметру (приблизительно 3,14159)
`Math.SQRT1_2`	Квадратный корень из $^1/_2$ (приблизительно 0,707)
`Math.SQRT2`	Квадратный корень из 2 (приблизительно 1,414)

Возведение числа в степень и извлечение корня

Для возведения числа в заданную степень используется метод `Math.pow()`. Данный метод имеет следующий синтаксис:

```
Math.pow(основание, показатель)
```

Параметр *основание* — это число, которое необходимо возвести в степень. Данный параметр может принимать любые числовые значения, положительные и отрицательные, целые и дробные.

Параметр *показатель* — это показатель степени. Данный параметр может принимать любые числовые значения, положительные и отрицательные, целые и дробные.

Напомним два определения из элементарной математики, которые могут оказаться полезны при использовании данного метода.

- Степень числа с отрицательным показателем — это единица, деленная на степень того же числа с положительным показателем, величина которого равна абсолютной величине отрицательного показателя:

$$a^{-m} = \frac{1}{a^m}.$$

- Возвести действительное число в степень *m*/*n* — означает извлечь корень *n*-ой степени из *m*-ой степени этого числа.

$$a^{\frac{m}{n}} = \sqrt[n]{a^m}.$$

Как следует из приведенных определений, с помощью метода `Math.pow()` можно не только возводить число в степень, но и извлекать из него корни заданной степени.

Рассмотрим несколько примеров:

```
Math.pow(2, 3); // Возвращает 8 = 2^3
Math.pow(2, -3); // Возвращает 0.125 = 1/2^3 = 2^-3
Math.pow(8, 1/3); // Возвращает 2 = 8^1/3 = ∛8
Math.pow(8, -1/3); // Возвращает 0.5 = 1/8^1/3 = 8^-1/3
```

Необходимо отметить, что если показатель степени задается дробным числом, параметр *основание* должен быть неотрицательным. В противном случае вместо вычисления результата метод `Math.pow()` возвращает значение `NaN`. Это связано с тем, что дробная степень отрицательного основания может не быть действительным числом. Flash не выполняет проверку, определяющую, является ли результат возведения в степень действительным числом. Так, например:

$\sqrt[3]{-8} = -2$ — действительное число;

$\sqrt[4]{-8}$ — недействительное число.

```
Math.pow(-8, 1/3); // Возвращает NaN
Math.pow(-8, 1/4); // Возвращает NaN
```

Несмотря на то, что вычисление квадратного корня можно выполнить с помощью метода `Math.pow()`, в классе `Math` для выполнения этой операции имеется отдельный метод `Math.sqrt()`. Метод `Math.sqrt()` в качестве параметра принимает целое неотрицательное число и возвращает приближенное значение его квадратного корня.

Ниже приводится функция, вычисляющая расстояние между двумя точками по теореме Пифагора:

```
function distance(x1, y1, x2, y2):Number {
       return Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}
```

В классе `Math` также определен метод `Math.exp()`, вычисляющий значение экспоненциальной показательной функции $y = e^x$, здесь e — это число Эйлера, приблизительно равное 2,718. В качестве параметра метод `Math.exp()` принимает значение показателя степени, в которую возводится число e. Вместо метода `Math.exp()` можно использовать метод `Math.pow()`, передавая ему в качестве первого параметра число e:

```
Math.pow(Math.E, 2) // Возвращает ≈7,389 = e^2
```

Аналогичный результат можно получить с помощью метода `Math.exp()`.

```
Math.exp(2); // Возвращает ≈7,389 = e^2
```

Вычисление логарифма числа

Логарифмом числа N по основанию a называется показатель степени x, в которую нужно возвести a, чтобы получить число N. Числа a и N могут быть целыми и дробными, но всегда должны быть положительными (если логарифм должен быть действительным числом). Логарифм записывается следующим образом:

$$\log_a N = x.$$

Эта запись эквивалентна следующей:

$$a^x = N.$$

В классе `Math` имеется только один метод, позволяющий вычислить натуральный логарифм числа. Этот метод называется `Math.log()`. Натуральный логарифм — это логарифм по основанию *e*. Натуральный логарифм показывает, в какую степень *x* нужно возвести число *e*, чтобы получить *N*.

```
Math.log(1) // Возвращает 0, так как e^0 = 1;
Math.log(Math.E) // Возвращает 1, так как e^1 = e;
```

В соответствии с определением логарифма метод `Math.log()` принимает в качестве параметра только положительное число.

Для того чтобы вычислить логарифм числа *N* по произвольному основанию, зная натуральный логарифм этого числа, можно воспользоваться следующей формулой:

$$\log_a N = \frac{\log_e N}{\log_e a}.$$

Ниже приводится функция `logN()`, вычисляющая логарифм числа *N* по заданному основанию.

```
function logN(base:Number, N:Number):Number{
       return Math.log(N)/Math.log(base);
}
```

Так, чтобы найти логарифм числа 25 по основанию 5, вызовем функцию `logN()` со следующими параметрами.

```
logN(5, 25) // Возвращает 2
```

Определение наибольшего и наименьшего значений

Довольно часто на практике возникает задача, связанная с определением наибольшего или наименьшего из набора имеющихся значений. С такой задачей можно столкнуться, например, при сортировке величин. В классе `Math` имеется два метода `Math.max()` и `Math.min()`, возвращающих, соответственно, максимальное и минимальное из двух переданных им в качестве параметров значений.

```
Math.max(5, 10); // Возвращает 10
Math.min(10, -5); // Возвращает -5
```

Методы `Math.max()` и `Math.min()` принимают только два параметра, в то время как на практике иногда необходимо определить наибольшее или наименьшее из более чем двух значений. В листинге 15.1 приводятся функции `max()` и `min()`, принимающие любое количество параметров и возвращающие, соответственно, наибольшее и наименьшее из их значений. Принцип действия этих функций основан на том, что значения их параметров попарно последовательно сравниваются и резуль-

тат сравнения сохраняется в локальных переменных. В качестве начального значения этих переменных берется максимальное и минимальное представимые числа. Таким образом, вначале первый параметр сравнивается с этим значением и большее (меньшее) из них сохраняется в переменной, затем второй параметр сравнивается со значением переменной и так до тех пор, пока не будут проверены все параметры. Поскольку количество фактических параметров заранее не известно, для обращения к ним из тела функции используется массив `arguments`, хранящий значения всех фактических параметров, переданных функции при вызове.

Листинг 15.1. Определение наибольшего и наименьшего из набора значений

```
// Функция, возвращающая наибольшее из переданных ей значений
function max():Number {
// Локальная переменная для хранения текущего результата сравнения
        var max:Number = Number.MIN_VALUE;
        for (var i = 0; i < arguments.length; i++) {
                max = Math.max(max, arguments[i]);
        }
        return max;
}
// Функция, возвращающая наименьшее из переданных ей значений
function min():Number {
// Локальная переменная для хранения текущего результата сравнения
        var min:Number = Number.MAX_VALUE;
        for (var i = 0; i < arguments.length; i++) {
                min = Math.min(min, arguments[i]);
        }
        return min;
}
```

Определение модуля числа

Модулем действительного числа является его абсолютная величина. Абсолютная величина неотрицательного числа совпадает с самим числом, а абсолютная величина отрицательного числа представляет собой положительное число, полученное путем перемены его знака (с минуса на плюс).

Для получения модуля числа используется метод `Math.abs()`. Данный метод принимает в качестве параметра число и возвращает его абсолютную величину.

```
Math.abs(-3); // Возвращает 3
Math.abs(.5); // Возвращает .5
```

Округление значений

В классе Math имеются три метода, позволяющие выполнять округление значения. Каждый из этих методов выполняет округление значения до целого числа:

- Math.round() — округление до ближайшего целого;
- Math.ceil() — округление до большего целого;
- Math.floor() — округление до меньшего целого.

Математическое округление

Метод Math.round() округляет значение, переданное ему в качестве параметра, до целого в соответствии с правилами математического округления.

Для неотрицательных чисел имеет место следующее правило. Если дробная часть числа больше или равна 0,5, округление выполняется до большего целого, если дробная часть числа меньше 0,5, округление выполняется до меньшего целого.

Для отрицательных чисел используется следующее правило. Если дробная часть числа меньше или равна 0,5, округление выполняется до большего целого, если дробная часть числа больше 0,5, округление выполняется до меньшего целого.

```
Math.round(3,1415926535); // Возвращает 3
Math.round(-2.345); // Возвращает -2
Math.round(-2.645); // Возвращает -3
Math.round(2.5); // Возвращает 3
Math.round(-2.5); // Возвращает -2, так как -2 больше, чем -3
```

Довольно часто приходится округлять числа не до целого, а до определенной значащей цифры. В этом случае можно предварительно умножить округляемое число на степень числа 10, соответствующую количеству значащих цифр. После этого такое число можно округлить до целого, а затем полученный результат разделить на ту же самую степень числа 10. Ниже приводится функция roundTo(), выполняющая округление числа (num) до указанного знака (order) в соответствии с этим алгоритмом.

```
function roundTo(num:Number, order:Number):Number {
return Math.round(num * Math.pow(10, order)) / Math.pow(10, order);
}
```

Испытаем эту функцию в действии:

```
roundTo(0.123456, 4); // Возвращает 0.1235
```

Метод Math.round() может оказаться полезным при необходимости проверить, является ли заданное число целым. Для проведения такой проверки необходимо сравнить само число с его округленным до целого значением. Если условие выполняется — число целое.

Ниже приводится функция `isInteger()`, выполняющая проверку того, является ее параметр целым числом или нет.

```
function isInteger(val:Number):Boolean {
        if (val == Math.round(val)) {
                return true;
        }
        else {
                return false;
        }
}
```

Проверим работу этой функции:

```
isInteger(2.33); // Возвращает false
isInteger(5); // Возвращает true
```

Округление до большего целого

Метод `Math.ceil()` (от англ. *ceiling* — потолок) округляет свой аргумент до ближайшего большего целого. При округлении значения отбрасываемых цифр (дробной части) не оказывают влияния на результат.

```
Math.ceil(2.34); // Возвращает 3
Math.ceil(-3.12); // Возвращает -3
Math.ceil(-3.95); // Возвращает -3
Math.ceil(5); // Возвращает 5
```

Округление до меньшего целого

Метод `Math.floor()` (от англ. *floor* — пол) округляет свой аргумент до ближайшего меньшего целого. При округлении значения отбрасываемых цифр (дробной части) не оказывают влияния на результат.

```
Math.floor(2.34); // Возвращает 2
Math.floor(-3.12); // Возвращает -4
Math.floor(-3.95); // Возвращает -4
Math.ceil(5); // Возвращает 5
```

Тригонометрические функции

Тригонометрические функции находят широкое применение при необходимости проведения геометрических вычислений и моделировании периодических процессов.

Перечень методов, предоставляемых классом `Math` для вычисления значений тригонометрических функций, приведен в табл. 15.2.

Таблица 15.2. *Методы класса* `Math`*. Тригонометрические функции*

Метод	Описание
`Math.cos()`	Вычисляет косинус угла
`Math.sin()`	Вычисляет синус угла
`Math.tan()`	Вычисляет тангенс угла
`Math.acos()`	Вычисляет арккосинус своего параметра
`Math.asin()`	Вычисляет арксинус своего параметра
`Math.atan()`	Вычисляет арктангенс своего параметра
`Math.atan2()`	Вычисляет угол между отрезком, проведенным из начала координат в точку с заданными координатами, и осью абсцисс

Методы `Math.cos()`, `Math.sin()` и `Math.tan()` в качестве параметра принимают угол в радианах. Для перевода градусной меры угла в радианную и наоборот применяются следующие соотношения:

$$\alpha_{рад} = \alpha_{град} \cdot \pi/180;$$

$$\alpha_{град} = \alpha_{рад} \cdot 180/\pi,$$

где $\alpha_{рад}$ — это угол в радианах, а $\alpha_{град}$ — угол в градусах.

Тригонометрические функции имеют чрезвычайно широкую область применения. Мы еще неоднократно на страницах этой книги будем прибегать к их использованию, здесь же приведем пример, наглядно демонстрирующий возможности тригонометрических функций. Поскольку тригонометрические функции периодические — их можно успешно применять для реализации периодического движения или движения по замкнутой траектории. Для решения этой задачи необходимо задать закон периодического изменения значений свойств `_x` и `_y` объекта. Если при перемещении объекта периодически выполнять его дублирование, будет создан так называемый эффект шлейфа. Сам объект будет перемещаться по заданной траектории, при этом оставляя шлейф из своих неподвижных дубликатов. Для описания траектории движения объекта возьмем кривую Гвидо Гранди, называемую "розой". Уравнение семейства этих кривых в полярных координатах имеет следующий вид:

$$\rho = \alpha \cdot \sin k\varphi$$

Здесь

α и k — это постоянные, которые задаются положительными числами. Коэффициент α определяет диаметр кривой, а коэффициент k определяет число лепестков "розы". На рис. 15.1 приведены изображения, полученные на основе данного уравнения при разных значениях коэффициента k. Для кривой Гвидо Гранди имеют место следующие закономерности:

- если k — целое число, "роза" состоит из k лепестков при k нечетном и из $2k$ лепестков при k четном;

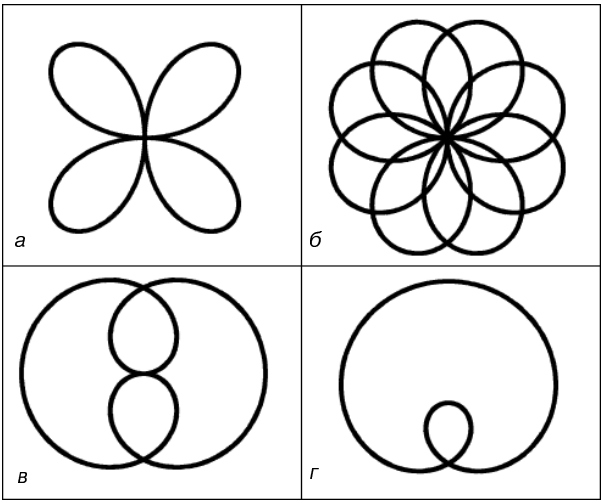


Рис. 15.1. Кривые Гвидо Гранди, "розы".
а — $k = 2$; *б* — $k = 4/3$; *в* — $k = 1/2$; *г* — $k = 1/3$

- если $k = m/n$ (при $n > 1$) — рациональное число, то "роза" состоит из m лепестков, если m и n — нечетные; и из $2m$ лепестков, если одно из этих чисел четное, причем следующий лепесток частично покрывает предыдущий;
- если k — целое число, то для получения полного графика "розы" полярный угол φ изменяется в диапазоне от 0 до π, если k — нечетное, и в диапазоне от 0 до 2π, если k — четное;
- если $k = m/n$ (при $n > 1$) — рациональное число, то для получения полного графика "розы" полярный угол φ изменяется в диапазоне от 0 до $2n\pi$, если m или n четное и в диапазоне от 0 до $n\pi$, если и числитель, и знаменатель нечетные.

Для реализации поставленной задачи, необходимо преобразовать полярные координаты в декартовы и получить параметрическое уравнение кривой. Напомним следующие формулы преобразования:

$$x = \rho\cos\varphi;$$

$$y = \rho\sin\varphi.$$

Исходя из этого, приведенное выше уравнение кривой в параметрическом виде можно записать следующим образом:

$$x = a\sin k\varphi\cdot\cos\varphi;$$

$$y = \sin k\varphi\cdot\sin\varphi.$$

Итак, закон изменения координат найден. Теперь необходимо осуществлять периодическое приращение угла φ и одновременно с этим выполнять дублирование перемещающегося клипа. После того как график "розы" будет полностью сформирован, нужно прекратить дальнейшее приращение значения угла (или можно было бы снова выполнять его приращение, начиная с 0).

В листинге 15.2 приведена функция `rose()`, выполняющая перемещение клипа по траектории, заданной уравнением "розы" с формированием шлейфа из его дубликатов. Приведенная функция принимает следующие параметры:

- `obj` — ссылка на клип, который должен перемещаться, формируя шлейф;
- `x` и `y` — начальные координаты клипа (это координаты центра "розы" за исключением случая, когда $k = 1/3$);
- `amp` — коэффициент α, отвечающий за диаметр "розы";
- `k` — коэффициент k, отвечающий за количество лепестков "розы". Если k — целое число, то этот параметр задается числовым литералом (или выражением, возвращающим число). Если k — дробное рациональное число, этот параметр задается массивом, состоящим из двух элементов. Первый элемент массива — числитель дроби, второй — ее знаменатель;
- `inc` — шаг приращения угла, отвечающий за скорость движения и число дубликатов. Чем меньше значение этого параметра, тем медленнее движение, выше "деталировка" графика кривой и, соответственно, больше нагрузка на процессор. Поскольку угол задается не в градусах, а в радианах, для получения приемлемых результатов значение этого параметра должно задаваться в диапазоне от 0,01 до 0,09 (читатель сам может поэкспериментировать).

Листинг 15.2. "Розы" Гвидо Гранди. Эффект перемещения с формированием шлейфа

```
function rose(obj:MovieClip, x:Number, y:Number, amp:Number, k,↵
inc:Number) {
        var thi = 0;
        // Если k целое четное
        if (k % 2 == 0) {
                var thiMax = 2 * Math.PI;
        }
        // Если k целое нечетное
        else if (k == Math.round(k)) {
                var thiMax = Math.PI;
        }
        else {
        /* Если k - рациональное число, которое задается обыкновенной
        дробью */
                // Если у дроби числитель или знаменатель четные
                if (k[0] % 2 == 0 || k[1] % 2 == 0){
                        var thiMax = 2 * k[1] * Math.PI;
                        k = k[0]/k[1];
                }
```

```
            // Если и числитель и знаменатель нечетные
            else {
                var thiMax = k[1] * Math.PI;
                k = k[0]/k[1];
            }
        }
        obj.onEnterFrame = function() {
            // Перемещение порождающего клипа
            this._x = amp * Math.sin(k * thi) * Math.cos(thi) + x;
            this._y = amp * Math.sin(k * thi) * Math.sin(thi) + y;
            // Процесс дублирования
            this.duplicateMovieClip("clip" +↵
            this._parent.getNextHighestDepth(),↵
            this._parent.getNextHighestDepth());
            // Поворачиваем порождающий клип
            this._rotation = thi / Math.PI * 180;
            // проверка на достижение максимального значения угла
            if (thi > thiMax) {
                delete this.onEnterFrame;
            }
            // приращение угла
            thi += inc;
        };
}
```

Используя тот факт, что монтажная линейка дубликата всегда начинает воспроизводиться с первого кадра вне зависимости от того, в каком кадре порождающего клипа находилась воспроизводящая головка, можно добиться довольно впечатляющего эффекта, если поместить на монтажную линейку клипа анимацию. Так, если на монтажной линейке клипа `obj` создать анимацию цветового перехода, сконструированная из его анимированных дубликатов "роза" будет переливаться разными цветами. Результат применения подобного приема представлен на рис. 15.2. Для формирования этого изображения функция `rose()` была вызвана со следующими параметрами:

```
rose(clip, 275, 200, 100, [3,4], 0.05);
```

Методы `Math.acos()`, `Math.asin()` и `Math.atan()` используются для вычисления обратных тригонометрических функций. Поскольку обратные тригонометрические функции являются многозначными, обычно исследование этих функций проводится в ограниченном интервале значений. Значения функции, расположенные в пределах этого интервала, называются *главными*. Методы `Math.acos()`, `Math.asin()` и `Math.atan()` возвращают только значения, расположенные на интервале главных

значений соответствующих функций. Для функции арккосинуса главными являются значения [0; π]. Для функции арксинуса главными являются значения [-π/2; π/2]. Для функции арктангенса главными являются значения (-π/2; π/2), причем график этой функции асимптотически приближается к значениям -π/2 и π/2, не достигая их.

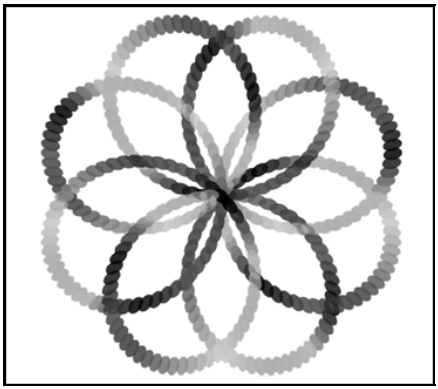


Рис. 15.2. Формирование шлейфа путем дублирования анимированного клипа

Метод `Math.atan2()` возвращает угол между отрезком, проведенным из начала координат в точку с координатами, заданными в качестве параметров, и осью абсцисс. Данный метод имеет следующий синтаксис:

```
Math.atan2(y, x);
```

Параметр `y` задает расстояние по вертикали от начала координат до точки, расположенной на конце отрезка, причем ось ординат считается направленной сверху вниз.

Параметр `x` задает расстояние по горизонтали от начала координат до точки, расположенной на конце отрезка, причем ось абсцисс считается направленной слева направо.

Метод `Math.atan2()` возвращает угол, на котором расположена точка с координатами (*x*, *y*) по отношению к положительной полуоси абсцисс. Причем возвращаемый угол отсчитывается в радианах по часовой стрелке[1]. По сути, этот метод возвращает значение арктангенса отношения противолежащего искомому углу катета к прилежащему катету, но не в диапазоне главных значений, а в диапазоне (-π, π). Графическое пояснение сказанного приведено на рис. 15.3.

[1] Возвращаемое методом `Math.atan2()` значение может быть интерпретировано исходя из более традиционной ориентации осей, используемой в математике. Так, если ось ординат направлена снизу вверх (ось абсцисс по-прежнему направлена слева направо), то за положительное направление отсчета углов, возвращаемых данным методом, следует принять направление против часовой стрелки. Здесь и далее мы будем исходить из расположения осей и направления отсчета углов, принятых в Flash. — *Прим. автора*.

Исходя из определения тангенса (тангенс — это отношение противолежащего катета к прилежащему) можно записать:

$$\mathrm{tg}\alpha = y / x;$$

отсюда следует, что угол α определяется следующим образом:

$$\alpha = \arctan(y / x).$$

Именно этот геометрический смысл и отражает метод `Math.atan2()`. Обратите внимание, что параметры этого метода намеренно указываются, начиная с *y*, поскольку это больше соответствует определению тангенса.

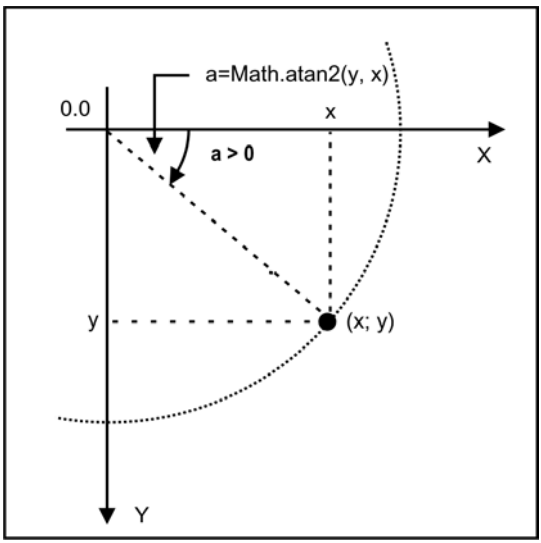


Рис. 15.3. Геометрическая интерпретация назначения метода `Math.atan2()`

С помощью данного метода можно определить угол, на котором находится произвольная точка по отношению к оси *X*. Причем даже если этот угол составляет 90° или −90°, то есть $\pi/2$ или $-\pi/2$ радиан (точка лежит на оси ординат), метод будет работать корректно, возвращая соответствующее значение.

Для поворота объекта в направлении курсора необходимо присвоить его свойству `_rotation` угол, полученный в результате вызова метода `Math.atan2()` со следующими параметрами:

```
Math.atan2((Yмыши - Yобъекта) / (Xмыши - Xобъекта))
```

Здесь `Yмыши` и `Xмыши` — координаты курсора мыши, `Yобъекта` и `Xобъекта` — координаты объекта. Необходимо помнить о том, что координаты мыши и координаты объекта должны быть указаны в одной и той же системе координат. Угол, возвращаемый методом, измеряется в радианах, в то время как свойство принимает значение в градусах. В связи с этим перед поворотом объекта необходимо выполнить пе-

ревод угла из радиан в градусы. В листинге 15.3 приводится пример сценария, заставляющего клип mc поворачиваться в направлении курсора.

Примечание

Важно иметь в виду, что при необходимости поворачивать объект вслед за курсором мыши его начальный угол поворота (установленный в рабочей среде) определяет то положение, в котором объект должен находиться по отношению к курсору. Другими словами, положение объекта, заданное при его создании в рабочей среде, соответствует нулевому углу поворота.

Листинг 15.3. Поворот объекта в направлении курсора

```
mc.onMouseMove = function() {
        /*Сохраняем расстояние между курсором и клипом по осям X и Y
        var x = this._parent._xmouse - this._x;
        var y = this._parent._ymouse - this._y;
        /*Вычисляем угол между линией, соединяющей точку регистрации
        клипа с курсором, и осью X, и переводим его в градусы */
        var angle = Math.atan2(y, x) / Math.PI * 180;
        //Поворачиваем объект вслед за курсором
        this._rotation = angle;
        updateAfterEvent();
};
```

Примеры использования данного метода также можно найти в листинге 17.1 и в листинге 17.5.

Применение генератора случайных чисел

Одной из отличительных характеристик программной анимации является возможность создания нерегулярных эффектов, то есть эффектов, в которых присутствует элемент случайности. Так, например, анимация, созданная с помощью монтажной линейки, всегда выполняется одинаково, объекты движутся по заранее заданной траектории в заранее заданных координатах с определенными цветами и т. д. Однако с применением программирования можно заставить объекты вести себя непредсказуемым образом. Они могут начинать движение из координат, определенных случайным образом, по траекториям, заданным с применением элемента случайности, и т. д. Во многих ситуациях для реализации естественного и живого движения такое поведение оказывается необходимым.

Подобные задачи решаются с помощью генератора случайных чисел. Генератор случайных чисел представлен методом Math.random(). Данный метод не принимает параметров и возвращает случайное число в диапазоне от 0 до 1, причем 1 не входит в этот диапазон. Для того чтобы получить случайное число в диапазоне от 0 до *N*,

нужно умножить возвращаемое этим методом значение на *N*. Так, например следующий код динамически создает экземпляр клипа с идентификатором связи `clip` и помещает его на сцену в точку, координаты которой определяются случайным образом (предполагается, что размеры рабочей области составляют 550×400 пикселов):

```
_root.attachMovie("clip", "newClip", 1, {
                                        _x:Math.random()*550,
                                        _y:Math.random()*400
                                                           }
);
```

Для того чтобы получить случайное число в заданном диапазоне, нужно использовать следующее выражение:

```
Math.random()*(max - min) + min,
```

где `min` — нижняя граница диапазона; `max` — верхняя граница диапазона (само значение `max` не входит в диапазон).

Ниже приводится функция, возвращающая случайное число в заданном диапазоне:

```
function rand (min:Number, max:Number):Number{
       return Math.random()*(max - min) + min;
}
```

Эффект падающего снега

В качестве иллюстрации применения генератора случайных чисел приведем пример создания эффекта падающего снега. Существует множество подходов к решению этой задачи. Предлагаемый здесь способ реализует эффект снегопада с учетом следующих функциональных особенностей.

- Снежинки должны появляться у верхней границы рабочей области в точке со случайной координатой по *X*.
- Снежинки должны иметь случайный размер.
- Скорость падения снежинки должна зависеть от ее размера. Чем больше размер (масса), тем выше скорость падения.
- Снежинки должны перемещаться сверху вниз, описывая синусоиду.
- Амплитуда и длина периода траектории движения снежинки должны зависеть от ее размера. Чем больше размер, тем меньше амплитуда и тем длиннее период.
- Степень прозрачности снежинки должна зависеть от ее размера. Чем меньше (дальше от зрителя) снежинка, тем выше степень ее прозрачности.
- Точки регистрации снежинок не должны выходить за границы рабочей области.

Вначале рассмотрим закон перемещения снежинки. Для простоты предположим, что вертикальная координата снежинки изменяется по линейному закону, а гори-

зонтальная координата изменяется по закону свободных колебаний без сопротивления. В параметрическом виде это выглядит следующим образом:

$$y = speed \cdot t;$$

$$x = a\sin(\omega t + \varphi).$$

Здесь *speed* — коэффициент, определяющий скорость падения снежинки; t — время; a — амплитуда горизонтального смещения; ω — циклическая частота; φ — начальная фаза.

Скорость вертикального смещения i-ой снежинки определим следующим образом

$$speed_i = _scale_i/(random + p).$$

Здесь $_scale_i$ — масштаб i-ой снежинки; *random* — случайное число в некотором диапазоне; p — постоянное число. Такая зависимость позволяет, с одной стороны, учесть размер снежинки (чем больше снежинка, тем она быстрее падает), с другой стороны, случайное число в знаменателе вносит элемент случайности. Если снежинки имеют одинаковые или близкие размеры, скорость их падения может быть разной. Это позволяет получить более естественное движение. Чем больше соотношение *random/p*, тем больше элемент случайности и меньше зависимость скорости от размера снежинки.

Амплитуда горизонтального смещения должна быть обратно пропорциональна размеру снежинки, поэтому для задания амплитуды можно использовать простую линейную зависимость:

$$a_i = k/_scale_i,$$

где k — постоянный коэффициент. Поскольку масштаб каждой снежинки определяется случайным образом, каждая снежинка будет иметь свою амплитуду.

Циклическая частота — это параметр, который определяет длину периода. Чем выше значение циклической частоты, тем короче период и наоборот. В нашем случае длина периода должна быть обратно пропорциональна размеру снежинки, поэтому циклическую частоту можно выразить через масштаб:

$$\omega_i = 1/_scale_i.$$

Поскольку масштаб каждой снежинки определяется случайным образом, каждая снежинка будет иметь собственное значение циклической частоты. Это означает, что длина периода для каждой снежинки устанавливается индивидуально.

Начальная фаза позволяет задать сдвиг фазы для разных снежинок так, чтобы их перемещение начиналось с разных точек синусоиды. Значение этого параметра должно быть разным для разных снежинок. Для его расчета можно воспользоваться значением начальной горизонтальной координаты снежинки (которое определяется случайным образом):

$$\varphi_i = initX_i / w,$$

где $initX_i$ — это начальная горизонтальная координата i-ой снежинки; w — ширина рабочей области.

Таким образом, с увеличением значения t снежинки будут двигаться в соответствии с описанными выше закономерностями.

При создании снегопада используется следующий алгоритм.

1. Создаем требуемое количество клипов, представляющих снежинки.
2. Для каждой снежинки устанавливаем случайный масштаб (пропорционально по ширине и высоте), помещаем ее в случайные координаты. Исходя из масштаба снежинки, задаем для нее коэффициент скорости `speed`, начальное положение `initX` и прозрачность.
3. Реализуем перемещение каждой снежинки до тех пор, пока ее точка регистрации не выйдет за пределы границ рабочей области. Если это произошло, возвращаем снежинку наверх, задаем ей новое случайное положение по горизонтали и продолжаем перемещение.

Как мы видим, в соответствии с этим алгоритмом закон движения для каждой снежинки определяется один раз в момент ее создания. После этого данная снежинка будет перемещаться по одной и той же траектории, но каждый раз в разном положении по горизонтали. Это создаст впечатление хаотичного движения. Можно было бы, наоборот, размещать снежинки в неизменном положении по горизонтали, но менять их размер после каждого цикла падения.

Сценарий, реализующий эффект снегопада, приведен в листинге 15.4. Периодическое приращение времени *t* осуществляется с помощью обработчика события `onEnterFrame`. Здесь, вместо того чтобы определять метод обработчика для каждого экземпляра клипа, представляющего снежинку, этот метод был определен в прототипе конструктора `MovieClip`. При этом использовался следующий синтаксис:

```
MovieClip.prototype.onEnterFrame = function(){
...
}
```

Свойства, помещенные в прототип конструктора, наследуются всеми экземплярами, создаваемыми этим конструктором. Информация о прототипе и организации наследования приводится в *главе 25*. Определив метод обработчика события `onEnterFrame` в прототипе конструктора `MovieClip`, мы снабдили им все клипы, используемые в документе, без необходимости определять его индивидуально для каждого экземпляра. Однако для того чтобы поведением снежинок обладали только клипы, выполняющие эту роль, мы добавили в метод обработчика проверку имени экземпляра. Все инструкции в теле метода обработчика выполняются только для экземпляров, имена которых начинаются с букв "mc". Этим мы исключаем добавление поведения снежинок для других экземпляров клипов.

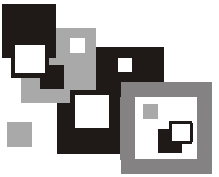
Листинг 15.4. Эффект падающего снега

```
// Переменная, хранящая текущее время
var t:Number = 0;
/* В прототипе конструктора MovieClip определяем метод обработчика
событий. Этот метод будет унаследован всеми экземплярами клипов */
MovieClip.prototype.onEnterFrame = function() {
        /* В снегопаде должны участвовать только клипы, представляющие
        снежинки, то есть клипы, имя которых будет начинаться с букв "mc"
        */
```

```
        if (this._name.substr(0, 2) == "mc") {
                // Горизонтальное смещение
                this._x = 2000 / this._xscale* Math.sin(t / this._xscale +↵
                this.initX/550) + this.initX;
                // Вертикальное смещение
                this._y += this.speed;
                // Приращение времени
                t += 0.1;
                // При выходе снежинки за пределы рабочей области
                if (this._y > 400 || this._x > 550 || this._x < 0) {
                        // Вновь помещаем снежинку наверх
                        this._y = this._height;
                        // Определяем новую случайную горизонтальную
                        // координату для снежинки
                        this.initX = Math.round(Math.random() * 550);
                        // Позиционируем снежинку по горизонтали
                        this._x = Math.round(2000 / this._xscale * ↵
                        Math.sin(t / this._xscale + this.initX/550) +↵
                        this.initX);
                }
        }
};
/* Создаем снежинки и размещаем их в случайных координатах по вертикали и
по горизонтали */
for (var i = 0; i < 80; i++) {
        this.attachMovie("mc", "mc" + i, i, {
                                        _x:Math.round(Math.random() * 550),
                                        _y:Math.round(Math.random() * 300)

        });
        // Задаем снежинкам случайные размеры
        this["mc" + i]._xscale = this["mc" + i]._yscale = ↵
        Math.round(Math.random() * 50 + 50);
        /* Для каждой снежинки определяем скорость ее падения, исходя из
        ее размера и добавляя элемент случайности */
        this["mc"+i].speed=this["mc"+i]._xscale / (Math.random()*5 + 10);
        /* Для каждой снежинки сохраняем ее начальную горизонтальную
        координату */
        this["mc" + i].initX = this["mc" + i]._x;
        /* Для каждой снежинки устанавливаем степень прозрачности, исходя
        из ее размера */
        this["mc" + i]._alpha = this["mc" + i]._xscale;
}
```

Глава 16



Класс *Date*. Контроль времени и даты

Нет лучше времени, чем настоящее, чтобы отложить то, чего не хочется делать.

Четвертый закон Хечта

Класс `Date` предоставляет широкие возможности работы с временем и датой. С его помощью можно получать информацию как о текущих времени и дате, так и об отдаленных моментах времени. В этой главе мы познакомимся с возможностями контроля даты и времени, а также рассмотрим примеры создания часов, таймера и календаря. Однако перед тем как перейти непосредственно к описанию класса `Date`, кратко рассмотрим понятия всеобщего скоординированного времени и машинного времени, поскольку они играют важную роль для понимания технологии измерения времени.

Всемирное скоординированное время и машинное время

До 70-х годов XX века общепринятым стандартом измерения времени являлось солнечное время. Измерение солнечного времени основывается на вращении Земли вокруг Солнца. При этом существует понятие *истинного* солнечного времени и *среднего* времени. Истинное солнечное время измеряется на основании реальных астрономических наблюдений Солнца. Среднее время, используемое в повседневной жизни, является приближением истинного солнечного времени, которое рассчитывается, исходя из равномерного перемещения Солнца. Между истинным и средним временем имеется некоторое расхождение, величина которого может достигать 16 минут. Наличие этого отклонения вызвано следующими причинами. Во-первых, орбита земли не является в точности круговой, и Земля перемещается по ней с несколько разной скоростью в зависимости от времени года. Во-вторых, поскольку плоскость орбиты Земли наклонена по отношению к плоскости ее экватора, ежедневный видимый маршрут солнца смещается в течение года с направления восток-запад. Это, соответственно, приводит к варьированию скорости изменения небесной широты Солнца, на основании которой отсчитывается солнечное время.

Очевидно, что в разных точках земного шара солнечное время различается. Появление понятия всемирного времени было вызвано необходимостью введения универсального времени, не зависящего от географического местоположения. Всемирным временем называется среднее солнечное время начального меридиана (GMT — Greenwich Mean Time). За начальный или нулевой меридиан принимается меридиан обсерватории в Гринвиче.

В середине XX века были созданы атомные часы, позволяющие с очень высокой степенью точности фиксировать промежутки времени. Принцип действия атомных часов основан на регулярности периодов излучения, испускаемого при квантовом переходе атома. Так, например, эталонная секунда определяется как 9 192 631 770 периодов излучения, отвечающего переходу между двумя сверхтонкими уровнями основного состояния атома цезия-133. Атомное время является более точным по сравнению с астрономическим. Атомная шкала времени (TAI) является результатом согласованной работы более 175 атомных часов, расположенных свыше, чем в 30 научных лабораториях и обсерваториях мира.

В 1972 году в качестве стандартной шкалы времени была принята шкала всемирного скоординированного времени (UTC). UTC координирует солнечное и атомное время. Солнечное и атомное время имеют некоторое расхождение, возникающее вследствие деятельности приливов и неравномерности сезонной скорости движения Земли. Величина этого расхождения может достигать 1 секунды в год. Для учета этой секунды (leap second) при возникновении соответствующей необходимости в последнюю минуту июня или декабря по UTC добавляется 61-я секунда (или, возможно, удаляется 60-я).

Теперь выясним, что же такое машинное время и как оно измеряется в ActionScript? В соответствии со стандартом ECMAScript время измеряется в миллисекундах, начиная с 1 января 1970 года по UTC. Таким образом, компьютер "не знает" месяца, дня или часа указанной даты, он "знает" только, сколько миллисекунд прошло с 0.00 часов 1 января 1970 года до указанной даты. Предшествующие даты задаются отрицательным числом, более поздним датам соответствует положительное число миллисекунд. Поскольку сам компьютер не может получить информацию о времени извне, активизация системных часов выполняется с помощью интерфейса оперативной системы. Для работы с временем и датой Flash обращается к системным часам. Следует отметить, что при измерении времени в ECMAScript не предусматривается учет расхождения солнечного и атомного времени, то есть предполагается, что каждые сутки состоят из 86 400 000 миллисекунд. Учитывая тот факт, что сутки всегда состоят из одинакового количества миллисекунд, если известно, сколько миллисекунд прошло с 1 января 1970 года до определенного момента, можно путем не очень сложных арифметических расчетов определить его дату и время. Класс `Date` призван избавить разработчика от необходимости самостоятельной реализации подобных расчетов, поскольку в него включен набор всех необходимых методов для определения и задания дат. Диапазон дат, которые могут быть представлены классом `Date`, составляет 100 000 000 дней до и 100 000 000 дней после 1 января 1970 года.

При работе с классом `Date` под UTC понимается скоординированное время, измеренное на нулевом (Гринвичском) меридиане (GMT). Для получения информации

или задания местного времени (учитывающего географическое положение) используются настройки оперативной системы, хранящей информацию о часовом поясе.

Создание экземпляра класса *Date*

Экземпляр класса `Date` — представляет конкретный момент времени, который реализуется как количество миллисекунд, прошедших с полуночи 1 января 1970 года до заданного момента. Все методы, вызванные этим экземпляром, осуществляют операции с представляемой им датой. Экземпляр класса создается с помощью оператора `new` и конструктора `Date()`. Существует несколько различных вариантов создания экземпляра с помощью конструктора.

Вызов конструктора без параметров

Первый способ создает экземпляр класса `Date`, представляющий текущий момент времени в местном времени. В этом случае конструктор `Date()` не принимает параметров:

```
var d:Date = new Date();
```

Ссылка на объект `Date`, созданный конструктором, сохраняется в переменной `d`. В данном случае объект содержит информацию о текущих времени и дате, то есть информацию, предоставляемую часами операционной системы.

Интересной особенностью объекта `Date` является то, что в результате его преобразования в строку получается строка, содержащая полную информацию о дате и времени, хранимых в данном объекте. В этом легко убедиться, если в панель **Output** вывести переменную `d`, хранящую ссылку на созданный выше объект `Date`.

```
trace(d);
```

В результате в панели **Output** будет выведено:

```
Thu Nov 4 16:45:48 GMT+0300 2004
```

Обозначение GMT+0300 означает, что местное время отличается от всемирного (гринвичского) на +3 часа.

Для того чтобы сохранить строковое представление объекта `Date`, можно воспользоваться методом `toString()` или глобальной функцией `String()`.

```
var d:Date = new Date();
var dateStr:String = String(d);
trace (dateStr); // Выводит Thu Nov 4 16:47:53 GMT+0300 2004
```

Вызов конструктора с передачей ему машинного времени

Для того чтобы создать экземпляр класса `Date`, представляющий конкретный указанный момент времени по местному времени, можно при вызове передать ему

число, указывающее количество миллисекунд, отделяющее этот момент от полуночи 1 января 1970 года.

```
var d:Date = Date(31536000000);
trace(d); // Выводит Fri Jan 1 03:00:00 GMT+0300 1971
```

Здесь конструктору в качестве параметра передано количество секунд в обычном (не високосном) году. Поскольку 1970 год не был високосным, созданный нами объект хранит дату первого дня следующего 1971 года по местному времени (поэтому время не 0 часов, а 3 часа утра). Аналогично можно было бы создать объект `Date` для хранения даты первого дня предыдущего 1969 года:

```
var d0:Date = Date(-31536000000);
trace(d0); // Выводит Wed Jan 1 03:00:00 GMT+0300 1969
```

Вызов конструктора с передачей ему параметров в формате даты

Для того чтобы создать экземпляр класса `Date`, представляющий конкретный указанный момент времени по местному времени, можно при вызове передать ему параметры, задающие год, месяц, число и время этого момента. При этом используется следующий синтаксис вызова конструктора:

```
new Date(year, month, date, hour, minute, second, millisecond)
```

Назначение параметров и диапазон принимаемых ими значений представлены в табл. 16.1.

Таблица 16.1. *Параметры конструктора* `Date()`

Параметр	Описание	Диапазон значений
`year`	Год	0—99; любое положительное четырехзначное число; отрицательное число. Если этот параметр задан двузначным числом, то год задается как сумма этого числа и 1900. Если этот параметр задан отрицательным числом, то год задается как сумма 1900 и этого числа
`month`	Месяц	от 0 до 11
`date`	Число	от 1 до 31
`hour`	Час	от 0 до 23
`minute`	Минута	от 0 до 59
`second`	Секунда	от 0 до 59
`millisecond`	Миллисекунда	от 0 до 999

Так, например, для того чтобы создать объект для 17.00 часов 26 августа 1979 года, можно вызвать конструктор, передав ему следующие параметры:

```
var d:Date = new Date(1979, 7, 26, 17, 0, 0);
```

Того же эффекта можно добиться, передав конструктору в качестве параметра количество миллисекунд, отделяющих указанную дату от 1 января 1970 года:

```
d1 = new Date(304520400000);
trace(d1); // Выводит Sun Aug 26 17:00:00 GMT+0400 1979
```

Следует отметить, что обязательными параметрами являются только первые два: год и месяц. Отсутствующие параметры принимают минимально допустимые значения. Так, например, вызов конструктора со следующими параметрами создаст объект `Date` для хранения момента времени, соответствующего 0 часов 22 апреля 1965 года:

```
var d2:Date = new Date(1965, 3, 22);
trace(d2);// Выводит Thu Apr 22 00:00:00 GMT+0400 1965
```

Еще раз отметим, что при создании объекта `Date` представляемое им время указывается в соответствии с местным временем, а не по UTC. Для того чтобы установить время по UTC, используются соответствующие методы класса `Date`.

Определение и установка параметров даты

Подавляющее большинство методов класса `Date` используется для определения или установки компонентов даты, представляемой объектом `Date`. Дата может определяться или устанавливаться по местному времени (то есть с учетом географического местоположения, определяющего принадлежность к часовому поясу) или по всемирному времени UTC (соответствующему времени на гринвичском меридиане).

Примечание

Следует иметь в виду, что при работе с методами, оперирующими с местным временем, возможно получение неточных сведений о датах, отстоящих от текущей. Это связано с тем, что Flash не учитывает в расчетах переходы на зимнее и летнее время. Однако сведения о текущем моменте времени выдаются абсолютно корректно, поскольку информация о переходе на зимнее или летнее время берется из настроек операционной системы.

Класс `Date` включает метод, позволяющий определить сдвиг местного времени по отношению ко всемирному (UTC). Этот метод называется `getTimezoneOffset()` и имеет следующий синтаксис:

```
дата.getTimezoneOffset()
```

где *дата* — это объект `Date`, соответствующий определенному моменту местного времени.

Данный метод возвращает разницу во времени, исчисляемую в минутах, между всемирным и местным временем. Причем результат вычисляется как разность UTC — — местное время.

```
trace(daysInMonth(2004, 1));
var d1:Date = new Date();
trace(d1.getTimezoneOffset()) /* Выводит - 180, часовой пояс
Санкт-Петербурга */
```

Определение местного времени и даты

Методы, относящиеся к этой категории, используются для получения информации о компонентах даты, представленной объектом `Date`. Возвращаемые этими методами результаты вычисляются на основании местного времени. Все методы, представленные в этом разделе (табл. 16.2) не принимают параметров и имеют схожий синтаксис:

дата.метод (),

где *дата* — это объект `Date`, соответствующий определенному моменту времени.

Таблица 16.2. *Методы для определения местного времени и даты*

Метод	Описание
`getFullYear()`	Возвращает год как четырехзначное число
`getYear()`	Возвращает значение, представляющее разность `getFullYear()` — 1900
`getMonth()`	Возвращает номер месяца (от 0 до 11)
`getDate()`	Возвращает число (от 1 до 31)
`getDay()`	Возвращает номер дня недели (от 0 до 6, причем 0 соответствует воскресенью, 1 — понедельнику и т. д.)
`getHours()`	Возвращает часы (от 0 до 23)
`getMinutes()`	Возвращает минуты (от 0 до 59)
`getSeconds()`	Возвращает секунды (от 0 до 59)
`getMilliseconds()`	Возвращает миллисекунды (от 0 до 999)

Следующий код выводит текущую дату в панель **Output**:

```
var months:Array = ["Январь", "Февраль", "Март", "Апрель", "Май", "Июнь",
"Июль", "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"];
var days:Array = ["Вс", "Пн", "Вт", "Ср", "Чт", "Пт", "Сб"];
var curDate:Date = new Date();
var y = curDate.getFullYear();
var m = months[curDate.getMonth()];
```

```
var d = curDate.getDate();
var weekDay = days[curDate.getDay()];
trace("Сегодня: " + d + " " + m + ", " + weekDay + ", " + y);
```

В листинге 16.1 приведен сценарий, реализующий часы (рис. 16.1). Для его работы на сцене необходимо разместить три экземпляра клипов, представляющих стрелки часов с именами `seconds`, `minutes` и `hours`. Также необходимо изобразить циферблат и правильно разметить его так, чтобы угол между соседними цифрами, изображающими часы, был равен 30 градусам. Важно, чтобы центр регистрации каждого символа, изображающего стрелку, совпадал с точкой, относительно которой будет выполняться вращение. В рабочей среде стрелки должны располагаться в вертикальном направлении, соответствующем 12-ти часам (при этом угол поворота каждого экземпляра должен быть равен нулю). Принцип действия сценария заключается в том, что текущее значение времени связывается с углом поворота соответствующей стрелки. Поскольку дуга окружности составляет 360 градусов, минутные и секундные деления размещаются на ней через каждые 6 градусов (360/60 = 6), а часовые деления — через 30 градусов (360/12 = 30). Для корректной работы часов важно точно разметить циферблат. Для вывода времени на сцене имеется динамическое текстовое поле с именем `timeTxt`.



Рис 16.1. Часы

Листинг 16.1. Часы

```
this.onEnterFrame = function() {
        // Создание объекта Date, хранящего текущее время
        var time:Date = new Date();
        //Сохранение текущего времени в локальных переменных
        var sec = time.getSeconds();
        var min = time.getMinutes();
        var hour = time.getHours();
        //Вращение стрелок
```

```
        seconds._rotation = sec * 6;
        minutes._rotation = (min + sec / 60) * 6;
        hours._rotation = (hour + min / 60) * 30;
        // Вывод времени (электронные часы)
        timeTxt.text = hour + " : " + min + " : " + sec;
};
```

Определение UTC-времени и даты

Методы, представленные здесь очень схожи с предыдущими, за исключением того, что они возвращают значение не по местному времени, а по всемирному UTC-времени. Таким образом, эти методы не принимают в расчет географическое положение и возвращают время гринвичского меридиана. Различие значений дат, возвращаемых в местном времени и в UTC-времени, составляет смещение для конкретного часового пояса. Так, например, разница во времени между Гринвичем и Санкт-Петербургом составляет −3 часа (20 часов по местному времени соответствует 17 часам по UTC). Полный перечень методов для определения UTC-времени и даты приведен в табл. 16.3.

Таблица 16.3. *Методы для определения UTC-времени и даты*

Метод	Описание
`getUTCFullYear()`	Возвращает год как четырехзначное число
`getUTCMonth()`	Возвращает номер месяца (от 0 до 11)
`getUTCDate()`	Возвращает число (от 1 до 31)
`getUTCDay()`	Возвращает номер дня недели (от 0 до 6, причем 0 соответствует воскресенью, 1 – понедельнику и т.д.)
`getUTCHours()`	Возвращает часы (от 0 до 23)
`getUTCMinutes()`	Возвращает минуты (от 0 до 59)
`getUTCSeconds()`	Возвращает секунды (от 0 до 59)
`getUTCMilliseconds()`	Возвращает миллисекунды (от 0 до 999)

Установка местного времени и даты

Как уже было сказано ранее, момент времени, представляемый объектом `Date`, может быть установлен непосредственно при создании экземпляра. Однако, уже после его создания может возникнуть необходимость в изменении даты. Для этих целей класс `Date` содержит ряд методов, позволяющих установить дату и время на основании местного времени.

Все методы, представленные в этом разделе (табл. 16.4), принимают один обязательный параметр (два из них могут принимать более одного параметра) и имеют схожий синтаксис:

дата.метод(*параметр*)

где *дата* — это объект Date, соответствующий определенному моменту времени.

Параметр *параметр* — это новое значения компонента даты. Параметр может быть как положительным, так и отрицательным числом. Полный перечень методов для установки местного времени и даты приведен в табл. 16.4.

Таблица 16.4. *Методы для установки местного времени и даты*

Метод	Описание
setFullYear()	Задает год как четырехзначное число. Возвращает новое время в миллисекундах. Наряду с первым обязательным параметром, задающим год, данный метод может принимать еще два необязательных параметра: месяц и число
setYear()	Задает число. Возвращает новое время в миллисекундах
setMonth()	Задает номер месяца. Возвращает новое время в миллисекундах. Наряду с первым обязательным параметром, задающим номер месяца, данный метод может принимать еще один необязательный параметр: число
setDate()	Задает номер дня недели (0 соответствует воскресенью, 1 — понедельнику и т. д.). Возвращает новое время в миллисекундах
setHours()	Задает часы. Возвращает новое время в миллисекундах
setMinutes()	Задает минуты. Возвращает новое время в миллисекундах
setSeconds()	Задает секунды. Возвращает новое время в миллисекундах
setMilliseconds()	Задает миллисекунды. Возвращает новое время в миллисекундах

Следует иметь в виду, что если значение параметра выходит за допустимые границы, он рассчитывается с учетом непрерывного следования времени. При этом значение других характеристик даты может измениться. Так, например, если текущий месяц декабрь (11), а метод setMonth() вызывается с параметром 12, то дата будет установлена на январь следующего года.

Ниже приводится функция daysInMonth(), вычисляющая количество дней в месяце для заданного года. Принцип ее действия основан на том, что функция создает два локальных объекта дат. Первая дата соответствует первому дню месяца, вторая дата соответствует первому дню следующего за ним месяца. Поскольку каждые сутки содержат 86 400 000 миллисекунд, можно просто вычислить количество дней в месяце.

```
function daysInMonth(year:Number, month:Number):Number{
        var d1:Date = new Date(year, month);
```

```
        var d2:Date = new Date(year, month)
        return Math.round((d2.setMonth(month+1)- d1)/86400000);
}
trace(daysInMonth(2004, 1)); // Выводит 29
```

Установка UTC-времени и даты

Установку даты на основании местного времени можно выполнить как при создании объекта `Date`, так и с помощью приведенных методов. Для того чтобы установить дату на основании всемирного времени (UTC), необходимо использовать методы, приведенные в табл. 16.5. По сути, эти методы являются аналогами методов, задающих дату по местному времени, однако они не учитывают географическое местоположение и ориентируются на гринвичское время.

Таблица 16.5. *Методы для установки UTC-времени и даты*

Метод	Описание
`setUTCFullYear()`	Задает год как четырехзначное число. Возвращает новое время в миллисекундах. Наряду с первым обязательным параметром, задающим год, данный метод может принимать еще два необязательных параметра: месяц и число
`setUTCMonth()`	Задает номер месяца. Возвращает новое время в миллисекундах. Наряду с первым обязательным параметром, задающим номер месяца, данный метод может принимать еще один необязательный параметр: число
`setUTCDate()`	Задает номер дня недели (0 соответствует воскресенью, 1 — понедельнику и т. д.). Возвращает новое время в миллисекундах
`setUTCHours()`	Задает часы. Возвращает новое время в миллисекундах
`setUTCMinutes()`	Задает минуты. Возвращает новое время в миллисекундах
`setUTCSeconds()`	Задает секунды. Возвращает новое время в миллисекундах
`setUTCMilliseconds()`	Задает миллисекунды. Возвращает новое время в миллисекундах

Работа с машинным временем

В классе `Date` имеется несколько методов, позволяющих оперировать машинным временем, то есть количеством миллисекунд, отделяющих полночь 1 января 1970 года от момента времени, представляемого объектом `Date`. К числу этих методов относятся:

- `getTime()` — возвращает машинное время для конкретного объекта `Date`. Данный метод не принимает параметров и возвращает количество миллисекунд, от-

деляющих полночь 1 января 1970 года по UTC от момента времени, представляемого объектом Date;

- setTime() — устанавливает дату и время на основании машинного времени, переданного ему в качестве параметра. Возвращает новое время в миллисекундах;
- Date.UTC() — возвращает временной промежуток в миллисекундах, отделяющий полночь 1 января 1970 года по UTC от момента времени, заданного параметрами данного метода. Этот метод (в отличие от всех остальных) является статическим, то есть вызывается с помощью конструктора Date(), а не с помощью конкретного объекта Date. Метод Date.UTC() имеет следующий синтаксис:

```
Date.UTC(year, month, date, hour, minute, second, millisecond)
```

Назначение параметров данного метода аналогично назначению параметров конструктора Date(), представленных в табл. 16.1. Обязательными являются только первые два из них. Данный метод может использоваться для создания объекта Date, представляющего момент времени по UTC.

```
// Задаем 22 часа, четверг по всемирному времени
var dUTC:Date = new Date(Date.UTC(2004, 10, 4, 22, 00));
trace(dUTC); /* Выводит Fri Nov 5 01:00:00 GMT+0300 2004 - локальное время, соответствующее, хранящемуся в dUTC всемирному */
```

Кроме того, следует иметь в виду, что в результате преобразования объекта Date в число возвращается машинное время, соответствующее хранимому в этом объекте моменту времени. Аналогичного результата можно добиться путем применения метода getTime().

```
var dUTC:Date = new Date(Date.UTC(1970, 0, 1, 0, 0, 0));
trace(Number(dUTC)); // Выводит 0
```

Создание календаря

В качестве иллюстрации возможностей работы с датой рассмотрим пример создания календаря. Наш календарь будет обладать следующими функциональными возможностями:

- вывод календаря за месяц в виде традиционной таблицы с указанием чисел и дней недели (рис. 16.2). Неделя должна начинаться с понедельника (а не с воскресенья, как в англоязычных странах) и воскресенья должны быть подсвечены красным цветом.
- возможность перевода месяца с помощью кнопок;
- возможность задания года;
- автоматическое обновление календаря при выборе нового месяца или задании нового года.

Сценарий, реализующий создание календаря, приведен в листинге 16.2. Листинг снабжен подробными комментариями, поэтому ограничимся принципиальным описанием его основных этапов.



Рис 16.2. Календарь

Ключевым моментом в создании календаря является формирование таблицы дат, в которой столбцы соответствуют дням недели, а строки — неделям. При этом следует учесть, что месяц может занимать от 4 до 6 недель, и что первое число месяца (как и последнее) может приходиться на любой день недели. Для формирования календаря мы создадим матрицу, состоящую из клипов, на монтажной линейке каждого из которых будет создано динамическое текстовое поле. Матрица будет состоять из 6 строк, соответствующих неделям, и из 7 столбцов, соответствующих дням недели. Создание клипов, как и создание текстовых полей, на их монтажных линейках будет выполняться программным образом. Клипы будут иметь имена, состоящие из слова `cell` и порядкового номера (от 0 до 41), а текстовые поля будут иметь имена, состоящие из слова `txt` и порядкового номера, совпадающего с порядковым номером родительских клипов. Позиционирование клипов для формирования матрицы выполняется с применением вложенного цикла, аналогично приему, показанному в листинге 12.3.

Заполнение таблицы дат будет выполняться в соответствии со следующим алгоритмом.

1. Определяем количество дней в месяце.
2. Определяем порядковый номер первого дня месяца. Причем, поскольку в нашем календаре первым днем недели должен быть понедельник, необходимо сместить нумерацию дней недели, используемую в классе `Date` на один день назад. Это означает, что понедельник будет иметь индекс 0, вторник — 1 и т. д, а воскресенье перейдет в конец недели, получив индекс 6.
3. Последовательно заполняем строки таблицы. При этом, начиная с текстового поля, соответствующего дню недели, определенному на шаге 2, и до тех пор, пока не будет заполнено количество полей, соответствующее количеству дней в месяце, выводим в текстовые поля числа 1, 2, 3, ..., N, где N — число последнего дня месяца. Во все остальные поля (то есть дни недели, предшествующие первому числу месяца и следующие за его последним числом) выводим пробельные символы `" "`.

Таким образом, формирование календаря на месяц выполняется только на основании сведений о числе дней в этом месяце и номере дня первого числа этого месяца.

Приведем краткое описание последовательности действий.

1. Вначале создается пустая таблица для вывода дат.
2. Отдельным блоком кода формируем еще одну строку таблицы с обозначениями дней недели. Эту строку помещаем над таблицей для вывода дат.
3. Создаем текстовое поле, в которое будет выводиться месяц. Это поле назовем `monthTxt` и по умолчанию будем выводить в него текущий месяц.
4. Создадим редактируемое текстовое поле для указания (вывода) года. Это поле назовем `yearTxt` и по умолчанию будем выводить в него текущий год.
5. Создадим функцию `calendar()`, осуществляющую вывод дат на основании переданных ей значений года и месяца.
6. Вызываем функцию `calendar()`, передавая ей текущие значения года и месяца, хранящиеся в переменных `curYear` и `curMonth` соответственно, для вывода календаря на текущий месяц по умолчанию.
7. Снабжаем созданные заранее кнопки `nextBut` и `prevBut` функциональностью, позволяющей при нажатии перемещаться к следующему или предыдущему месяцу. Новый месяц будет выводиться в поле `monthTxt`, при этом будет обновляться значение переменной `curMonth`, передаваемой в качестве параметра функции `calendar()` для обновления календаря. Для того чтобы не увеличивать объем кода, кнопки `nextBut` и `prevBut` были созданы в рабочей среде, хотя их позиционирование выполняется программным образом.
8. Обеспечиваем функциональность поля `yearTxt`. Если содержимое этого поля меняется, то при нажатии левой кнопки мыши заданное значение года вступает в силу и инициирует обновление календаря. Здесь в качестве источника события используется один из клипов, формирующих таблицу дат. В принципе, можно было бы создать объект-источник (listener), транслирующий это событие, однако это потребовало бы создания более громоздкого кода, не дающего в этой ситуации никаких преимуществ.

Листинг 16.2. Календарь

```
var months:Array = ["Январь", "Февраль", "Март", "Апрель", "Май", "Июнь",
"Июль", "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"];

var days:Array = ["Пн", "Вт", "Ср", "Чт", "Пт", "Сб", "Вс"];

var xSpace = ySpace = 10; // Расстояние между полями дат

var initX = initY = 200; // Координаты левого верхнего угла таблицы дат
                         // календаря

//СОХРАНЯЕМ ЗНАЧЕНИЯ ТЕКУЩЕЙ ДАТЫ

var curDate = new Date();

var curMonth = curDate.getMonth();

var curYear = curDate.getFullYear();
```

```
// СОЗДАНИЕ ТАБЛИЦЫ ИЗ ТЕКСТОВЫХ ПОЛЕЙ ДЛЯ ВЫВОДА ЧИСЕЛ МЕСЯЦА
for (var i = 0; i < 6; i++) {
        for (var j = 0; j < 7; j++) {
                this.createEmptyMovieClip("cell" + (i * 7 + j), i * 7 + j);
                this["cell" + (i * 7 + j)].createTextField("txt" +(i * 7 +
                j), i * 7 + j, 0, 0, 0, 0);
                with (this["cell" + (i * 7 + j)]["txt" + (i * 7 + j)]) {
                        autoSize = true;
                        text = " "
                        if (j % 6 == 0 && j != 0) textColor = 0xFF0000;
                }
                this["cell" + (i * 7 + j)]._x = j * (this.cell0._width +
                xSpace) + initX;
                this["cell" + (i * 7 + j)]._y = i * (this.cell0._height +
                ySpace) + initY;
        }
}
// СОЗДАНИЕ ЗАГОЛОВКА С ДНЯМИ НЕДЕЛИ
for (var i = 0; i < 7; i++) {
        this.createEmptyMovieClip("day" + i, i + 42);
        this["day" + i].createTextField("dTxt" + i, i + 42, 0, 0, 18, 20);
        with (this["day" + i]["dTxt" + i]) {
                background = true;
                backgroundColor = 0xCCCCCC;
                text = days[i];
        }
        this["day" + i]._x = this["cell" + i]._x - .5 * xSpace;
        this["day" + i]._y = initY - 30;
}
// ВЫВОД ТЕКУЩЕГО МЕСЯЦА
this.createTextField("monthTxt", 100, (day0._x + day6._x +
day6._width)/2, initY - 60, 0, 0);
monthTxt.autoSize = "center";
nextBut._x = day6._x;
prevBut._x = day0._x + day0._width;
nextBut._y = prevBut._y = monthTxt._y;
// СОЗДАНИЕ ПОЛЯ ДЛЯ ВВОДА (И ВЫВОДА) ТЕКУЩЕГО ГОДА
this.createTextField("yearTxt", 101, (day0._x + day6._x+ day6._width)/2,
initY - 80, 0, 0);
yearTxt.autoSize = "center";
```

```
yearTxt.type = "input";
yearTxt.maxChars = 4;
yearTxt.restrict = "0-9";
// ФУНКЦИЯ, ЗАПОЛНЯЮЩАЯ ПОЛЯ ДАТ НА ОСНОВАНИИ УСТАНОВЛЕННЫХ ГОДА И МЕСЯЦА
function calendar(year, month) {
        yearTxt.text = curYear;
        monthTxt.text = months[month];
        // Определяем количество дней в указанном месяце
        var d1:Date = new Date(year, month);
        var d2:Date = new Date(year, month)
        // Количество дней в указанном месяце
        var days = Math.round((d2.setMonth(month+1)- d1)/86400000);
        // s1 - переменная для хранения номера первого дня текущего
        // месяца. Следующая строка выполняет переход к более традиционной
        // системе нумерации дней, в соответствии с которой неделя
        // начинается с понедельника, то есть воскресенье имеет индекс 6,
        // а не 0.
        var s1 = (d1.getDay() == 0) ? 6 : (d1.getDay() - 1);
        var s = 1; /* Вспомогательная переменная, используемая как счетчик
        дней текущего месяца при выводе дат */
        // Заполнение полей дат для текущего месяца
        for (var i = 0; i < 6; i++) {
                for (var j = 0; j < 7; j++) {
                with (this["cell" + (i * 7 + j)]["txt" + (i * 7 + j)]) {
                        /* Заполняем подряд поля дат, начиная с поля,
                        соответствующего первому дню месяца, до тех пор,
                        пока не будет заполнено количество полей,
                        соответствующее количеству дней в текущем месяце */
                                if ((i * 7 + j) >= s1 && s <= days) {
                                        text = s++;
                                }
                                /* Поля дней, не вошедших в текущий месяц,
                                оставляем пустыми */
                                else {
                                        text = " ";
                                }
                                // Выделяем воскресенья красным цветом
                                if (j % 6 == 0 && j != 0) {
                                        textColor = 0xFF0000;
}}}}}
```

```
// ПО УМОЛЧАНИЮ ВЫВОДИМ КАЛЕНДАРЬ НА ТЕКУЩИЙ ГОД И МЕСЯЦ
calendar(curYear, curMonth);
// КНОПКИ, МЕНЯЮЩИЕИЕ ТЕКУЩИЙ МЕСЯЦ И ДАТЫ В КАЛЕНДАРЕ
nextBut.onPress = function(){
if (curMonth < 11) curMonth++;
else curMonth = 0;
calendar(curYear, curMonth);
}
prevBut.onPress = function(){
if (curMonth > 0) curMonth--;
else curMonth = 11;
calendar(curYear, curMonth);
}
// КОНТРОЛЬ ВВОДА ГОДА ПОЛЬЗОВАТЕЛЕМ
//При изменении содержимого поля года...
yearTxt.onChanged = function(){
/* Определяем обработчик, выполняющий обновление календаря при нажатии
мыши. Здесь в качестве источника события onMouseDown используем один из
клипов, формирующих таблицу дат */
this._parent.cell0.onMouseDown = function(){
// При нажатии мыши изменяем показания календаря...
curYear = Number(yearTxt.text);
calendar(curYear, curMonth);
/* ...и удаляем обработчик onMouseDown, чтобы обновление календаря
выполнялось только в случае изменения значения года. Если пользователь
просто передаст фокус текстовому полю, не внося в него изменения,
обновления календаря не произойдет */
delete this._parent.cell0.onMouseDown;
        }
}
```

Глобальная функция *getTimer()*

Наряду с методами класса `Date` в ActionScript имеется еще одна достаточно полезная глобальная функция, не имеющая отношения к классу `Date`, но связанная с контролем времени. Эта функция называется `getTimer()` и используется для контроля интервалов времени на протяжении воспроизведения Flash-фильма. Функция не принимает параметров и возвращает количество миллисекунд, прошедших с момента запуска Flash-фильма. Поскольку на протяжении воспроизведения фильма показания данной функции постоянно увеличиваются, это дает возможность замерить отрезок времени между двумя моментами. Так, например, если пользователь в последний раз нажал кнопку в момент, когда функция `getTimer()` возвращала значение A,

то для того, чтобы определить, сколько миллисекунд прошло между этим и следующим нажатием, нужно при очередном нажатии кнопки вычислить значение выражения:

```
var timeLeft = getTimer() − A
```

Узнать, сколько минут и секунд заключается в полученном интервале, можно следующим образом:

```
var min = Math.floor(timeLeft / 60000) ;
var sec = Math.floor((timeLeft % 60000) / 1000);
```

Функция `getTimer()` уже использовалась ранее в листинге 11.3 для определения временного интервала между щелчками мыши. Здесь же приведем пример создания таймера, выполняющего обратный отсчет от заданного в минутах времени.

Таймер состоит из текстового поля с именем `timerTxt` и двух кнопок с именами `setTimerBut` и `stopTimerBut` (рис. 16.3). В текстовое поле `timerTxt` пользователь вводит время в минутах. При нажатии кнопки `setTimerBut` происходит активация таймера, и в поле `timerTxt` периодически выводятся цифры обратного отсчета времени. По истечении заданного времени раздается звуковой сигнал. Для более корректной работы таймера кнопка `setTimerBut` становится неактивной во время выполнения обратного отсчета, а текстовое поле `timerTxt` недоступно для редактирования. Кнопка `stopTimerBut` применяется для остановки работы таймера. Ее нажатие приводит к обнулению значения в поле `timerTxt` и активации кнопки `setTimerBut` для следующего запуска таймера.

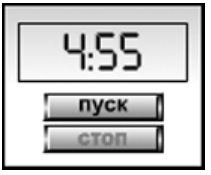


Рис. 16.3. Таймер

Сценарий, реализующий поставленную задачу, приведен в листинге 16.3. Отметим несколько деталей. Функция возвращает интервал времени с точностью до миллисекунды. Однако поскольку речь идет о постоянном выводе результатов обратного отсчета, для достижения подобной точности необходимо получать эти результаты каждую миллисекунду. В связи с этим наш таймер, строго говоря, не будет абсолютно точным, поскольку вычисление временного интервала будет выполняться не каждую миллисекунду, а значительно реже. В приведенном сценарии вычисление временного интервала выполняется с частотой, равной скорости воспроизведения фильма (с применением обработчика события `onEnterFrame`). Это означает, что при скорости воспроизведения 12 кадров в секунду погрешность таймера составит около 100 миллисекунд, то есть 1/10 секунды (при увеличении скорости воспроизведения, погрешность будет уменьшаться). Такая погрешность является вполне приемлемой

для большинства практических задач (например, для создания теста, где на каждый вопрос отводится определенный интервал времени). При необходимости читатель может сам внести изменения в представленный код, попробовав реализовать выполнение периодических замеров времени с помощью глобальной функции `setInterval()` (это даст независимость от скорости воспроизведения фильма и может увеличить точность). В представленном здесь таймере выводятся только минуты и секунды, поэтому мы считаем, что заданное время истекло, как только число секунд достигло нуля, и пренебрегаем оставшимися миллисекундами.

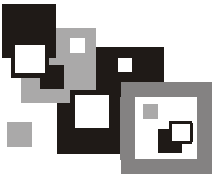
В приведенном ниже сценарии для динамического присоединения и запуска звука используются методы класса `Sound`. Эти возможности будут подробно описаны в *главе 19*.

Листинг 16.3. Таймер

```
/* Функция, реализующая функциональность таймера. Параметр t задает время
в миллисекундах, а параметр beeps указывает, сколько раз нужно повторить
звуковой сигнал по истечении времени t */
function timer(t:Number, beeps:Number):Void {
    // Фиксируем момент запуска таймера
    var t0 = getTimer();
    // Создание объекта Sound и присоединение звука из библиотеки
    var s:Sound = new Sound();
    s.attachSound("beep");
    this.onEnterFrame = function() {
        // Переменная, хранящая оставшееся время в миллисекундах
        var timeLeft = t - (getTimer() - t0);
        /* Вывод оставшегося времени в текстовое поле в минутах и
        секундах */
        timerTxt.text = Math.floor(timeLeft / 60000) + ":" +↳
        Math.floor((timeLeft % 60000) / 1000);
        // Если осталось меньше секунды...
        if (timeLeft < 1000) {
            // Обнуляем показания таймера
            timerTxt.text = "0:0";
            // Запускаем звук заданное количество раз
            s.start(0, beeps);
            /* Делаем текстовое поле редактируемым для ввода
            следующего значения */
            timerTxt.type = "input";
            // Активизируем кнопку запуска таймера
            setTimerBut.enabled = true;
```

```
                    // Удаляем обработчик
                    delete this.onEnterFrame;
                }
        };
}
// Кнопка для запуска таймера
setTimerBut.onPress = function() {
        /* Запускаем таймер только, если в текстовое поле введено число
        минут */
        if(Number(timerTxt.text)){
        /* Активизируем таймер, передавая функции timer() время, введенное
        в текстовое поле и количество повторов звукового сигнала */
        timer(Number(timerTxt.text) * 60000, 5);
        /* Отключаем кнопку запуска таймера, чтобы во время его работы
        нельзя было запустить его вновь */
        this.enabled = false;
        /* Отключаем возможность редактирования текстового поля во время
        работы таймера */
        timerTxt.type = "dynamic";
        }
};
// Кнопка останавливающая работу таймера
stopTimerBut.onPress = function() {
        timerTxt.text = "0:0";
        s.start(0, beeps);
        timerTxt.type = "input";
        setTimerBut.enabled = true;
        delete this._parent.onEnterFrame;
};
```

Глава 17



Классы *Mouse* и *Key*. Контроль мыши и клавиатуры

Кто может — делает, кто не может — учит.

Закон Х. Л. Менкена

Кто не может учить — управляет.

Дополнение Мартина

В этой главе будут рассмотрены возможности программного контроля мыши и клавиатуры, которые реализуются с помощью классов `Mouse` и `Key`.

Класс `Mouse` предоставляет возможность управления отображением стандартного курсора мыши, а также позволяет транслировать объектам сообщения о событиях, связанных с мышью.

Класс `Key` предоставляет набор свойств и методов, позволяющих осуществлять программный контроль клавиатуры. Эта возможность широко используется при разработке интерактивности, предполагающей управление объектами или выполнение определенных действий с помощью клавиатурных клавиш.

Контроль отображения курсора и обработка событий мыши

В классе `Mouse` имеется только четыре метода, причем все они являются статическими, то есть не требуют создания экземпляра, и вызываются непосредственно с помощью объекта `Mouse` с применением следующего синтаксиса:

`Mouse.`*`метод`*`()`

Два из этих методов отвечают за отображение стандартного курсора в пределах окна проигрывателя (`show()` и `hide()`). Два других позволяют реализовать трансляцию событий, обнаруживаемых объектом `Mouse`, другим объектам (`addListener()` и `removeListener()`).

Управление отображением курсора

Для управления отображением стандартного курсора используются методы `hide()` и `show()`. Метод `hide()` отключает отображение стандартного курсора мыши, а метод `show()` включает его отображение. Действие этих методов оказывает влияние на отображение курсора, только пока он находится в пределах окна проигрывателя. Отключить отображение курсора за пределами окна Flash Player, используя только средства ActionScript, нельзя. Также следует иметь в виду, что даже если отображение стандартного курсора отключено, при наведении на выделяемое текстовое поле будет выводиться стандартный текстовый курсор. Кроме того, при нажатии правой кнопки мыши в области окна проигрывателя выводится стандартное контекстное меню проигрывателя, и отображение курсора мыши автоматически включается. Причем курсор остается включен даже после закрытия контекстного меню. Пример использования методов `hide()` и `show()` был приведен в листинге 11.1.

Обработка событий мыши

Объект `Mouse` генерирует четыре различных события, связанных с мышью.

- Событие `onMouseDown` — генерируется при нажатии левой кнопки мыши в пределах окна проигрывателя.
- Событие `onMouseUp` — генерируется при отпускании левой кнопки мыши в пределах окна проигрывателя. Данное событие также генерируется при отпускании кнопки мыши за пределами окна автономного проигрывателя, если нажатие было выполнено при нахождении курсора мыши внутри его окна. Для того чтобы отпускание кнопки мыши за пределами окна фильма, размещенного на HTML-странице, привело к генерированию события `onMouseUp`, необходимо, чтобы нажатие было выполнено либо на кнопке, либо на выделяемом текстовом поле.
- Событие `onMouseMove` — генерируется периодически во время перемещения курсора внутри окна проигрывателя. Частота генерирования данного события определяется возможностями компьютера. Как и предыдущее, событие `onMouseMove` также генерируется при перемещении курсора с нажатой левой кнопкой мыши за пределами окна автономного проигрывателя до тех пор, пока кнопка не будет отпущена. Для того чтобы перемещение курсора мыши за пределами окна фильма, размещенного на HTML-странице, вызывало генерирование данного события, необходимо, чтобы при выходе курсора за пределы окна кнопка мыши была нажата либо на кнопке, либо на выделяемом текстовом поле. Событие `onMouseMove` генерируется при перемещении курсора внутри окна фильма вне зависимости от того, находится окно проигрывателя в фокусе (на HTML-странице) или нет.
- Событие `onMouseWheel` — генерируется при прокрутке колеса мыши.

Для того чтобы реализовать обработку событий мыши, необходимо создать объект-приемник и зарегистрировать этот объект на получение уведомлений о событиях от объекта `Mouse`. Эта процедура, как и первые три из перечисленных событий, уже была описана в *главе 11*. Там данные события были представлены как события, которые могут обрабатываться экземплярами клипов. В действительности все объекты `MovieClip` просто зарегистрированы в объекте `Mouse` на получе-

ние уведомлений о наступлении событий onMouseDown, onMouseUp и onMouseMove. Используя модель "источник-приемник", можно заставить любой объект реагировать на события мыши.

Для обработки события onMouseWheel необходимо создать объект-приемник и зарегистрировать его в объекте Mouse. В отличие от остальных обработчиков события мыши, метод обработчика события onMouseWheel принимает два параметра и имеет следующий синтаксис:

```
приемник.onMouseWheel = function(скорость прокрутки, целевой объект){
блок кода;
};
```

Здесь *приемник* — это объект-приемник, осуществляющий обработку события onMouseWheel, сгенерированного объектом Mouse.

Параметры передаются объектом Mouse при трансляции события onMouseWheel.

Параметр скорость прокрутки указывает число, определяющее, сколько строк было прокручено при повороте колеса мыши, инициировавшем событие onMouseWheel. Обычно за смещение колеса мыши на одну позицию выполняется прокрутка на 1—3 строки, хотя при резком повороте это значение может оказаться в 5—6 раз больше. При повороте колеса мыши вверх данный параметр принимает положительные значения, а при повороте вниз — отрицательные.

Параметр целевой объект — это ссылка на самый верхний в стопке объект (клип или текстовое поле), над которым находится курсор мыши во время поворота колеса. Если при вращении колеса мыши курсор не расположен на объекте, этот параметр принимает значение undefined. Если требуется указать только параметр *целевой объект*, то в качестве параметра *скорость прокрутки* можно передать значение null.

Приведем простой пример, демонстрирующий применение события onMouseWheel. Следующий сценарий позволяет с помощью колеса мыши поворачивать клип или текстовое поле, над которым находится указатель курсора. Если друг под другом расположены несколько объектов, поворачиваться будет только самый верхний из них. При вращении колеса мыши над векторной формой или кнопкой (button) будет поворачиваться весь уровень стека документов, к которому принадлежит данный объект. Это связано с тем, что при вращении колеса мыши над графическим объектом или кнопкой параметр *целевой объект* возвращает ссылку на уровень стека документов (например, _level0), к которому он принадлежит.

Примечание

Для того чтобы получить возможность контроля события onMouseWheel, необходимо передать фокус Flash-фильму. Для этого нужно щелкнуть мышью в любой точке внутри окна проигрывателя.

```
var obj:Object = new Object();
Mouse.addListener(obj);
```

```
obj.onMouseWheel = function(delta, target) {
        target._rotation += delta;
};
```

Следует отметить, что указание формальных параметров при определении метода обработчика не является обязательным. При трансляции события `onMouseWheel` фактические параметры все равно передаются методу обработчика, поэтому если их требуется использовать в теле функции, присваиваемой свойству `onMouseWheel`, к значению фактических параметров можно обратиться с помощью объекта `arguments`. Предыдущий пример можно переписать без применения формальных параметров следующим образом:

```
var obj:Object = new Object();
Mouse.addListener(obj);
obj.onMouseWheel = function() {
        arguments[1]._rotation += arguments[0];
};
```

Во многих случаях обработка событий выполняется с помощью экземпляров клипов, которые, как известно, не требуют регистрации в объекте `Mouse`, поскольку они регистрируются автоматически. Тем не менее в некоторых ситуациях уведомление о событиях мыши необходимо отправлять другим объектам, которые должны определенным образом реагировать на эти события. Рассмотрим пример создания шлейфа мыши, представленного на рис. 17.1. Сценарий, реализующий эту задачу, приведен в листинге 17.1.

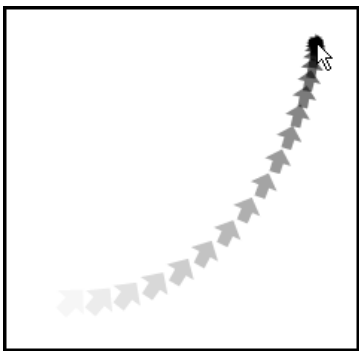


Рис. 17.1. Шлейф мыши

Шлейф состоит из динамически создаваемых экземпляров клипа с идентификатором связи `onion`, расположенного в библиотеке документа. Количество экземпляров, определяющее длину шлейфа, задается переменной `onionLength`. Имя каждого экземпляра состоит из слова `onion` и его порядкового номера. Размер экземпляров и их степень прозрачности постепенно изменяются от нулевого экземпляра до экземпляра с номером `onionLength`−1. При перемещении курсора мыши шлейф должен

следовать за ним, описывая траекторию, пройденную курсором, причем каждый клип должен быть повернут в направлении курсора. Для задания закона перемещения клипов, формирующих шлейф, мы использовали следующие рекуррентные формулы:

$$X_i^{t+1} = X_i^t + (X_{i-1}^t - X_i^t)/2;$$

$$Y_i^{t+1} = Y_i^t + (Y_{i-1}^t - Y_i^t)/2,$$

где $i = 1, 2, ..., \texttt{onionLength}-1$.

Здесь X_i^t и Y_i^t — это координаты *i*-го клипа в момент времени *t* по горизонтали и вертикали соответственно.

Обеспечение поворота в направлении курсора мыши реализуется с помощью метода `Math.atan2()` (*см. главу 15* и листинг 15.3). Позиционирование клипов, формирующих шлейф, выполняется с помощью функции `setOnion()`. Для регулярного обновления положения шлейфа функция `setOnion()` периодически вызывается с помощью глобальной функции `setInterval()`[1].

Для получения более аккуратного и естественного эффекта нужно сделать так, чтобы шлейф появлялся только при перемещении курсора и исчезал, если курсор остается неподвижным. Эту задачу можно решить с помощью следующего алгоритма.

1. Во время перемещения курсора фиксируем время последнего обнаружения события `onMouseMove`. Для этого создаем объект-приемник, обрабатывающий данное событие, и в теле метода обработчика события сохраняем значение, возвращаемое функцией `getTimer()` в переменной `lastMouse`.
2. Создаем функцию `checkUserMouse()`, сравнивающую значение `lastMouse` со значением, возвращаемым функцией `getTimer()`. Функции при вызове передается параметр `delay`, определяющий, сколько миллисекунд должен отображаться шлейф, пока курсор остается неподвижным. Если разница между сравниваемыми значениями больше значения `delay`, отключаем отображение клипов, формирующих шлейф (так как курсор находится в неподвижном состоянии больше указанного времени). В противном случае включаем их отображение (на тот случай, если оно было отключено, а курсор вновь начал двигаться).
3. Осуществляем периодический вызов функции `checkUserMouse()` с помощью функции `setInterval()`.

Следует отметить, что если время, передаваемое функции `checkUserMouse()` параметром `delay`, будет константой, то зритель либо не успеет оценить остаточное движение очень длинного шлейфа (при неподвижном курсоре), либо, наоборот, остаточное движение короткого шлейфа будет присутствовать на экране слишком

[1] Мы намеренно не использовали событие `onMouseMove` для позиционирования шлейфа, поскольку шлейф должен обладать остаточным движением. Если курсор уже остановился, шлейф должен еще какое-то время продолжать двигаться. Поэтому событие `onMouseMove` в данном случае не подходит. — *Прим. автора.*

долго. В связи с этим данный параметр мы передаем как функцию расстояния между начальным и конечным клипами шлейфа:

$$\texttt{delay} = \texttt{dist}^2,$$

где `dist` — это расстояние между начальным и конечным клипами шлейфа. Для вычисления этого расстояния используется функция `dist()`.

Листинг 17.1. Шлейф мыши

```
// Количество клипов, формирующих шлейф
var onionLength = 25;
// Масштаб последнего клипа в шлейфе
var maxSize = 150;
/* Создаем приемник события onMouseMove и регистрируем его на получение
уведомлений от объекта Mouse */
var obj:Object = new Object();
Mouse.addListener(obj);
/* В переменной lastMouse сохраняем время последней регистрации события
onMouseMove */
obj.onMouseMove = function() {
       lastMouse = getTimer();
};
/* Создаем экземпляры клипа, формирующие шлейф и задаем для них масштаб и
прозрачность с помощью объекта инициализации */
for (var i = 0; i < onionLength; i++) {
       this.attachMovie("onion", "onion" + i, i, {
                      _xscale:maxSize / onionLength * i + 100,
                      _yscale:maxSize / onionLength * i + 100,
                      _alpha:100 - 100 / onionLength * i
                      });
}
/* Отключаем отображение первого клипа в шлейфе, поскольку его координаты
будут совпадать с координатами курсора и поэтому к нему нельзя успешно
применить метод Math.atan2() для поворота в направлении курсора */
onion0._visible = false;
// Функция, реализующая поведение шлейфа
function setOnion(o:MovieClip):Void {
       // Заставляем первый клип следовать за курсором
       onion0._x = _xmouse;
       onion0._y = _ymouse;
       // Расчет координат и угла поворота для остальных клипов шлейфа
       for (var i = 1; i < onionLength; i++) {
```

```
                /* Модифицируем координаты i-го клипа на половину расстоя-
                ния между (i-1)-м и i-м клипами в шлейфе по горизонтали и
                вертикали соответственно */
                o["onion" + i]._x += (o["onion" + (i-1)]._x - o["onion" +↓
                i]._x) / 2;
                o["onion" + i]._y += (o["onion" + (i-1)]._y - o["onion" +↓
                i]._y) / 2;
                // Поворачиваем i-й клип в направлении курсора
                o["onion" + i]._rotation = Math.atan2((o["onion" + i]._y -↓
                _ymouse), (o["onion" + i]._x - _xmouse)) * 180 / Math.PI;
        }
}
/* Функция, вычисляющая (по теореме Пифагора) расстояние между начальным
и конечным клипами шлейфа и возвращающая полученный результат */
function onionDist():Number {
        return Math.sqrt(Math.pow(this["onion" + (onionLength - 1)]._x -↓
        onion0._x, 2) + Math.pow(this["onion" + (onionLength - 1)]._y -↓
        onion0._y, 2));
}
/* Функция, отключающая отображение шлейфа, если курсор мыши пребывает в
состоянии неподвижности в течение некоторого времени. Параметр delay
влияет на промежуток времени, в течение которого шлейф остается виден при
неподвижной мыши. Это время также ставится в зависимость от длины шлейфа,
чем длиннее шлейф, тем дольше он виден */
function checkUserMouse(o:MovieClip, delay:Number):Void {
        if (getTimer() - lastMouse > Math.pow(onionDist(), delay)) {
                for (var i = 1; i < onionLength; i++) {
                        o["onion" + i]._visible = false;
                }
        }
        else {
                for (var i = 1; i < onionLength; i++) {
                        o["onion" + i]._visible = true;
                }
        }
}
// Обновление состояния шлейфа
setInterval(setOnion, 50, this);
// Проверка неподвижности курсора
setInterval(checkUserMouse, 50, this, 2);
```

Контроль клавиатуры

Класс Key предоставляет программный инструментарий, позволяющий контролировать клавиатуру и реализовывать интерактивное взаимодействие с пользователем посредством клавиатуры.

Все свойства и методы класса Key являются статическими, то есть доступ к ним осуществляется без создания экземпляра.

События клавиатуры

Объект Key генерирует два события, позволяющие контролировать факт нажатия и отпускания клавиатурной клавиши. Объект Key может выступать транслятором данных событий, передавая сообщения об их обнаружении зарегистрированным объектам-приемникам. Для этих целей в классе Key определены стандартные методы addListener() и removeListener(), позволяющие зарегистрировать и отменить регистрацию объекта-приемника. События клавиатуры носят названия onKeyDown и onKeyUp. Данные события уже обсуждались в *главе 11*, поскольку, как и в случае с объектом Mouse, экземпляры клипов автоматически регистрируются на получение уведомлений от объекта Key.

Напомним, что обработка данных событий клипами может осуществляться либо с помощью обработчика onClipEvent, либо с применением методов обработчиков событий. Причем при работе с методами обработчиков событий клип реагирует на события клавиатуры только в том случае, если он находится в фокусе. Если сценарий непосредственно прикрепляется к клипу с помощью обработчика onClipEvent, то для контроля событий клавиатуры достаточно того, чтобы в фокусе находилось окно проигрывателя.

Эта особенность делает применение методов обработчиков onKeyDown и onKeyUp достаточно неэффективным и негибким. В то же время прикрепление сценария к клипу имеет целый ряд недостатков, которые обсуждались ранее. В связи с этим наиболее универсальным и действенным подходом при обработке событий клавиатуры является создание объекта-приемника и регистрация его на получение уведомлений о событиях от объекта Key. Для реализации подобной модели достаточно только того, чтобы в фокусе находилось окно проигрывателя.

Важной особенностью события является то, что при удерживании нажатой клавиши оно генерируется периодически с достаточно высокой частотой. Это имеет место при обработке данного события как с помощью метода обработчика объекта-приемника, так и при использовании обработчика клипа onClipEvent.

Примечание

Заметим, что частота генерирования события onKeyDown при удерживании нажатой клавиши определяется настройками операционной системы. В Windows эти настройки можно задать на вкладке **Speed** панели управления **Control Panel**.

Следующий код поворачивает клип mc на 15° при нажатии любой клавиатурной клавиши. Если клавиша остается нажатой, клип будет вращаться с дискретностью 15°. При отпускании клавиши клип mc возвращается в исходное положение.

```
var obj = new Object();
Key.addListener(obj);
obj.onKeyDown = function() {
       mc._rotation += 15;
};
obj.onKeyUp = function() {
       mc._rotation = 0;
};
```

Обратите внимание, что в этом сценарии не производится проверка, какая именно клавиша нажата (в то время как чаще всего конкретные задачи требуют именно этого). Для выполнения такой проверки можно воспользоваться методом Key.isDown(), описание которого приводится далее.

Определение характеристик и проверка состояния клавиши

В классе Key имеется два метода, позволяющих определить две разные характеристики нажатой клавиши. Дело в том, что каждой клавиатурной клавише соответствует физическое местоположение на клавиатуре, и при этом данная клавиша может отвечать за ввод различных символов. Местоположение клавиш универсально, в то время как символ клавиши зависит от языка клавиатуры и используемой системой кодировки. В связи с этим следует различать понятие виртуального кода клавиши и кодовой позиции символа клавиши в используемой кодировке. Под виртуальным кодом понимается число, соответствующее местоположению клавиши. Виртуальный код клавиши отвечает ее физическому местоположению на клавиатуре, а не соответствующему ей символу, и не зависит от регистра и используемого языка. Он фиксирует только местоположение клавиши на клавиатуре. В связи с этим, например, одинаковые цифровые клавиши на главной и на дополнительной клавиатуре имеют разные значения виртуального кода. Аналогично разным символам "я" и "z" соответствует один и тот же виртуальный код, поскольку они соответствуют одной и той же клавише.

В то же время код символа, соответствующего клавише, определяется кодировкой, поэтому для символов "я" и "z", расположенных на одной клавише, значения кодов символов будут разными. Значение кода символа клавиши зависит от кодировки, раскладки клавиатуры и от регистра.

Определение виртуального кода клавиши

Для определения виртуального кода клавиши используется метод getCode(), имеющий следующий синтаксис:

```
Key.getCode()
```

Данный метод возвращает число, соответствующее виртуальному коду последней нажатой клавиши. Следует отметить, что при работе на платформе, отличной от Windows, Flash Player автоматически выполняет перевод местных кодов в Windows-эквивалентные. В связи с этим данный метод является универсальным средством определения местоположения нажатой клавиши. Следующий код выводит в панель **Output** виртуальный код последней нажатой клавиши.

```
var obj = new Object();
Key.addListener(obj);
obj.onKeyDown = function() {
       trace(Key.getCode());
};
```

Если немного поэкспериментировать, то можно убедиться, что метод `getCode()` возвращает виртуальные коды большинства клавиш, расположенных на клавиатуре (включая, например, клавиши функций <F1>, < F2> и т. д.). Исключение составляют только некоторые специфические клавиши (например, <Print Screen>) и клавиши, наличие которых на разных клавиатурах отнюдь не обязательно (например, клавиша вывода меню Windows **Start Up** (Пуск)). Если на момент вызова метода `getCode()` ни одна клавиша не была нажата, метод возвращает значение 0. Приведенный в листинге 17.2 код позволяет вращать клип `mc` по и против часовой стрелки при нажатии клавиш <Right> (виртуальный код 39) и <Left> (виртуальный код 37) соответственно.

Листинг 17.2. Управление объектом с помощью клавиатуры (I)

```
var obj = new Object();
Key.addListener(obj);
obj.onKeyDown = function(){
mc.onEnterFrame = function() {
       if(Key.getCode() == 37) mc._rotation -= 15;
       if(Key.getCode() == 39) mc._rotation += 15;
       };
};
obj.onKeyUp = function(){
delete mc.onEnterFrame;
};
```

Для удобства виртуальные коды специальных клавиш хранятся в классе `Key` в качестве статических свойств (констант). Перечень этих свойств представлен в табл. 17.1.

Таблица 17.1. *Свойства класса* `Key`*. Виртуальные коды клавиш*

Свойство	Код	Свойство	Код	Свойство	Код
`Key.BACKSPACE`	8	`Key.ENTER`	13	`Key.PGUP`	33
`Key.CAPSLOCK`	20	`Key.ESCAPE`	27	`Key.RIGHT`	39
`Key.CONTROL`	17	`Key.HOME`	36	`Key.SHIFT`	16
`Key.DELETEKEY`	46	`Key.INSERT`	45	`Key.SPACE`	32
`Key.DOWN`	40	`Key.LEFT`	37	`Key.TAB`	9
`Key.END`	35	`Key.PGDN`	34	`Key.UP`	38

Определение кода символа клавиши

Для определения кода символа клавиши используется метод `getAscii()`, имеющий следующий синтаксис:

```
Key.getAscii()
```

Метод `getAscii()` возвращает код символа последней нажатой клавиши в кодировке, используемой операционной системой. Формально коды символов, возвращаемые данным методом, соответствуют кодировке Latin 1, описывающей 256 символов (первые 128 символов которой совпадают с ASCII). Поскольку код символа клавиши зависит от текущего языка клавиатуры, значение, возвращаемое методом `getAscii()`, для одной и той же клавиши может отличаться. Цифрам и символам английского алфавита соответствуют кодовые значения кодировки ASCII (отсюда и название метода). Однако кодовые значения кириллических символов выходят за пределы кодировки ASCII и формально соответствуют кодовым позициям символов второй половины кодировки Latin 1.

Поскольку строчным и прописным буквам соответствуют разные значения кода при вводе буквенного символа с одновременным нажатием клавиши <Shift>, метод `getAscii()` возвращает код прописной буквы. Следует отметить, что метод `getAscii()` различает регистр латинских букв, однако возвращает одни и те же значения для строчных и прописных кириллических.

В отличие от метода `getCode()` метод `getAscii()` способен регистрировать нажатие только некоторых специальных клавиш. К их числу относятся: <Backspace> (8), <Delete> (127), <Enter> (13), <Escape> (27), <Space> (32), <Tab> (9). При нажатии остальных небуквенных и нечисловых клавиш метод `getAscii()` возвращает значение 0.

В листинге 17.3 приведен сценарий, выводящий виртуальный код и код символа последней нажатой клавиши в динамические текстовые поля с именами `getCodeTxt` и `getAsciiTxt` соответственно.

Листинг 17.3. Определение кода нажатой клавиши

```
this.createTextField("getCodeTxt", 1, 50, 5, 35, 20); // Поля для вывода
this.createTextField("getAsciiTxt", 2, 50, 25, 35, 20);
```

```
this.createTextField("codeLabelTxt", 3, 0, 5, 50, 20); // Подписи к полям
this.createTextField("asciiLabelTxt", 4, 0, 25, 50, 20);
getCodeTxt.border = true;
getAsciiTxt.border = true;
codeLabelTxt.text = "CODE";
asciiLabelTxt.text = "ASCII";
var obj = new Object();
Key.addListener(obj);
obj.onKeyDown = function() {
       getCodeTxt.text = Key.getCode();
       getAsciiTxt.text = Key.getAscii();
};
```

Проверка состояния режимов клавиатуры

В некоторых ситуациях может потребоваться определить, активен ли один из режимов, контролируемых специальными клавишами (например, режим Caps Lock). Так, например, при вводе пользователем пароля можно сообщить ему об активном режиме Caps Lock, чтобы исключить ввод ошибочного пароля (прописные буквы вместо строчных). Для определения состояния таких клавиш используется метод `isToggled()`. Метод `isToggled()` позволяет контролировать состояние специальных клавиш <Caps Lock> и <Num Lock>, возвращая булево значение `true` или `false` в зависимости от того, включен соответствующий режим или нет. Вызов данного метода осуществляется следующим образом:

```
Key.isToggled(код)
```

Здесь *код* — это код клавиши <Caps Lock> (20), <Num Lock> (144) или <Scroll Lock> (145). Если в качестве параметра методу передается код любой другой клавиши, он возвращает значение `true`. Следующий сценарий иллюстрирует применение данного метода:

```
var obj = new Object();
Key.addListener(obj);
obj.onKeyDown = function() {
       if (Key.isToggled(20)) {
              trace("Режим Caps Lock включен");
       }
       else {
              trace("Режим Caps Lock отключен");
       }
};
```

Проверка нажатия заданной клавиши

Для того чтобы проверить, является ли заданная клавиша нажатой, используется метод `isDown()`. Данный метод осуществляет проверку нажатия клавиши, указанной в качестве его параметра, и возвращает булево значение `true`, если эта клавиша нажата, и `false` — если клавиша не нажата. Метод `isDown()` имеет следующий синтаксис:

```
Key.isDown(код)
```

Здесь *код* — это виртуальный код клавиши.

Метод `isDown()` часто используется в проверочных выражениях условных предложений для проверки условия нажатия клавиши. В принципе, когда речь идет о проверке нажатия одной клавиши, с этой задачей можно справиться, используя подход, показанный в листинге 17.2. Приведем несколько измененный код листинга 17.2, в котором факт нажатия проверяется с помощью метода `isDown()`.

```
var obj = new Object();
Key.addListener(obj);
obj.onKeyDown = function(){
mc.onEnterFrame = function() {
       if(Key.isDown(Key.LEFT)) mc._rotation -= 15;
       if(Key.isDown(Key.RIGHT)) mc._rotation += 15;
       };
};
obj.onKeyUp = function(){
delete mc.onEnterFrame;
};
```

Важным преимуществом, делающим метод `isDown()` незаменимым, является возможность проверки нажатия сразу нескольких клавиш. При одновременном нажатии нескольких клавиш каждый метод `isDown()`, проверяющий нажатие одной из них, будет возвращать значение `true`. Так, при тестировании приведенного сценария можно обнаружить, что при одновременном нажатии клавиш <Right> и <Left> вращение клипа `mc` прекращается. Это происходит вследствие того, что при одновременном удерживании двух клавиш выполняются условия обоих условных предложений и поскольку дискретность поворота в обоих направлениях одинакова, поворот не выполняется.

В листинге 17.2 использовался подход, при котором метод обработчика события `onEnterFrame` определялся только при нажатии клавиши и удалялся при ее отпускании. Этот прием позволяет сэкономить машинные ресурсы, поскольку инструкции обработчика события выполняются только на протяжении времени, пока клавиша остается нажатой. При разработке интерактивности, предполагающей интенсивное использование клавиатуры и контроль нажатия сочетаний клавиш (например, в играх), можно постоянно осуществлять проверку нажатия клавиш в теле обработчика события `onEnterFrame` и не удалять его при отпускании клавиши. Если же следо-

вать принципу, показанному в листинге 17.2, то в методе обработчика onKeyUp необходимо предусмотреть проверку того, что все используемые клавиши остаются ненажатыми, и только в этом случае удалить метод обработчика onEnterFrame. Это необходимо сделать для контроля ситуации, когда пользователь нажимает одновременно две клавиши (например, вверх и вправо), а затем отпускает одну из них. При этом перемещение объекта должно продолжаться, даже несмотря на обнаружение события onKeyUp. В качестве пояснения приведем сценарий, реализующий управление объектом с помощью клавиатуры. В листинге 17.4. представлен код, позволяющий перемещать клип mc с помощью клавиш управления курсором.

Листинг 17.4. Управление объектом с помощью клавиатуры (II)

```
var increment:Number = 10; // Шаг смещения клипа
var obj = new Object();
Key.addListener(obj);
obj.onKeyDown = function(){
mc.onEnterFrame = function() {
        if(Key.isDown(Key.LEFT)) mc._x -= increment;
        if(Key.isDown(Key.RIGHT)) mc._x += increment;
        if(Key.isDown(Key.UP)) mc._y -= increment;
        if(Key.isDown(Key.DOWN)) mc._y += increment;
        };
};
obj.onKeyUp = function(){
// Если ни одна из клавиш не нажата, прекращаем перемещение
if (!Key.isDown(Key.LEFT) && !Key.isDown(Key.RIGHT) &&
!Key.isDown(Key.UP) && !Key.isDown(Key.DOWN)){
delete mc.onEnterFrame;
        }
};
```

Сценарий, представленный в листинге 17.5, позволяет реализовать более сложный характер движения управляемого клипа с именем mc, учитывающий его инерционность и возможность придания ускорения. При нажатии и удерживании клавиши управления курсором клавиатуры клип mc начинает перемещаться, автоматически разворачиваясь в направлении движения (для этого используется метод Math.atan2()). При нажатии клавиши <Space> клип получает ускорение, однако скорость может возрастать только до заданного максимального значения, хранящегося в переменной maxSpeed. При отпускании клавиши или нажатии другой клавиши клип еще некоторое время движется по инерции.

Приведенный код снабжен подробными комментариями, поэтому рассмотрим только принцип реализации перемещения клипа. Перемещение клипа реализуется с помощью обработчика onEnterFrame. Это означает, что расчет положения клипа выполняется постоянно с частотой, равной скорости воспроизведения документа.

Координаты клипа изменяются по следующему закону:

$$X_{t+1} = X_t + \texttt{xSpeed}_{t+1};$$

$$Y_{t+1} = Y_t + \texttt{ySpeed}_{t+1}.$$

Здесь X_t и Y_t — это координаты клипа в момент времени t.

Переменные $\texttt{xSpeed}_t$ и $\texttt{ySpeed}_t$ определяют смещение клипа в момент времени t по горизонтали и по вертикали соответственно, то есть это составляющие общего перемещения. При нажатии соответствующих клавиш значения данных переменных рассчитывается по следующим формулам:

$$\texttt{xSpeed}_{t+1} = \texttt{xSpeed}_t + \texttt{speedUp} \cdot \texttt{accelerate};$$

$$\texttt{ySpeed}_{t+1} = \texttt{ySpeed}_t + \texttt{speedUp} \cdot \texttt{accelerate}.$$

Если же клавиши, управляющие движением в соответствующем направлении, не нажаты, то $\texttt{xSpeed}_t$ и $\texttt{ySpeed}_t$ вычисляются следующим образом:

$$\texttt{xSpeed}_{t+1} = \texttt{xSpeed}_t \cdot \texttt{decay};$$

$$\texttt{ySpeed}_{t+1} = \texttt{ySpeed}_t \cdot \texttt{decay},$$

где `decay` — это переменная, хранящая постоянное значение, определяющее инерционность клипа. Значение переменной `decay` должно лежать в диапазоне от 0 до 1.

Переменная `speedUp` хранит постоянное значение, определяющее скорость разгона клипа.

Переменная `accelerate` хранит значение ускорения. Величина этого параметра меняется в зависимости от того, нажата ли клавиша <Space>, но всегда должна оставаться больше 1. При нажатой клавише `accelerate` вычисляется следующим образом:

$$\texttt{accelerate}_{t+1} = \texttt{accelerate}_t + 1$$

Если клавиша <Space> не нажата, значение ускорения уменьшается, причем при достижении единицы уменьшение прекращается (это обеспечивается проверкой с применением условного предложения):

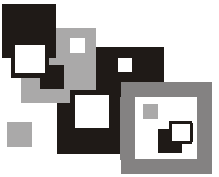
$$\texttt{accelerate}_{t+1} = (1 - \texttt{decay}) \cdot \texttt{accelerate}_t$$

Листинг 17.5. Перемещение клипа с помощью клавиатуры с учетом инерции и контролем ускорения

```
var xSpeed = 0;//Начальное смещение по горизонтали
var ySpeed = 0;//Начальное смещение по вертикали
var speedUp = 0.5;//Скорость разгона
var accelerate = 1;//Начальное значение ускорения
var decay=0.9;//Константа затухания (определяет инерционность)
var maxSpeed=25;//Максимально допустимая скорость
mc.onEnterFrame=function(){
//------------УСКОРЕНИЕ-----------------------------
//При нажатии SPACE - ускорение возрастает
```

```
if(Key.isDown(Key.SPACE)) accelerate+=1;
/*Если SPACE не нажата, ускорение затухает, достигая значения 1, при
котором ускорение отсутствует, то есть клип движется равномерно или
покоится*/
else accelerate*=1-decay;
if (accelerate<1) accelerate=1;
//-----------ВЫЧИСЛЕНИЕ СМЕЩЕНИЙ----------------------------------------
/*Если нажата клавиша LEFT или RIGHT, клип движется в соответствующем
направлении с текущим значением ускорения, в противном случае смещение
по горизонтали затухает*/
if(Key.isDown(Key.LEFT)) xSpeed-=speedUp*accelerate;
       else if (Key.isDown(Key.RIGHT)) xSpeed+=speedUp*accelerate;
       else xSpeed*=decay;
/*Если нажата клавиша UP или DOWN, клип движется в соответствующем
направлении с текущим значением ускорения, в противном случае смещение
по вертикали по инерции затухает*/
if(Key.isDown(Key.UP)) ySpeed-=speedUp*accelerate;
       else if(Key.isDown(Key.DOWN)) ySpeed+=speedUp*accelerate;
       else ySpeed*=decay;
//-----------КОНТРОЛЬ СКОРОСТИ-----------------------------------------
/*Фактическая скорость вычисляется по теореме Пифагора как квадратный
корень из суммы квадратов смещений по горизонтали и вертикали*/
var speed=Math.sqrt(xSpeed*xSpeed+ySpeed*ySpeed);
/*Если фактическая скорость достигла или превысила предельное значение,
снижаем ее путем уменьшения смещения клипа по горизонтали и вертикали
пропорционально превышению скорости*/
if (speed>=maxSpeed) {
       xSpeed*=maxSpeed/speed;
       ySpeed*=maxSpeed/speed;
}
//-----------ПЕРЕМЕЩЕНИЕ КЛИПА-----------------------------------------
//Перемещение клипа
mc._x+=xSpeed;
mc._y+=ySpeed;
//Поворот клипа по направлению движения
mc._rotation = Math.atan2(ySpeed, xSpeed)/(Math.PI/180)
//Ограничение выхода клипа за пределы рабочей области
if(mc._x>Stage.width+.5*mc._width) mc._x = - .5*mc._width;
if(mc._x<-.5*mc._width) mc._x = Stage.width + .5*mc._width;
if(mc._y>Stage.height + .5*mc._height) mc._y = - .5*mc._height;
if(mc._y< - .5*mc._height) mc._y = Stage.height + .5*mc._height;
};
```

Глава 18



Класс *Color*. Работа с цветом

Истина — в оттенках.

В основе работы с цветом в Flash лежит цветовая модель RGB. Это связано с тем, что Flash-фильм предназначен исключительно для вывода на экран монитора (или на иное электронное устройство, например, мультимедийный проектор), в котором для визуализации изображения используется именно эта модель цвета. В рабочей среде Flash для синтеза цвета можно использовать как модель RGB, так и более близкую к реальному восприятию цвета человеком модель HSB. Тем не менее все методы класса `Color` работают исключительно с RGB-представлением цвета. В связи с этим для программного синтеза цвета на основе модели HSB необходимо выполнить переход из одной модели в другую. Описание цветовых моделей RGB и HSB, а также алгоритмы перевода значения цвета из одной модели в другую будут приведены в этой главе.

Одной из важнейших характеристик цвета является глубина (color depth). Глубина цвета характеризует объем информации, необходимый для хранения значения этого цвета. Цвет в Flash может иметь глубину 24 или 32 бита. Характерной особенностью работы с цветом в Flash является возможность придания цвету прозрачности. Объект, окрашенный прозрачным цветом, сам обладает прозрачностью, сквозь него можно видеть нижележащие объекты. Для представления непрозрачных цветов в Flash используется 24-битовое число. Для представления цветов, обладающих прозрачностью, требуется уже 32 бита (то есть прозрачный цвет занимает на 1 байт больше, чем такой же непрозрачный).

Класс `Color` предоставляет возможности динамического управления цветом клипов, кнопок и текстовых полей (типа Dynamic или Input). Несмотря на то, что класс `Color` содержит всего четыре метода, с их помощью можно реализовать полный контроль над цветом (или конкретной цветовой составляющей) объекта и над степенью его прозрачности. Принцип применения методов класса `Color` достаточно прост, однако для их успешного использования необходимо хорошо понимать механику и закономерности цветовых моделей, лежащих в основе управления цветом в компьютерной среде. В связи с этим в начале данной главы мы приведем некоторые сведения по работе с цветом, а затем на их основе опишем методы класса `Color` и рассмотрим практические примеры их применения.

Цветовая модель RGB

Цветовая модель RGB (Red, Green, Blue — красный, зеленый, синий) играет ключевую роль в физической реализации формирования цвета аппаратными устройствами, в частности, дисплеем. Именно на основе модели RGB происходит синтезирование цвета монитором компьютера, поэтому при работе с проектами, ориентированными на распространение в компьютерной среде, нужно понимать общие принципы ее применения.

В основе цветовой модели RGB лежат три *первичных,* или *основных* цвета: **R**ed (красный), **G**reen (зеленый) и **B**lue (синий). Первичными называются цвета, позволяющие при смешении воспроизвести наиболее широкий цветовой охват, при этом характерной особенностью первичных цветов является то, что ни один их них не может быть получен путем смешения двух других. Свое название модель RGB получила по первым буквам образующих ее первичных цветов.

Все цвета, входящие в модель RGB, получаются путем аддитивного смешения (сложения) первичных цветов в определенных пропорциях. При этом следует понимать, что речь идет не о смешении цветовых пигментов, а о синтезе цветных световых лучей или *цветовых стимулов.* Аддитивным называется смешение, происходящее, когда свет от нескольких источников соединяется прежде, чем достигнет глаза. В результате такого смешения получается световой луч большей интенсивности и иного оттенка.

Каждая цветовая составляющая модели RGB может иметь 256 градаций интенсивности, характеризующих долю участия этой составляющей в формировании результирующего цвета. Значения интенсивности для каждой цветовой составляющей обычно обозначаются числами от 0 до 255. Модель RGB применяется в работе с излучающими объектами, поэтому нулевое значение цветовой составляющей означает отсутствие излучения для данного цвета, значение 255 соответствует максимальной интенсивности данного цвета.

Графическое представление цветовой модели RGB изображено на рис. 18.1. Цветовое пространство RGB имеет размерность 3, и любой синтезируемый в рамках этой модели цвет представляется точкой, расположение которой описывается тремя координатами.

Каждая координатная ось цветового пространства соответствует определенному первичному цвету. По оси отсчитывается интенсивность цвета, которая имеет нулевое значение в начале координат и значение 255 в максимуме. Цветовое тело модели RGB имеет форму куба. Таким образом, любой цвет описывается тремя значениями координат как точка, лежащая внутри кубического цветового тела. В начале координат (0, 0, 0) все цветовые составляющие имеют нулевую интенсивность, то есть излучение отсутствует, что соответствует *черному* цвету. В точке с максимальными значениями трех цветовых составляющих (255, 255, 255) находится белый цвет. Смешение первичных цветов в равных пропорциях дает серый цвет. Чем ниже интенсивность каждой составляющей, тем ближе серый цвет к черному. С увеличением интенсивности серый приближается к белому цвету. Точки с равными значениями интенсивности каждой составляющей лежат на диагонали куба, соединяющей точки черного и белого цветов. Соответственно, на этой диагонали располагается

шкала серого (Grayscale). Цвета, лежащие на диагонали цветового куба, называются *ахроматическими*, поскольку они не обладают характеристикой цветового тона.

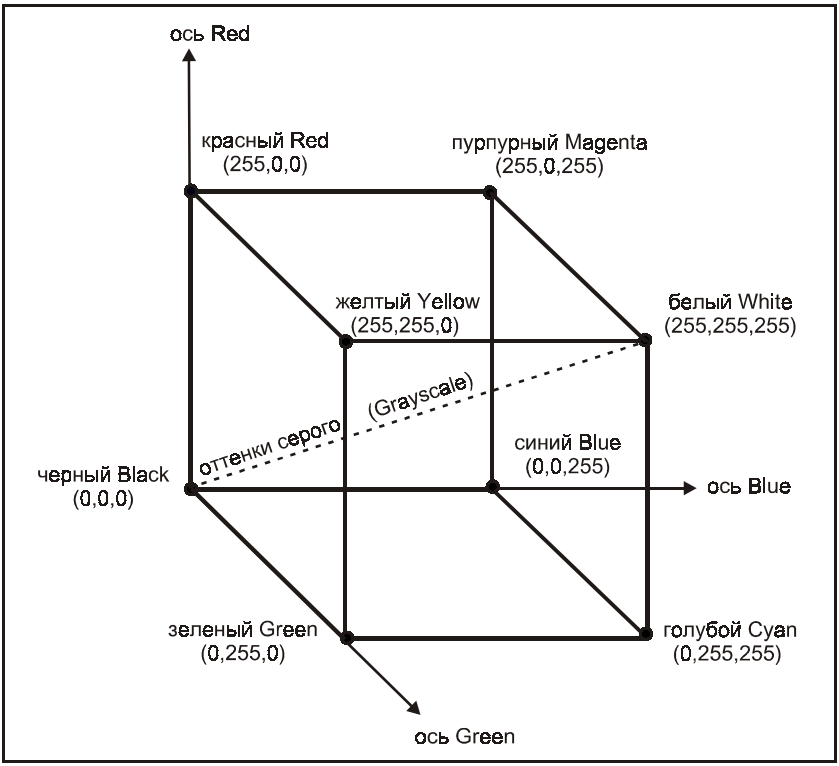


Рис. 18.1. Цветовое пространство модели RGB

Хроматические (то есть имеющие цветовой тон) цвета могут быть получены путем смешения трех составляющих в разных соотношениях. Причем поскольку речь идет не о смешении красок, а о смешении световых потоков, результаты этого смешения сильно отличаются от художественной практики. Так, при смешении красной и зеленой красок на холсте мы получаем бурый (или коричневый) цвет, в то время как смешение этих цветов в модели RGB даст нам желтый. Основные закономерности такого смешения представлены на рис. 18.2 и описаны далее.

Смешение **красного** и **зеленого** цветов при нулевом значении синего дает желтый цвет. При уменьшении интенсивности красного получается желто-зеленый цвет, а уменьшение интенсивности зеленого дает оранжевый.

Смешение **зеленого** и **синего** цветов при нулевом значении красного дает голубой цвет. С уменьшением интенсивности зеленого получается синеватый цвет; соответственно, уменьшая интенсивность синего, можно получить зеленоватый цвет.

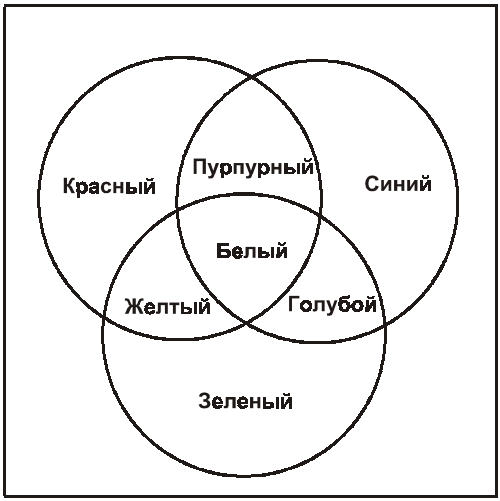


Рис. 18.2. Аддитивное смешение цветов в модели RGB

Смешение **синего** и **красного** дает пурпурный цвет. При ослаблении синей составляющей цветовой тон смещается в сторону розового, а при уменьшении интенсивности красной — в сторону фиолетового.

Для получения чистых красного, зеленого или синего цветов необходимо задать максимальную интенсивность соответствующих составляющих (255), при этом задав значения интенсивности двух других составляющих равными нулю.

Как уже было сказано, любой цвет в модели RGB может быть описан тремя координатами, значения которых могут изменяться в диапазоне от 0 до 255. Следовательно, для описания цвета необходимо использовать набор из трех чисел. Однако существует более компактная форма записи цветовых координат, использующая шестнадцатеричный формат (**Hex**adecimal). Такая форма представления цвета широко используется в приложениях, ориентированных на Интернет. Шестнадцатеричное представление цвета позволяет обозначать цвет одним шестиразрядным числом. В этом числе на значение каждой цветовой составляющей отводится два шестнадцатеричных разряда. Значение каждой составляющей изменяется от 0 до FF. Два левых разряда описывают интенсивность красной составляющей, два следующих — зеленой, а два правых разряда — синей. Так, красный цвет имеет значение FF0000, зеленый — 00FF00, синий — 0000FF, черный — 000000, а белый — FFFFFF.

Довольно часто на практике возникают задачи, требующие выделения значений цветовых составляющих из шестнадцатеричного значения цвета или, наоборот, объединения отдельных значений цветовых составляющих в единое число. Для выполнения этих действий очень удобно использовать побитовые операции. Их описание, а также пример выделения цветовых составляющих содержатся в *разд. "Побитовые операции" главы 6*. Здесь приведем две функции `rgbToComps()` и `rgbFromComps()`, выполняющие выделение составляющих из значения цвета и объединение значений составляющих в единое число соответственно.

```
/* Функция принимает в качестве параметра шестнадцатеричное число
и возвращает массив со значениями цветовых составляющих */
function rgbToComps(rgb:Number):Array {
        var r = (rgb&0xFF0000)>>16;
        var g = (rgb&0xFF00)>>8;
        var b = (rgb&0xFF);
        return [r, g, b];
}
/* Функция принимает в качестве параметра массив со значениями цветовых
составляющих и возвращает шестнадцатеричное число, описывающее цвет */
function rgbFromComps(rgb:Array):Number{
        return rgb[0]<<16|rgb[1]<<8|rgb[2];
}
```

Цветовая модель HSB

Цветовая модель RGB отражает физические принципы формирования цвета, однако не учитывает особенностей восприятия цвета человеком. Цветовая модель HSB позволяет реализовать способ синтеза цвета, который гораздо ближе соответствует психологическому цветовому восприятию.

В рамках модели HSB цвет формируется из трех составляющих: **H**ue (Тон), **S**aturation (насыщенность) и **B**rightness (Яркость). Графическое представление этой модели изображено на рис. 18.3.

Hue (Тон) — цветовой тон, характеристика цвета, определяемая длиной световой волны. Именно этот параметр определяет, к какой части спектра (для спектральных цветов) относится цвет. В графической интерпретации модели HSB цветовой тон характеризуется положением на цветовом круге (представляющем собой горизонтальное сечение цилиндра) и определяется величиной угла, который изменяется в диапазоне от 0 до 359 градусов. Красному цвету соответствует угол в 0 градусов. Отсчет углов ведется против часовой стрелки. Таким образом, цветовой круг разделен на 360 равных секторов, каждый из которых соответствует определенному цветовому тону.

Saturation (Насыщенность) — данный параметр характеризует концентрацию или *чистоту* цветового тона. Уменьшение насыщенности можно представить как добавление в цвет серого и ослабление его хроматической составляющей. Наоборот, максимальная степень насыщенности дает наиболее интенсивный (чистый) хроматический цвет. Значение параметра насыщенности задается в процентах и лежит в диапазоне от 0 до 100. Графически насыщенность представляется как расстояние от центральной вертикальной оси цилиндра до его края. На цветовом круге, образованном горизонтальным сечением цилиндра, равнонасыщенные цвета лежат на одной окружности. Чем дальше от центральной оси цилиндра находится точка цвета, тем выше степень его насыщенности. Точки, расположенные на центральной верти-

кальной оси цилиндра, обладают нулевой насыщенностью и представляют шкалу серого, то есть ахроматические цвета. Отметим, что при нулевой насыщенности цветовой тон считается неопределенным.

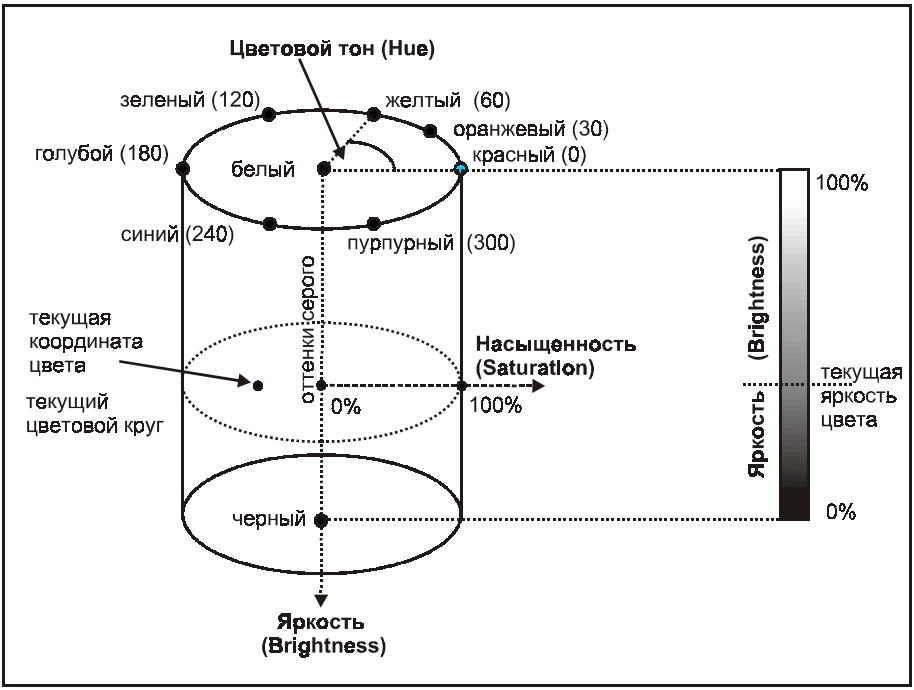


Рис. 18.3. Цветовое пространство модели HSB

Brightness (Яркость) — параметр, задающий степень освещенности цвета. При нулевой яркости освещение отсутствует, и любой цветовой тон превращается в черный цвет. Максимальная яркость соответствует хорошо освещенной цветной поверхности. Значение яркости задается в процентах, в диапазоне от 0 до 100. В цилиндрическом цветовом теле модели HSB яркость изменяется снизу вверх вдоль вертикальной оси. Основанию цилиндра соответствует нулевая яркость, верхней плоскости соответствует максимальная яркость — 100 процентов. Все цвета, лежащие на цветовом круге, полученном с помощью горизонтального сечения цилиндра, обладают одинаковой степенью яркости. При этом в верхней точке (с нулевой насыщенностью и максимальной яркостью) центральной оси цилиндра находится белый цвет (H, 0, 100).

В табл. 18.1 приведены цветовые координаты в моделях RGB и HSB для основных цветов.

Таблица 18.1. *Цветовые координаты*

Цвет	Цветовые координаты	
	Модель цвета RGB	**Модель цвета HSB**
Ахроматические цвета		
Белый (White)	0,0,0	H,0,0[1]
Черный (Black)	255,255,255	H,0,100
Хроматические цвета		
Красный (Red)	255,0,0	0,100,100
Оранжевый (Orange)	255,128,0	30,100,100
Желтый (Yellow)	255,255,0	60,100,100
Зеленый (Green)	0,255,0	90,100,100
Голубой (Cyan)	0,255,255	180,100,100
Синий (Blue)	0,0,255	240,100,100
Пурпурный (Magenta)	255,0,255	300,100,100

[1] Значение цветового тона не указано, поскольку при нулевой насыщенности он считается неопределенным. На практике при нулевой насыщенности можно использовать произвольный тон, например, 0 (красный цвет). — *Прим. автора.*

Перевод цвета из модели RGB в модель HSB и наоборот

Довольно часто на практике при решении поставленных задач цветовая модель RGB оказывается непригодной или ее применение оказывается сопряжено со значительными трудностями. Эта проблема часто имеет место в ситуации, требующей подхода к цвету с точки зрения человеческого восприятия. В качестве примера можно привести задачу создания анимации цветового перехода, представляющего последовательную смену основных цветов, представленных в табл. 18.1. Реализация этой задачи с помощью модели RGB потребует создания довольно громоздкого кода, который должен осуществлять приращение и уменьшение значений составляющих цвета в определенные моменты. Это связано с тем, что в модели RGB цветовой тон (как и насыщенность с яркостью) определяется совокупностью значений всех составляющих. В то же время для решения этой задачи с помощью модели HSB необходимо просто периодически с заданным шагом изменять значение параметра Hue от 0° до 359°. Очевидно, что эта процедура не вызывает трудностей в реализации. Другой пример, иллюстрирующий необходимость обращения к модели HSB, — сортировка некоего набора цветов по значению цветового тона. При работе с моделью HSB для выполнения такой сортировки достаточно лишь сравнить значения пара-

метра Hue для двух цветов. В модели RGB нельзя сравнить цветовой тон для двух произвольных цветов без выполнения дополнительных вычислений.

Методы класса `Color`, обеспечивающие возможности программного контроля цвета, используют только цветовую модель RGB. В связи с этим при необходимости работы с цветом, учитывающей такие характеристики, как цветовой тон, насыщенность и яркость, необходимо самостоятельно выполнить перевод цвета в модель HSB. Для того чтобы цвет, полученный в пространстве модели HSB, можно было синтезировать, необходимо выполнить обратный перевод из HSB в RGB.

График, представленный на рис. 18.4, изображает взаимосвязь моделей RGB и HSB, позволяющую переводить цветовые значения, заданные в пространстве одной модели, в пространство другой. По горизонтальной оси графика откладываются значения цветового тона, задаваемые в градусах (от 0 до 360) и расположенные в такой же последовательности, как и на цветовом круге. Таким образом, горизонтальная ось H представляет собой развернутый в полосу цветовой круг. По вертикальной оси графика откладываются значения интенсивности для цветовых составляющих RGB (от 0 до 255). Тремя линиями с различными стилями изображено изменение значений интенсивности на протяжении диапазона цветовых тонов для каждой составляющей RGB. Как видно из графика, цветовые составляющие на разных участках горизонтальной оси вносят различный вклад в формирование результирующего цвета. Диапазон изменения цветового тона (цветовой круг) можно разделить на 6 участков или секторов по 60° каждый. Так, например, в секторах от 300° до 0° и от 0° до 60° максимальный вклад в результирующий цвет вносит красная составляющая. Именно поэтому данным секторам соответствуют "красные" цвета: от пурпурного до желтого. По мере перемещения от 0° до 300° красный приближается к пурпурному за счет увеличения значения синей составляющей. А при перемещении от 0° до 60° происходит увеличение интенсивности зеленой составляющей, и результирующий цвет приближается к желтому.

Яркость цвета (B) определяется максимальной интенсивностью его составляющей. Таким образом, параметр Brightness (яркость) определяется по вертикальной оси графика как максимальное значение одной из трех цветовых составляющих RGB. На графике фактическому значению яркости соответствует положение верхней границы значений цветовых составляющих. Поскольку интенсивность обычно измеряется в диапазоне от 0 до 255, а яркость измеряется в процентах от 0 до 100, необходимо выполнить нормирование.

Насыщенность по графику определяется как процентное выражение расстояния от верхней границы интенсивности (соответствующей яркости) до нижней границы интенсивности (определяемой минимальным значением цветовой составляющей в цвете), от расстояния между верхней границей и нулем. Поскольку насыщенность отсчитывается от верхней границы вниз, ось насыщенности направлена сверху вниз. С учетом приведенных на графике условных обозначений выражение для вычисления насыщенности на основании значений интенсивности цветовых составляющих можно представить так:

$$S^{\%} = (\max - \min) / \max \cdot 100\%.$$

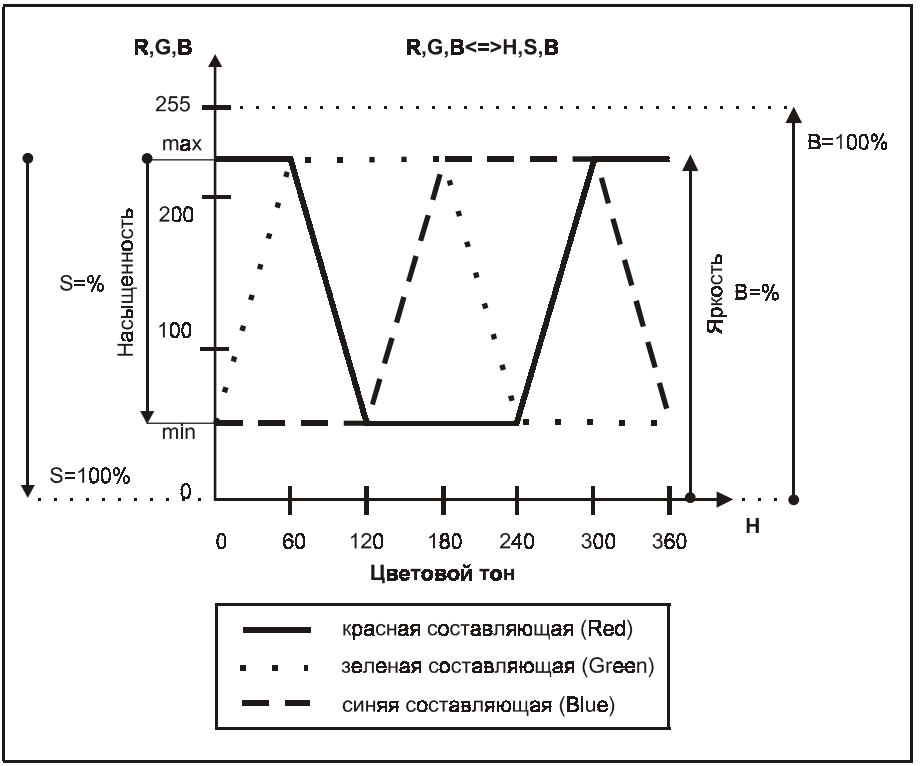


Рис. 18.4. Графическое представление перевода значений цвета для моделей RGB и HSB

Приведем общий алгоритм перевода значения цвета из модели RGB в модель HSB. Пусть имеется цвет, заданный тремя RGB-координатами. Требуется найти эквивалентное значение этого цвета в модели HSB.

1. Определяем яркость как значение цветовой составляющей, имеющей наибольшую интенсивность. На данном шаге яркость определяется числом от 0 до 255. Так, цвету с координатами (220, 50, 100) соответствует яркость 220.
2. Определяем разницу Delta между яркостью (максимальной интенсивностью) и минимальным значением интенсивности цветовой составляющей. В цвете с координатами (220, 50, 100) разница составит 220 − 50 = 170.
3. Для определения насыщенности при ненулевой яркости разделим значение, полученное на шаге 2, на значение яркости, полученное на шаге 1. Для выражения насыщенности в процентах полученное частное умножаем на 100. Если интенсивность каждой составляющей равна нулю, получаем черный цвет, насыщенность которого равна нулю.
4. Если значения трех составляющих цвета одинаковы (насыщенность равна нулю), этот цвет является ахроматическим (серым) и располагается на вертикальной центральной оси цилиндра. В этом случае параметру Hue (цветовой тон) присво-

им значение ноль (хотя, строго говоря, при нулевой насыщенности тон может быть любым).

Для хроматических цветов (с ненулевой насыщенностью) цветовой тон определяется в зависимости от того, какая составляющая имеет наиболее высокую интенсивность (см. рис. 18.4). Далее интенсивности цветовых составляющих обозначим буквами R, G и B.

- Если максимальное значение имеет красная составляющая, тон определяется по следующей формуле:

$$H = 60 \cdot (G - B) / delta.$$

- Если максимальное значение имеет зеленая составляющая, тон определяется по следующей формуле:

$$H = 120 + 60 \cdot (B - R) / delta.$$

- И, наконец, если максимальное значение имеет синяя составляющая, тон определяется по следующей формуле:

$$H = 240 + 60 \cdot (R-G) / delta.$$

Если значение тона оказывается отрицательным, прибавляем к нему 360°.

В листинге 18.1 приведена функция `RGBtoHSB()`, выполняющая перевод значения цвета из модели RGB в модель HSB. Исходный программный код функции сохранен в файле RGBtoHSB.as, который находится в папке Глава 18 на сопровождающем книгу компакт-диске. В приведенном сценарии используется директива `#include`, загружающая внешний файл со сценарием, содержащим объявление функций `min()` и `max()`, приведенных в листинге 15.1.

Листинг 18.1. Перевод значения цвета из модели RGB в модель HSB

```
function RGBtoHSB (rgb:Number):Array{
// Выделяем цветовые составляющие
var R=(rgb&0xff0000)>>16;
var G=(rgb&0xff00)>>8;
var B=(rgb&0xff);
// Яркость определяется как максимальная интенсивность составляющей (от 0
// до 255)
var Br=max(R, G, B);
// Определяем минимальное значение интенсивности составляющей в цвете
var minVal=min(R, G, B);
// Диапазон значений, соответствующих насыщенности цвета
var delta=Br-minVal;
// Если интенсивность каждой составляющей равна нулю, получаем черный
// цвет, насыщенность которого равна нулю
if (Br==0){
```

```
S=0;
}
// При ненулевой яркости насыщенность определяется как отношение разности
// максимальной и минимальной интенсивности составляющих цвета к его
// яркости (от 0 до 255), умноженное на 100
else {
S=delta / Br * 100;
}
// При нулевой насыщенности - ахроматический цвет, поэтому параметру Hue
// присвоим значение 0 (строго говоря, тон не определен)
if (S==0)H=0;
// При ненулевой насыщенности - хроматический цвет. Определяем тон как
угол(в градусах) на цветовом круге
else {
// Если красная составляющая имеет максимальную интенсивность, то цвето-
вой тон лежит между пурпурным и желтым
if (R==Br){
  H=60*(G-B)/delta;
}
else{
// Если зеленая составляющая имеет максимальную интенсивность,
// то цветовой тон лежит между голубым и желтым
  if (G==Br){
       H=120+60*(B-R)/delta;
       }
// Если синяя составляющая имеет максимальную интенсивность, то цветовой
// тон лежит между пурпурным и голубым
      else {
        H=240+60*(R-G)/delta;
                    }
               }
       }
// Если тон получился отрицательным, добавляем полный оборот для
// получения положительного значения
  if (H<0) H += 360;
// Возвращаем массив с полученными значениями Hue, Saturation
// и Brightness
  return ([Math.round(H), Math.round(S), Math.round(Br/255*100)]);
}
```

Для понимания механизма перевода значения цвета из модели HSB в модель RGB нужно обратиться к приведенному ранее рис. 18.4. Зная значение яркости, можно

определить максимальную интенсивность цветовой составляющей. Поскольку яркость обычно задается в диапазоне от 0 до 100, а значение интенсивности в диапазоне от 0 до 255, необходимо выполнить нормировку, приведя значение яркости к диапазону 0—255. Для этого используется следующая формула:

$$B^{0-255} = B^{\%} / 100 \cdot 255.$$

Минимальное значение цветовой составляющей можно легко определить, зная яркость и насыщенность. Для этого значение яркости принимается за 100%, и от этого значения вычисляется величина, процент которой соответствует проценту заданной насыщенности.

Значения цветовых составляющих RGB определяются как вертикальные координаты точек пересечения перпендикуляра, восстановленного из точки с заданным тоном на горизонтальной оси H, и линии, представляющей цветовую составляющую. Как видно из графика на рис. 18.2, существует четыре возможных варианта расположения точки, соответствующей цветовой составляющей.

- Если точка лежит на линии максимальной интенсивности, значение данной составляющей определяется значением яркости B (нормированным в диапазоне от 0 до 255).
- Если точка лежит на линии минимальной интенсивности, значение данной составляющей определяется значением разности яркости и насыщенности (S), вычисленной, как процент от величины яркости (B).
- Если точка лежит на возрастающем участке графика, значение интенсивности определяется с помощью следующего выражения:

  $$\texttt{inc} = B \cdot (1 - (S \cdot (1 - \texttt{resH}) / 100)),$$

  где B — яркость (в диапазоне от 0 до 255); S — насыщенность в процентах; `resH` — доля 60°-го сектора, соответствующая заданному значению тона H. Для определения значения можно использовать следующее выражение:

  $$\texttt{resH} = H/60 - \texttt{Math.floor}(H / 60).$$

  Так, например, если тон равен 150°, он находится в середине третьего от нуля 60°-го сектора. Таким образом, для данного тона величина параметра `resH` составляет 0,5 (`resH` = 150/60 — `Math.floor`(150/60) = 2,5 − 2).
- Если точка лежит на убывающем участке графика, значение интенсивности определяется с помощью следующего выражения:

  $$\texttt{dec} = B \cdot (1 - S \cdot \texttt{resH} / 100).$$

Фактическое местоположение точек, соответствующих интенсивностям цветовых составляющих RGB, определяется на основании заданного значения тона H. В соответствии с геометрической интерпретацией, приведенной на рис. 18.4, в программном коде выполняется последовательная проверка того, в каком 60°-ном секторе находится заданный тон. В зависимости от этого определяются значения составляющих RGB.

В листинге 18.2 приведена функция `HSBtoRGB()`, выполняющая перевод значения цвета из модели HSB в модель RGB. Исходный программный код функции сохра-

нен в файле HSBtoRGB.as, который находится в папке Глава 18 на сопровождающем книгу компакт-диске.

Листинг 18.2. Перевод значения цвета из модели HSB в модель RGB

```
function HSBtoRGB(H, S, Br) {
/* Нормируем Br, так чтобы значение яркости изменялось в диапазоне от 0
до 255 */
Br = Br/100 * 255;
/* При нулевой насыщенности - ахроматический цвет (оттенок серого). Все
цветовые составляющие имеют одинаковое значение, определяемое яркостью */
if (S==0){
  R=G=B=Br;
}
// Если насыщенность отлична от нуля - хроматический цвет
else {
/* Вычисляем долю 60°-го сектора цветового круга, отделяющую
заданный тон от границы этого сектора */
var resH = H/60 - Math.floor(H/60);
// Определяем минимальное значение цветовой составляющей
var bot=(Br*(1-S/100));
/* Определяем промежуточное значение цветовой составляющей на убывающей
части графика */
var dec=(Br*(1-(S*resH)/100));
/* Определяем промежуточное значение цветовой составляющей
на возрастающей части графика */
var inc=(Br*(1-(S*(1-resH)/100)));
/* В зависимости от принадлежности заданного тона к определенному сектору
цветового круга устанавливаем значения цветовых составляющих
в соответствии с графиком */
switch(Math.floor(H/60)){
   case 0: // Красный цвет
    R = Br;
    G = inc;
    B = bot;
               break;
   case 1: // Желтый цвет
    R = dec;
    G = Br;
    B = bot;
               break;
```

```
   case 2: // Зеленый цвет
    R = bot;
    G = Br;
    B = inc;
               break;
   case 3: // Голубой цвет
    R = bot;
    G = dec;
    B = Br;
               break;
   case 4: // Синий цвет
    R = inc;
    G = bot;
    B = Br;
               break;
   case 5: // Пурпурный цвет
    R = Br;
    G = bot;
    B = dec;
               break;
       }
}
// Возвращаем шестнадцатеричное значение цвета
return Math.round(R)<<16|Math.round(G)<<8|Math.round(B);
}
```

Создание объекта *Color*

Реализация программного управления цветом объекта с помощью возможностей класса `Color` включает два этапа.

- Создание объекта `Color`, представляющего цвет некоего объекта A.
- Вызов методов класса `Color` с помощью созданного на шаге 1 объекта `Color` для управления цветовыми характеристиками объекта A.

Таким образом, для управления цветовыми характеристиками объекта необходимо создать соответствующий ему объект `Color`. Объект `Color` создается как экземпляр класса `Color` с помощью оператора `new` и конструктора. В качестве параметра конструктору должна быть передана ссылка на объект, цвет и прозрачность которого будут контролироваться создаваемым объектом `Color`. Ссылка на созданный объект `Color` сохраняется в переменной.

```
var переменная:Color = new Color(объект)
```

Здесь *объект* — это экземпляр клипа, кнопка или текстовый блок, цветовые характеристики которого будут контролироваться с помощью объекта `Color`, ссылка на который сохраняется в переменной *переменная*.

Методы класса `Color` позволяют программным образом выполнять с цветом объектов операции, аналогичные применению цветовых эффектов экземпляров (color effects) в рабочей среде. К числу таких операций относятся следующие:

- окрашивание объекта сплошным цветом, при этом вся исходная цветовая информация удаляется. Данная операция реализуется с помощью методов `setRGB()/getRGB`;
- изменение текущих значений цветовых составляющих цвета объекта: красной, зеленой, синей и прозрачности. Данная операция реализуется с помощью методов `setTransform()/getTransform()`.

В то время как в рабочей среде цветовые эффекты экземпляров могут применяться только к экземплярам символов, класс `Color` позволяет управлять цветом и прозрачностью экземпляров клипов, кнопок, а также динамических и пользовательских текстовых блоков.

Окрашивание клипов. Методы *setRGB()* и *getRGB()*

Метод `setRGB()` окрашивает сопоставленный ему объект в заданный цвет. В результате применения этого метода исходная цветовая информация объекта полностью утрачивается, он равномерно окрашивается новым цветом. Данный метод имеет следующий синтаксис:

цветовой объект`.setRGB(`*цвет*`)`

Здесь *цветовой объект* — это объект `Color`, отвечающий за цветовые характеристики объекта.

Параметр *цвет* — это число, задающее цвет окрашивания на основе его RGB-значения. Обычно данный параметр задается шестнадцатеричным числом, хотя можно использовать и его десятичное представление. Параметр *цвет* может задаваться числовым литералом или выражением, возвращающим число.

Метод `setRGB()` окрашивает объект сплошным цветом, однако не влияет на его прозрачность. Если объект содержит прозрачные области (или градиент с прозрачностью), прозрачность сохраняется даже после окрашивания данным методом.

Следующий код окрашивает клип `mc` в красный цвет:

```
// Создаем объект Color для клипа mc
var col:Color = new Color(mc);
// Окрашиваем клип в красный цвет
col.setRGB(0xFFFFFF);
```

Приведенный ниже код окрашивает клип `mc` в цвет, выбранный случайным образом:

```
col.setRGB(Math.random()*0xFFFFFF);
```

Для того чтобы определить цвет, хранящийся в объекте `Color`, можно воспользоваться методом `getRGB()`. Он имеет следующий синтаксис:

```
цветовой_объект.getRGB()
```

Метод `getRGB()` возвращает RGB-значение цвета, представленного объектом *цветовой_объект*. Поскольку в Flash для счета применяются числа десятичной системы счисления, данный метод возвращает десятичное представление шестнадцатеричного числа. При необходимости (например, для вывода значения цвета в текстовое поле или в панель **Output**) возвращенное методом `getRGB()` десятичное значение можно преобразовать в шестнадцатеричное с помощью метода `toString()`.

Метод `getRGB()` возвращает значение цвета, установленного последним вызовом метода `setRGB()` или `setTransform()`. Если на момент вызова метода `getRGB()` в объекте `Color` не было установлено никакого цвета, метод возвращает значение 0 (черный цвет).

Следующая строка кода выводит шестнадцатеричное RGB-значение текущего цвета объекта `Color` в панель **Output**.

```
trace(col.getRGB().toString(16));
```

В качестве примера использования метода `setRGB()` рассмотрим создание анимации цвета объекта, цветовой тон которого будет постепенно изменяться по окружности цветового круга от 0° до 360°. В качестве окрашиваемого объекта будет выступать изображение хамелеона, причем окрашиваться будет только тело, а полосы будут сохранять исходный цвет (рис. 18.5). Для этого, нарисовав изображение хамелеона, мы конвертировали в клип только часть заливки между полосами на его теле. Полученный таким образом клип был назван `mc`.

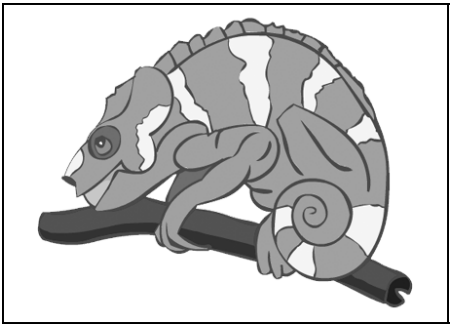


Рис. 18.5. Применение метода `setRGB()`

Для реализации поставленной задачи мы прибегнем к функции `HSBtoRGB()`, которая переводит значение цвета из модели HSB в модель RGB и представлена в листинге 18.2. Достигнуть требуемого результата значительно проще, используя модель HSB, поэтому параметры начального цвета мы зададим как тройку составляющих `hue`, `saturation` и `brightness`. Далее необходимо постепенно изменять значение параметра `hue` от 0 до 360 и конвертировать новое значение цвета из модели HSB в

модель RGB с помощью функции HSBtoRGB(). Полученное таким образом RGB-значение цвета в качестве параметра передается методу setRGB(), выполняющему окрашивание.

При нажатии на клип mc изменение цвета прекращается, а при повторном нажатии анимация цветового перехода возобновляется с текущего положения на цветовом круге. Для этих целей используется описанный алгоритм в сочетании с применением кнопки-флага. Программный код, реализующий описанную функциональность, приводится в листинге 18.3.

Листинг 18.3. Анимация перехода цвета по окружности цветового круга

```
// Включаем код функции для перевода HSB-значения в RGB
#include "HSBtoRGB.as"
// Создаем объект Color для управления цветом клипа mc
var col:Color = new Color(mc);
// Исходные значения тона, насыщенности и яркости
hue = 120; // Зеленый
saturation = brightness = 70;
// Автоматически запускаем цветовой переход
mc.onEnterFrame = function() {
       col.setRGB(HSBtoRGB(hue, saturation, brightness));
       // Если тон вышел за пределы 360, возвращаем его в ноль
       hue >= 360 ? hue = 0 : hue ++;
};
// При нажатии на клип mc прекращаем изменение цвета. При повторном
// нажатии продолжаем изменение цвета.
mc.onPress = function(){
if(!stopColor){
delete this.onEnterFrame;
stopColor = true;
}
else {
this.onEnterFrame = function() {
       col.setRGB(HSBtoRGB(hue, saturation, brightness));
       hue >= 360 ? hue = 0 : hue ++;
       };
       stopColor = false;
       }
};
```

Формирование каталога цветов на основе безопасной Web-палитры. Сортировка по тону

Хорошим примером по работе с шестнадцатеричным представлением цвета является формирование "безопасной" Web-палитры. "Безопасная" палитра состоит из 216 цветов, которые корректно (без диффузии) отображаются во всех браузерах на любых платформах при 8-битной глубине цвета. Актуальность применения этой палитры была чрезвычайно высока, когда большой процент пользователей сети Интернет работал с мониторами, не способными отобразить более 256 цветов. В рабочей среде Flash вывести Web-палитру в панель **Color Swatches** можно с помощью команды **Web 216** из ее контекстного меню.

"Безопасная" палитра получается путем комбинирования значений красного, зеленого и синего с шагом в 20% (33 в шестнадцатеричном представлении или 51 — в десятичном) от нулевого до максимального RGB-значения (FF или 255). Таким образом, данная палитра состоит из всех возможных комбинаций цветов, цветовые составляющие которых принимают значения 00 (0), 33 (51), 66 (102), 99 (153), CC (204) и FF (255). Подсчитав количество таких комбинаций, получим $6^3 = 216$ цветов.

В листинге 18.4 приводится функция `web216()`, возвращающая массив с RGB-значениями цветов "безопасной" Web-палитры. Принцип работы данной функции состоит в следующем. Создается массив, состоящий из строковых представлений шестнадцатеричных значений, соответствующих 20%-ным шагам RGB. Здесь намеренно используются строки, а не числа, поскольку с ними можно выполнить операцию конкатенации (соединения), получив нужную комбинацию значений цветовых составляющих. Далее с помощью двух последовательно вложенных циклов выполняется формирование строковых значений цвета путем конкатенации исходных строк. На каждой итерации строковое значение преобразуется в шестнадцатеричное число, представляющее RGB-значение цвета, которое сохраняется в массиве `ar`.

Листинг 18.4. Формирование цветовых значений "безопасной" Web-палитры

```
//Генерирует RGB-цвета на основе комбинации заданных значений
function web216(Void):Array{
/*Локальный массив, хранящий числовые значения (в строковом представлении),
комбинация которых даст набор из 216 "безопасных" цветов */
var col=new Array("00","33","66","99","cc","ff");
// Локальный массив для хранения "безопасных" цветов
var ar:Array = new Array();
  for (var r=0 ; r<=5 ; r++ ) {
   for (var g=0 ; g<=5 ; g++ ) {
     for (var b=0 ; b<=5 ; b++ ) {
              /* Добавляем в массив числовое шестнадцатеричное значение
              цвета, полученное путем соединения строк */
```

```
        ar.push(parseInt((col[r]+col[g]+col[b]), 16))
      }
    }
  }
return ar; // Возвращаем заполненный массив с 216 "безопасными" цветами
}
```

Набор цветовых значений, полученный с помощью функции `web216()`, не отсортирован по цветовому тону. Для большего удобства использования цветовой палитры нужно выполнить такую сортировку и сформировать таблицу цветовых образцов, расположенных в соответствии со значениями их цветового тона. Щелчок на образце должен выводить в текстовые поля его шестнадцатеричное RGB-представление, а также соответствующие этому значению HSB-составляющие. Программный код, выполняющий эту задачу, снабжен подробными комментариями и приведен в листинге 18.5. Полученный в результате его применения каталог цветов представлен на рис. 18.6.



Рис. 18.6. Каталог цветовых образцов "безопасной" Web-палитры, отсортированных по тону

Кратко опишем последовательность основных действий.

Поскольку сортировка цветов будет выполняться по тону, хроматические цвета нужно сортировать отдельно от ахроматических. В связи с этим создаются два массива: `colorSet` — для хроматических и `grays` — для ахроматических цветов.

Для каждого цвета создается объект с именем, состоящим из слова `colObj` и порядкового номера этого цвета. Объект `colObj` обладает четырьмя свойствами: `hue`, `saturation`, `brightness` и `rgb` для хранения HSB-представления и шестнадцатеричного RGB-представления цвета соответственно.

Объекты с ненулевым значением свойства `saturation` помещаются в массив `colorSet`, а объекты, представляющие цвета с нулевой насыщенностью, — в массив `grays`.

Выполняем сортировку в массивах colorSet и grays. Для сортировки хроматических цветов используем функцию сравнения compare(). Наибольшим приоритетом при этой сортировке обладает тон: если тон у сравниваемых цветов совпадает, они сортируются по насыщенности; при одинаковой насыщенности в качестве критерия используется яркость. После выполнения сортировки элементы из массива grays добавляются в конец массива colorSet.

Формирование таблицы цветовых образцов выполняется с применением алгоритма, описанного в листинге 12.13, путем динамического присоединения экземпляра клипа с идентификатором связи mc. Каждый вновь созданный экземпляр окрашивается цветом, индекс которого в отсортированном массиве colorSet совпадает с порядковым номером экземпляра. Отметим, что окрашиванию подвергается не сам экземпляр mc_i, а клип, расположенный на его монтажной линейке, имя которого состоит из слова swatch и порядкового номера. Это сделано для того, чтобы цветовой образец мог иметь рамку и тень, которые не должны окрашиваться.

Для каждого созданного экземпляра определяется метод обработчика события onPress, который выводит характеристики цвета в предназначенные для этого текстовые поля и сохраняет их в переменных, которые можно использовать при создании реального проекта.

Листинг 18.5. Формирование каталога цветовых образцов "безопасной" Web-палитры, отсортированных по тону

```
// Включаем функцию для формирования набора цветов WEB 216
#include "web216.as"
// Включаем функцию для перевода RGB-значения цвета в HSB
#include "RGBtoHSB.as"
//Сохраняем в массиве значения "безопасных" цветов
var colors:Array = web216();
//----------СОРТИРОВКА ЦВЕТОВ В ПАЛИТРЕ ПО ТОНУ----------------
/* Массив для хранения объектов со свойствами hue, saturation, brightness
и rgb, хранящих тон, насыщенность, яркость и исходное RGB-значение цветов
соответственно */
var colorSet:Array = new Array();
// Массив для ахроматических цветов (шкала серого)
var grays:Array = new Array();
for(i=0;i<colors.length;i++){
// Объект для хранения значения RGB и HSB значений цвета
this["colObj"+i]=new Object();
/* В переменной hsb сохраняем массив с HSB-составляющими исходного
RGB-цвета, полученный с помощью функции RGBtoHSB() */
var hsb:Array = RGBtoHSB(colors[i]);
// В свойстве hue сохраняем оттенок
this["colObj"+i].hue = hsb[0];
```

```
//В свойстве saturation сохраняем насыщенность
this["colObj"+i].saturation = hsb[1];
//В свойстве brightness сохраняем яркость
this["colObj"+i].brightness = hsb[2];
// Проверка на серый цвет (если насыщенность равна 0)
if(hsb[1]==0){
grays.push(this["colObj"+i]); /* Объект ахроматического цвета добавляем в
массив с серыми цветами */
}
else{
colorSet.push(this["colObj"+i]);/* Хроматический цвет помещаем в массив
colorSet */
}
this["colObj"+i].rgb = colors[i]; /* В свойстве rgb сохраняем исходное
шестнадцатеричное RGB-значение */
}
/* Сортируем серые по RGB-значению (все три компонента одинаковы, поэтому
можно сортировать просто по значению интенсивности) */
grays.sortOn("rgb", Array.NUMERIC);
/* Функция сравнения, используемая далее при сортировке хроматических
цветов в массиве colorSet. Критерии сортировки в порядке понижения
приоритета: Тон>Насыщенность>Яркость */
function compare(a, b){
if(a.hue==b.hue){ // Если тона совпадают
        if(a.brightness==b.brightness){ // Если яркости совпадают...
        return a.saturation-b.saturation; // ...сортируем по насыщенности
        }
        else{ // Сортируем по яркости
        return a.brightness-b.brightness;
        }
}
else { // Сортируем по тону
return a.hue-b.hue;
}
}
//Сортируем хроматические цвета по тону
colorSet.sort(compare, Array.NUMERIC);
//Соединяем серые и хроматические в единый массив colorSet
colorSet = colorSet.concat(grays);
```

```
//---ФОРМИРОВАНИЕ ТАБЛИЦЫ КЛИПОВ ДЛЯ ВЫВОДА ПАЛИТРЫ--------------
var nRow:Number =12;//Количество рядов
var nCol:Number = 18;//Количество столбцов
var xSpace:Number = 1;//Расстояние по X
var ySpace:Number = 1;//Расстояние по Y
var initX:Number = 45;//Координата левого верхнего клипа по X
var initY:Number = 40;//Координата левого верхнего клипа по Y
//Формирование таблицы
for (i=0; i<nRow; i++) {//Формирование строк
       for (j=0; j<nCol; j++) {//Формирование столбцов
       attachMovie("mc", "mc"+(j+nCol*i),j+nCol*i);
       this["mc"+(j+nCol*i)]._x =↳
(j)*(this["mc"+(j+nCol*i)]._width+xSpace)+initX;
       this["mc"+(j+nCol*i)]._y =↳
(i)*(this["mc"+(j+nCol*i)]._height+ySpace)+initY;
       }
}
//Окраска и обработка нажатия клипов
for(var i=0;i<colorSet.length;i++){
/* Красим только поле образца без рамки, то есть клип с именем swatch,
а не сам "mc"+i */
       this["col"+i]=new Color("mc"+i+".swatch");
       this["col"+i].setRGB(colorSet[i].rgb);// Окрашивание
/* При нажатии реагируют только окрашенные клипы, так как счетчик i
проходит цвета, сохраненные в массиве colorSet. Если клип не окрашен -
значит, цвета нет в массиве, значит, счетчик не примет значение большее,
чем число цветов (на случай, если количество образцов больше, чем число
цветов) */
this["mc"+i].onPress = function(){
//--------СОХРАНЕНИЕ И ВЫВОД ИНФОРМАЦИИ О ПРЕДСТАВЛЕНИИ ЦВЕТА В HSB------
/* Сохраняем HSB-компоненты выбранного цвета в переменных, которые можно
будет использовать в дальнейшем */
       hue = colorSet[this._name.slice(2)].hue;
       saturation = colorSet[this._name.slice(2)].saturation;
       brightness = colorSet[this._name.slice(2)].brightness;
       // Сохраняем значение цвета в RGB-представлении
       rgb = colorSet[this._name.slice(2)].rgb;
/* Вывод параметров цвета в текстовые поля при нажатии на образец. Имя
объекта Color генерируем на основании имени клипа, чей цвет требуется
узнать */
       hTxt.text = hue;
```

```
        sTxt.text = saturation;
        bTxt.text = brightness;
        rgbTxt.text = rgb.toString(16).toUpperCase();
        };
}
```

Цветовые трансформации. Методы *setTransform()* и *getTransform()*

Метод `setRGB()`, с которым мы познакомились ранее, позволяет равномерно окрашивать объекты в заданный цвет. При этом он не дает возможности управлять интенсивностью окрашивания или воздействовать только на определенную цветовую составляющую. С помощью метода `setRGB()` нельзя, например, увеличить интенсивность красной составляющей в изображении или убрать из голубого цвета синюю составляющую, чтобы получить зеленый.

Для выполнения избирательного и более гибкого окрашивания в классе `Color` имеется метод `setTransform()`. С помощью этого метода можно избирательно воздействовать на любую RGB-составляющую цвета, называемую цветовым каналом. Метод `setTransform()` позволяет не только выполнять цветовые трансформации, но и управлять степенью прозрачности объекта.

Типы цветовых трансформаций

Метод `setTransform()` позволяет реализовать два типа цветовой трансформации.

- *Мультипликативная* — данная трансформация реализуется путем умножения текущего значения цветовой составляющей на заданный множитель M, который может принимать как отрицательные, так и положительные значения. При работе с методом `setTransform()` множитель M задается значением от −100 до 100 процентов. Используя мультипликативную трансформацию, можно задать интенсивность каждой составляющей цвета, а также прозрачность в процентах от ее текущего значения. При этом результирующее значение для каждой цветовой составляющей и прозрачности рассчитывается по следующим формулам:

$$R' = (R \cdot M_{Red}) / 100;$$
$$G' = (G \cdot M_{Green}) / 100;$$
$$B' = (B \cdot M_{Blue}) / 100;$$
$$A' = (A \cdot M_{Alpha}) / 100,$$

  где R', G', B', A' — результирующие значения составляющих; R, G, B, A — исходные значения составляющих; M_i — значения множителя для каждой составляющей.

- *Аддитивная* — данная трансформация реализуется путем добавления к текущему значению цветовой составляющей фиксированного числового значения в диапазоне от −255 до +255. С помощью аддитивной трансформации можно усилить или ослабить цветовую составляющую или прозрачность, сместив ее значение на

указанную величину. Результирующее значение для каждой цветовой составляющей и прозрачности рассчитывается по следующим формулам:

$$R' = (R + A_{Red});$$
$$G' = (G + A_{Green});$$
$$B' = (B + A_{Blue});$$
$$A' = (A + A_{Alpha}),$$

где A_i — значения смещений для каждой составляющей.

При использовании обоих типов трансформации результирующий эффект рассчитывается как сумма мультипликативного и аддитивного эффектов. При этом необходимо иметь в виду, что если значение цветовой составляющей или прозрачности, полученное в результате применения трансформации, выходит за пределы диапазона 0—255, для него устанавливается минимальное или максимальное граничное значение. Так, если результат трансформации для определенной составляющей дает число, большее 255, ее значение устанавливается равным 255. Если результат меньше 0, значение составляющей устанавливается равным нулю. В рабочей среде Flash применение мультипликативной и аддитивной цветовых трансформаций наглядно представлено цветовым эффектом **Advanced**.

Применение методов *setTransform()* и *getTransform()*

Метод `setTransform()` является программным аналогом цветового эффекта экземпляров **Advanced** и имеет следующий синтаксис:

цветовой объект`.setTransform(`*объект трансформации*`)`

Здесь *цветовой объект* — это объект `Color`, отвечающий за цветовые характеристики представляемого им объекта.

Параметр *объект трансформации* — это объект `Object`, содержащий фиксированный набор свойств, задающих значения мультипликатора M_i и коэффициента смещения A_i для каждой цветовой составляющей и для составляющей прозрачности цвета объекта.

Объект трансформации должен обладать следующим обязательным набором свойств:

- `ra` — мультипликатор, задающий процентное значение красной составляющей. Принимает значение в диапазоне от −100 до 100;
- `rb` — коэффициент смещения для красной составляющей. Принимает значение в диапазоне от −255 до 255;
- `ga` — мультипликатор, задающий процентное значение зеленой составляющей. Принимает значение в диапазоне от −100 до 100;
- `gb` — коэффициент смещения для зеленой составляющей. Принимает значение в диапазоне от −255 до 255;
- `ba` — мультипликатор, задающий процентное значение синей составляющей. Принимает значение в диапазоне от −100 до 100;

- `bb` — коэффициент смещения для синей составляющей. Принимает значение в диапазоне от −255 до 255;
- `aa` — мультипликатор, задающий процентное значение прозрачности. Принимает значение в диапазоне от −100 до 100;
- `ab` — коэффициент смещения для прозрачности. Принимает значение в диапазоне от −255 до 255.

Примечание

Отметим, что при работе со свойством `_alpha` прозрачность объекта задается в процентах от 0 до 100, в то время как во внутренней реализации Flash прозрачность изменяется в диапазоне от 0 до 255, что и отражает свойство объекта трансформации `ab`. Именно с этим связано то, что при чтении свойства `_alpha` возвращаемое значение часто является дробным числом. Установка значения свойству `_alpha` объекта отражается на значении свойства `aa` его объекта цветовой трансформации. Собственно говоря, `_alpha` — это просто значение мультипликатора, задаваемое числом в диапазоне от 0 до 100.

Параметр *`объект трансформации`* может передаваться как литерал объекта или как выражение, возвращающее ссылку на объект с перечисленными выше свойствами.

Так, вызов метода `setTransform()` с приведенным ниже объектом трансформации максимально увеличивает значение синей составляющей, делая желтые цвета белыми:

```
var col:Color = new Color(mc);
col.setTransform({ra:100,rb:0,ga:100,gb:0,ba:100,bb:255,aa:100,ab:0});
```

Возможность задания отрицательных значений свойствам объекта трансформации, в частности, позволяет получить цвет дополнительный к исходному. *Дополнительным* называется цвет, расположенный на противоположном конце диаметра цветового круга. Координаты дополнительного цвета получаются по следующей формуле перевода цвета из модели RGB в модель CMY[1]:

$$C = 255 - R;$$
$$M = 255 - G;$$
$$Y = 255 - B.$$

Для того чтобы получить изображение в дополнительных цветах, необходимо передать методу `setTransform()` следующий объект трансформации:

```
{ra:-100,rb:255,ga:-100,gb:255,ba:-100,bb:255,aa:100,ab:0}
```

Результатом выполнения этой операции будет негативное изображение, в котором каждый цвет будет замещен дополнительным цветом.

Для определения параметров трансформации, установленных для объекта `Color`, можно воспользоваться методом `getTransform()`. Данный метод возвращает объект цветовой трансформации, содержащий цветовые настройки, хранящиеся в объекте

[1] Модель CMY получила свое название по начальным буквам трех лежащих в ее основе цветов: **C**yan — циан (бирюзовый), **M**agenta — пурпурный, **Y**ellow — желтый. Эта модель, дополненная черным цветом — blac**K**, широко применяется в полиграфии и называется CMYK. — *Прим. автора*.

Color. Структура этого объекта в точности соответствует структуре объекта, передаваемого в качестве параметра методу setTransform().

Метод getTransform() не принимает параметров и имеет следующий синтаксис:

```
цветовой_объект.getTransform()
```

Следует отметить, что в цветовом объекте, возвращаемом данным методом, отражаются не только параметры цветовой трансформации, которая задается методом setTransform(). На значения свойств этого объекта также оказывают влияние вызовы метода setRGB() и установка значения свойства _alpha для объекта, цветовые характеристики которого описываются рассматриваемым цветовым объектом. Так, например, окрашивание клипа в красный цвет с помощью метода setRGB() эквивалентно установке максимального смещения для красной составляющей и установке в ноль процентной величины зеленой и синей составляющих.

Если цветовые характеристики объекта не подвергались изменениям, по умолчанию его объект цветовой трансформации имеет следующий вид:

```
{ra:100, rb:0, ga:100, gb:0, ba:100, bb:0, aa:100, ab:0}
```

В качестве примера использования возможностей цветовых трансформаций рассмотрим создание панели, имитирующей применение эффекта **Advanced** в рабочей среде Flash. Применение цветовых трансформаций наиболее наглядно можно продемонстрировать на примере растрового изображения (которое необходимо конвертировать в клип). На рис. 18.7 представлен ролик с реализацией данной задачи.

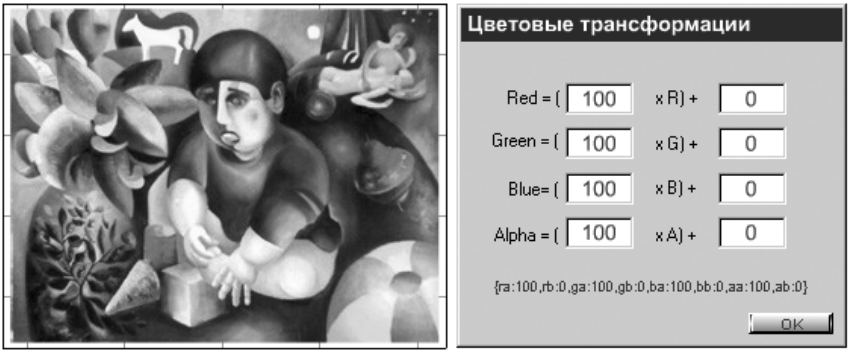


Рис. 18.7. Применение цветовой трансформации

Клип, содержащий экземпляр растрового изображения, имеет имя pic. Пользовательские текстовые поля, предназначенные для ввода значений параметров трансформации, имеют имена, состоящие из имени соответствующего свойства объекта трансформации и суффикса Txt. При нажатии на кнопку applyBut заданные настройки вступают в силу. При этом в динамическое поле tObjTxt выводится строка, содержащая литерал текущего объекта цветовой трансформации. Программный код, реализующий описанную функциональность, представлен в листинге 18.6.

Листинг 18.6. Управление цветовой трансформацией

```
// Создаем объект Color для клипа pic
var col:Color = new Color(pic);
// Функция, выполняющая цветовую трансформацию клипа pic на основании
значений, указанных в текстовых полях
function applyTransform() {
        // Создаем объект цветовой трансформации
        var tObj:Object = new Object();
        /* Установка значений свойств объекта цветовой трансформации на
        основании значений, указанных в текстовых полях */
        tObj.ra = Number(raTxt.text);
        tObj.rb = Number(rbTxt.text);
        tObj.ga = Number(gaTxt.text);
        tObj.gb = Number(gbTxt.text);
        tObj.ba = Number(baTxt.text);
        tObj.bb = Number(bbTxt.text);
        tObj.aa = Number(aaTxt.text);
        tObj.ab = Number(abTxt.text);
        col.setTransform(tObj);
}
// При нажатии на кнопку применяем трансформации
applyBut.onPress = function() {
        /* При вводе значений, выходящих за допустимый диапазон, выводим в
        текстовые поля граничные значения. Данная проверка не приводится в
        этом листинге, чтобы не загромождать программный код. Полный код
        программы можно найти на сопровождающем книгу компакт-диске */
        // Применяем трансформации
        applyTransform();
        // Сохраняем ссылку на объект трансформации в локальной
        // переменной
        var tObj:Object = col.getTransform();
        /* Массив для хранения строкового представления свойств объекта
        трансформации */
        var ar:Array = new Array();
        // Перечисляем свойства объекта tObj (в обратном порядке)
        for(i in tObj){
        // Заполняем массив
        ar.push(i + ":" + tObj[i]);
        }
```

```
        /* Реверсируем массив ar так, чтобы порядок следования его
        элементов соответствовал синтаксису объекта цветовой трансформации */
        ar.reverse();
        // Соединяем элементы массива в строку и выводим в текстовое поле
        tObjTxt.text = "{" + ar.join() + "}";
};
```

Создание панели синтеза цвета на основе модели HSB

Панель синтеза цвета на основе модели HSB позволяет синтезировать цвет на основании его тона и насыщенности, определяемых на цветовом круге, и по значению яркости, задаваемому с помощью вертикального ползунка. Внешний вид готовой панели представлен на рис. 18.8. Цветовой круг представляет собой горизонтальное сечение цилиндрического тела цветовой модели HSB (см. рис. 18.3), выполненное на высоте, заданной положением вертикального ползунка яркости.

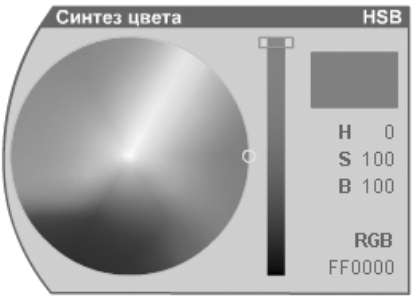


Рис. 18.8. Палитра синтеза цвета на основе модели HSB

Рассмотрим основные элементы, составляющие данную панель. Цветовой круг представляет собой растровое изображение, конвертированное в клип с именем `cCircle`. Круглый маркер на цветовом круге — это указатель текущего цвета, представляющий собой клип с именем `cPicker`. Перемещая его внутри цветового круга, можно регулировать тон и насыщенность цвета. Оба клипа `cCircle` и `cPicker` помещены на монтажную линейку клипа `cSpace` так, чтобы центр цветового круга совпадал с точкой его регистрации. Текущий цветовой образец (в правом верхнем углу) — это клип с именем `swatch`.

Полоса регулировки яркости состоит из следующих элементов.

- Ползунок — это клип с именем `scroller`.
- Сама полоса — это клип с именем `slideBar`, который на своей монтажной линейке содержит клип `colorSlideBar`, поверх которого наложен линейный градиент от черного (внизу) к полностью прозрачному (вверху). Клип содержит верти-

кальную полосу, окрашенную сплошным цветом. Цвет клипа `colorSlideBar` задается в соответствии с местоположением указателя на цветовом круге. Таким образом, полоса регулятора яркости всегда представляет собой градиент от текущего цвета к черному, наглядно изображая вертикальное сечение цилиндра цветового тела HSB в текущем положении указателя.

Обеспечение функциональности панели синтеза цвета включает две задачи.

- Возможность перемещения указателя внутри цветового круга и определение тона и насыщенности цвета в зависимости от его положения. Для того чтобы в любой точке цветового круга указатель был хорошо виден, мы будем окрашивать его противоположным цветом. Характеристики цвета, выбранные на цветовом круге, должны отражаться на цвете фона полосы регулировки яркости (то есть на цвете клипа `colorSlideBar`).
- Установка яркости выбранного цвета с помощью слайдера. Изменение степени яркости цвета должно отражаться на цветовом круге. По мере понижения яркости цветовой круг должен затемняться.

Возможность перемещения указателя реализуется с применением подхода, аналогичного способу, показанному в листинге 12.6. Тон определяется как угол поворота точки центра указателя относительно положительной полуоси *OX* с помощью метода `atan2()` (рис. 18.9). При выходе курсора мыши за пределы окружности указатель помещается на ее границу. Для расчета координат указателя в этой ситуации используется параметрическое уравнение окружности:

$$x = R \cdot \text{Math.cos}(a);$$

$$y = R \cdot \text{Math.sin}(a).$$

Здесь a — угол поворота точки центра указателя относительно положительной полуоси *OX*; R — радиус круга.

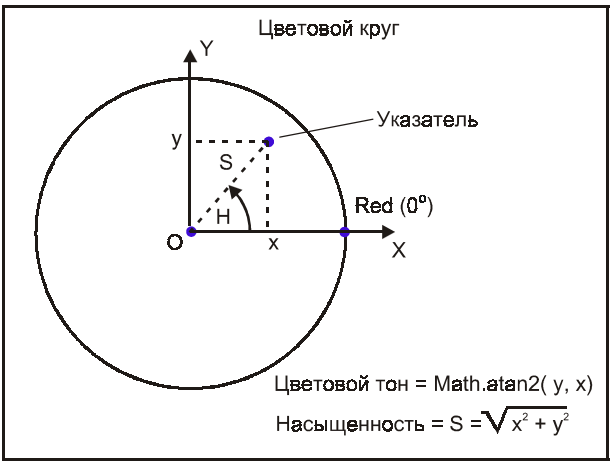


Рис. 18.9. Определение тона и насыщенности по координатам точки на цветовом круге

Насыщенность определяется как расстояние между центром указателя и центром цветового круга по теореме Пифагора.

Перемещение ползунка регулятора яркости в вертикальном направлении реализуется с применением метода `startDrag()`. Горизонтальные координаты области перетаскивания, переданные данному методу в качестве параметров, равны, и это не позволяет перемещать ползунок в горизонтальном направлении. Яркость вычисляется на основании расстояния от верхней границы ползунка до верха полосы регулировки яркости. Для затемнения цветового круга при изменении степени яркости используется метод `setTransform()`, которому в качестве параметра передается объект трансформации с одинаковыми значениями мультипликатора, равными заданной яркости синтезированного цвета.

Как и в случае с каталогом цветов "безопасной" Web-палитры, значения HSB-составляющих цвета и его шестнадцатеричное RGB-представление выводятся в текстовые поля и сохраняются в переменных. Эти переменные могут быть использованы в дальнейшем при реализации конкретного проекта, как показано в следующем разделе данной главы.

Программный код, реализующий создание панели синтеза цвета, снабжен подробными комментариями и представлен в листинге 18.7.

Листинг 18.7. Создание панели синтеза цвета на основе модели HSB, представленной в виде цветового круга

```
// Включаем функцию для перевода HSB-значения цвета в RGB
#include "HSBtoRGB.as"
// Создаем объект-приемник для событий мыши
var obj:Object = new Object();
// Регистрируем obj на получение сообщений от Mouse
Mouse.addListener(obj);
/* Задаем начальные значения цветовых составляющих HSB. Цвет по
умолчанию - красный */
var hue = 0; // Тон
var saturation = 100; // Насыщенность
var brightness = 100; // Яркость
var rgb = 0xFF0000; // RGB-значение
var swatchCol:Color = new Color(swatch); // Цвет клипа-образца
var circleCol:Color = new Color(cSpace.cCircle); // Цвет цветового круга
var slideBarCol:Color = new Color(slideBar.colorSlideBar); /* Цвет
подложки для полосы установки яркости */
var cPickerCol:Color = new Color(cSpace.cPicker); // Цвет указателя цвета
//---------УСТАНОВКА ЦВЕТОВОГО ТОНА И НАСЫЩЕННОСТИ----------------------
cSpace.cPicker.onPress = function() {// При нажатии на указатель цвета...
        obj.onMouseMove = function() {
```

```
// Перемещаем указатель за курсором
cSpace.cPicker._x = cSpace._xmouse;
cSpace.cPicker._y = cSpace._ymouse;
// Расстояния между центром указателя и центром круга по
вертикали и по горизонтали
var xDist = cSpace.cPicker._x - cSpace.cCircle._x;
var yDist = cSpace.cPicker._y - cSpace.cCircle._y;
// Расстояние между центром указателя и центром круга
var dist = Math.sqrt(Math.pow(xDist, 2) + Math.pow(yDist,
2));
// Угол указателя в радианах относительно центра круга
var a = Math.atan2(yDist, xDist);
// При выходе курсора за границы круга...
if (dist >= cSpace.cCircle._width / 2) {
       // Помещаем указатель на границу круга
       dist = 100; // Радиус круга = 100
       cSpace.cPicker._x = cSpace.cCircle._width / 2 *
       Math.cos(a);
       cSpace.cPicker._y = cSpace.cCircle._width / 2 *
       Math.sin(a);
}
// Вычисляем оттенок (от 0 до 360 градусов)
hue = a = Math.round(a/Math.PI*180);
// Переводим значение hue в границы от 0 до 360
if(a > 0) hue = 360 - a;
else hue = -a;
/* Насыщенность определяется расстоянием от центра круга до
центра указателя */
saturation = dist;
// Сохраняем текущий цвет в RGB-представлении
rgb = HSBtoRGB(hue, saturation, brightness)
// Красим клип - образец
swatchCol.setRGB(rgb);
/* Красим фон полосы установки яркости (только цвет
и насыщенность, яркость - максимальна) */
slideBarCol.setRGB(HSBtoRGB(hue, saturation, 100));
// Выводим координаты цвета в текстовые поля
hTxt.text = hue;
sTxt.text = Math.round(saturation);
bTxt.text = brightness;
```

```
            rgbTxt.text = rgb.toString(16).toUpperCase();
            /* Вычисляем противоположный цвет, то есть расположенный
            напротив на цветовом круге (для окрашивания указателя) */
            if(hue<180){
            oppositeHue = 180 + hue;
            }
            else {
            oppositeHue = hue - 180;
            }
            // Красим указатель противоположным цветом
            cPickerCol.setRGB(HSBtoRGB(oppositeHue, 100, 100));
            updateAfterEvent();
        };
};
// При отпускании кнопки мыши прекращаем перемещать указатель
obj.onMouseUp = function() {
        delete obj.onMouseMove;
};
//-----------РЕГУЛИРОВКА ЯРКОСТИ С ПОМОЩЬЮ ПОЛЗУНКА--------------------
scroller.onPress = function() {
        this.startDrag(false, slideBar._x, slideBar._y, slideBar._x,↓
        slideBar._y+slideBar._height);
        obj.onMouseMove = function(){
        // Устанавливаем значение яркости исходя из положения ползунка
        brightness = Math.round(100 - 100/slideBar._height*(scroller._y -↓
        slideBar._y));
        // Меняем яркость цветового круга (для этого задаем одинаковые
        // значения мультипликативной трансформации для каждого канала)
        circleCol.setTransform({ra: brightness, rb: '0', ga: brightness,↓
        gb: '0', ba: brightness, bb: '0', aa: '100', ab: '0'});
        // Вычисляем RGB для полученного оттенка
        rgb = HSBtoRGB(hue, saturation, brightness);
        // Красим клип-образец
        swatchCol.setRGB(rgb);
        // Выводим координаты цвета в текстовые поля
        bTxt.text = brightness;
        rgbTxt.text = rgb.toString(16).toUpperCase();
        };
};
```

```
// При отпускании мыши прекращаем перетаскивание ползунка
scroller.onRelease = function() {
        stopDrag();
        delete obj.onMouseMove;
};
```

Создание интерактивного цветового редактора

В качестве иллюстрации применения созданных ранее каталога цветов и панели синтеза цвета на основе модели HSB приведем пример создания интерактивного цветового редактора, позволяющего раскрашивать объекты в фильме.

Внешний вид конечного ролика представлен на рис. 18.10. Изображение фигуры девушки представляет собой набор из шести клипов с именами, состоящими из слова `clip` и порядкового номера от 1 до 6. Прямоугольник фона — это клип с именем `clip0`, который также может окрашиваться. Все клипы, являющиеся деталями изображения, являются клипами-кнопками. Щелчок на клипе равномерно окрашивает его в текущий цвет, образец которого выводится в правом верхнем углу панели синтеза цвета. Поскольку очертания фигуры (также являющиеся деталью изображения) имеют маленькую площадь, для окрашивания контура создана специальная кнопка `outlinesBut`, которая находится в правом верхнем углу.

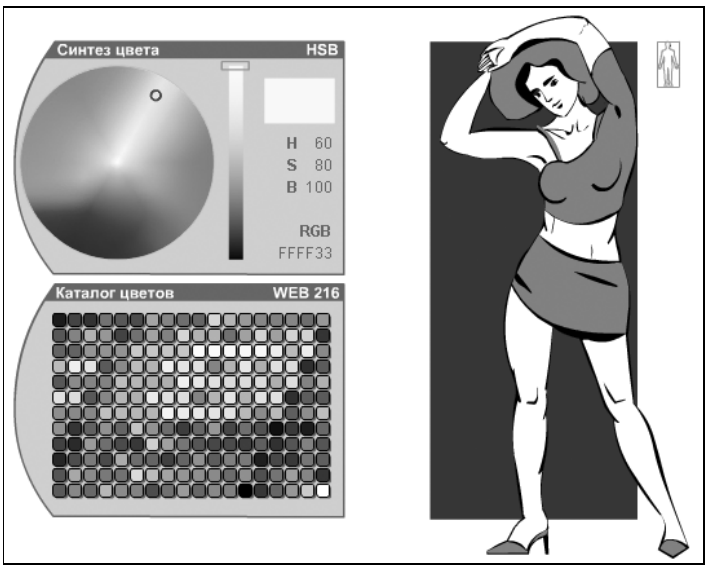


Рис. 18.10. Цветовой редактор

Панель синтеза цвета и каталог "безопасной" Web-палитры помещены в отдельный файл с именем colorMixer.swf, который динамически загружается в ролик цветового

редактора с помощью метода loadMovie(). Таким образом, в сам файл цветового редактора не требуется включать графический интерфейс и программный код, обеспечивающие представление и функциональность панелей работы с цветом. Их реализация полностью содержится в файле colorMixer.fla. Для корректного функционирования механизма синтезирования цвета необходимо согласовать работу каталога цветов с панелью синтеза цвета. При щелчке по цветовому образцу в каталоге цветов, необходимо позиционировать указатель на цветовом круге и установить ползунок яркости в соответствующее положение.

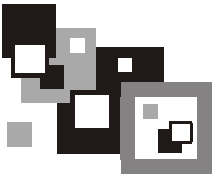
Связь между цветовыми панелями выполняется с помощью переменных hue, saturation, brightness и rgb. При определении цвета на цветовом круге или выборе его в каталоге цветов в этих переменных сохраняются координаты цвета в HSB и шестнадцатеричном RGB-представлении. При щелчке по цветовому образцу в каталоге необходимо выполнить изменение состояния элементов панели синтеза цвета в соответствии со значениями составляющих выбранного цвета. Фактически эта процедура является обратной для процедуры синтеза цвета, описанной ранее. Программный код комбинированной панели для работы с цветом, снабженный подробными комментариями, читатель может увидеть в файле colorMixer.fla, расположенном в папке Глава 18 на сопровождающем книгу компакт диске.

В листинге 18.8 приведен программный код, обеспечивающий функциональность цветового редактора.

Листинг 18.8. Применение панелей синтеза цвета для создания цветового редактора

```
// Загружаем панели синтеза цвета HSB и Web216
holder.loadMovie("colorMixer.swf", 1);
// Масштабируем их
holder._xscale = holder._yscale = 75;
// Создаем объекты Color для каждой раскрашиваемой детали рисунка и
обеспечиваем изменение цвета при нажатии на нее
for (var i = 0; i < 7; i++) {
       this["clipCol" + i] = new Color(this["clip" + i]);
       this["clip" + i].onPress = function() {
       /* rgb - это переменная, созданная загруженным файлом
       colorMixer.swf, в которой хранится шестнадцатеричное представление
       синтезированного цвета */
       this._parent["clipCol"+this._name.substr(-1)].setRGB(holder.rgb);
       };
}
// Кнопка, меняющая цвет очертаний рисунка
outlinesBut.onPress = function() {
       clipCol6.setRGB(holder.rgb);// clipCol6 - клип с контурами
                                   // изображения
};
```

Глава 19



Класс *Sound*. Управление звуком

Класс `Sound` предоставляет возможности программного контроля звуковой составляющей Flash-фильма. С его помощью можно программным образом осуществлять управление параметрами звука, настройка которых может быть выполнена вручную в рабочей среде. При этом класс `Sound` позволяет реализовать возможности контроля звука, доступные только для программной обработки. Приведем перечень основных возможностей по работе со звуком с применением программного инструментария класса `Sound`.

- Создание экземпляра звука на основе звукового файла, находящегося в библиотеке документа.
- Динамическая загрузка внешнего MP3-файла. Контроль процесса загрузки.
- Чтение ID3-тегов из загруженного внешнего MP3-файла.
- Запуск и остановка воспроизведения звуков.
- Контроль уровня звука (громкости).
- Регулирование звукового баланса каналов.

Механизм применения класса `Sound` похож на механизм использования класса `Color`. Для управления звуковым содержимым клипа необходимо создать объект `Sound`, представляющий характеристики всех звуков, прикрепленных к этому клипу, точно так же, как для контроля цветовых характеристик клипа, создается связанный с ним объект `Color`. Объект `Sound` может контролировать как звуковое содержимое конкретного клипа, так и все звуковое сопровождение фильма. С его помощью можно осуществлять программное управление звуками, помещенными на монтажную линейку документа вручную в рабочей среде, а также звуками, динамически прикрепленными к объекту (клипу, кнопке или текстовому блоку).

Создание объекта *Sound*

Объект `Sound` создается с помощью конструктора `Sound()` и оператора `new`. В зависимости от передаваемого конструктору параметра объект будет контролировать либо все звуковое сопровождение фильма, либо звуковое содержимое конкретного объекта.

Если при создании объекта `Sound` конструктор вызывается без параметров, объект будет ассоциирован со всеми звуками, используемыми в фильме. Такой объект на-

зывается *глобальным*, поскольку выполняемые с его помощью действия распространяются сразу на все звуковое сопровождение документа. Для создания глобального звукового объекта используется следующий синтаксис:

```
var переменная:Sound = new Sound()
```

Здесь *переменная* — это идентификатор переменной, в которой сохраняется ссылка на созданный конструктором объект Sound.

Для того чтобы контролировать один или несколько звуков независимо от остальных, необходимо связать объект Sound с объектом, к которому будут прикреплены (или на монтажной линейке которого находятся) данные звуки. Для этого конструктору Sound() в качестве параметра необходимо передать ссылку на объект, звуковое содержимое которого будет контролироваться с помощью создаваемого объекта Sound.

```
var переменная:Sound = new Sound(объект)
```

Параметр *объект* — это ссылка на экземпляр клипа, кнопки или текстового блока, с которыми будет ассоциирован объект Sound.

Если конструктору передается ссылка на экземпляр клипа, создаваемый объект Sound будет контролировать только звуки, помещенные на монтажную линейку данного клипа в рабочей среде или прикрепленные к нему динамически. Возможность передачи конструктору ссылки на текстовый блок (с которым в рабочей среде связать звук нельзя) обусловлена тем, что звук может быть динамически прикреплен к экземпляру текстового блока (хотя на практике такая возможность вряд ли может оказаться полезной).

Следует иметь в виду, что объект Sound не дифференцирует звуки, прикрепленные к ассоциированному с ним объекту. Это означает, что операции, производимые с объектом Sound, будут оказывать одинаковое влияние сразу на все представляемые им звуки. Для того чтобы избирательно задавать параметры каждому звуку объекта в отдельности, необходимо для каждого звука создавать свой объект Sound. С одним объектом (клипом, кнопкой или текстовым блоком) может быть ассоциировано любое количество объектов Sound. Конструктор Sound можно применять многократно, при каждом вызове передавая ему в качестве параметра ссылку на один и тот же объект. Ниже приведен пример, в котором создаются десять объектов, которые ассоциируются с одним клипом mc. В дальнейшем каждый из созданных объектов может осуществить динамическое присоединение звука из библиотеки или загрузку из внешнего файла и управлять его характеристиками независимо от остальных.

```
for (var i = 0; i < 10; i++) {
        // Создаем объект Sound для каждого звука
        this["s" + i] = new Sound(mc);
}
```

Отметим, что при ассоциировании объекта Sound с кнопкой или текстовым блоком можно реализовать далеко не все возможности управления звуком. Так, например, в этом случае нельзя управлять громкостью звука и балансом каналов. В связи с этим при динамическом присоединении звука или его загрузке из внешнего файла лучше

всего ассоциировать объект Sound с экземпляром клипа. Этот клип может быть создан как вручную в рабочей среде, так и динамически, например, с помощью метода createEmptyMovieClip().

Присоединение звука

Присоединение звука — это процедура, позволяющая поместить в фильм звук из библиотеки. Формально это очень напоминает присоединение экземпляра символа с помощью метода attachMovie(). Присоединение звука выполняется с помощью объекта Sound, причем каждый объект может присоединить только один звук. Для присоединения нескольких разных звуков или нескольких экземпляров одного и того же звука необходимо создать соответствующее количество объектов Sound.

Для того чтобы звук можно было динамически присоединить, его необходимо снабдить идентификатором связи (Linkage Identifier) с помощью диалогового окна **Linkage Properties** точно так же, как это делается с символом, экземпляры которого создаются с помощью метода attachMovie() *(см. разд. "Присоединение экземпляра. Метод attachMovie()" главы 12)*. Важно иметь в виду, что, как и в случае с символами, снятие флажка **Export in first frame** в диалоговом окне **Linkage Properties** предотвратит загрузку присоединяемого звука перед загрузкой первого кадра фильма. В этом случае необходимо в рабочей среде разместить экземпляр данного звука на монтажной линейке до появления сценария, осуществляющего присоединение, причем звук должен использовать синхронизацию типа **Event** или **Start**. В противном случае звук не будет загружен и не сможет быть присоединен.

Для присоединения звука используется метод attachSound(), который имеет следующий синтаксис:

звуковой объект.attachSound(*идентификатор связи*)

Здесь *звуковой объект* — это объект Sound, ассоциированный с каким-либо объектом в фильме.

Параметр *идентификатор связи* — это строка, содержащая идентификатор связи присоединяемого звука. Данный параметр может быть задан строковым литералом или выражением, возвращающим строку.

В результате присоединения звука он формально прикрепляется к объекту, звуковое содержимое которого контролируется объектом Sound, вызвавшим метод attachSound(). К одному объекту (например, клипу) можно прикрепить любое количество звуков, однако для каждого из них необходимо создать индивидуальный объект Sound. Если объект Sound уже выполнил присоединение звука, повторное присоединение приведет к удалению предыдущего звука и замене его новым.

Для того чтобы динамически присоединенный звук можно было контролировать из любой сцены (Scene) документа, этот звук должен быть прикреплен к основной монтажной линейке. Для этого при создании объекта Sound в качестве параметра конструктору можно передать ссылку на основную монтажную линейку либо создать глобальный объект Sound, вызвав конструктор без параметра.

Примечание

В результате присоединения звук не начинает автоматически воспроизводиться. Чтобы запустить его воспроизведение, необходимо использовать метод `start()`.

Следующий код создает объект `Sound` для клипа `mc` и присоединяет к нему звук из библиотеки с идентификатором связи `soundID`:

```
var s:Sound = new Sound(mc);
s.attachSound("soundID");
```

Управление воспроизведением звука

При работе со звуками в рабочей среде документа момент их запуска определяется местоположением ключевого кадра, в который помещен экземпляр звука. Момент остановки воспроизведения зависит от параметров синхронизации экземпляра звука и может определяться либо его продолжительностью (для событийных звуков), либо временем экспонирования ключевого кадра, содержащего экземпляр звука (для потоковых звуков). Кроме того, воспроизведение событийного звука в рабочей среде можно остановить с помощью другого экземпляра этого звука, задав ему синхронизацию типа Stop. Класс Sound позволяет обеспечить гораздо более гибкий контроль воспроизведения и остановки звуков, размещенных на монтажных линейках в рабочей среде.

Запуск и остановка воспроизведения звуков, динамически присоединенных или загруженных из внешних файлов, можно выполнить только программным образом.

Запуск воспроизведения

Для запуска воспроизведения динамически присоединенного или загруженного из внешнего файла звука используется метод `start()`. Этот метод может принимать два необязательных параметра и имеет следующий синтаксис:

звуковой объект`.start(`*смещение*`, `*зацикливание*`)`

Здесь *звуковой объект* — это объект `Sound`, представляющий звук, который требуется запустить.

Параметр *смещение* задает позицию в секундах, с которой начнется воспроизведение звука. Применение этого параметра эквивалентно установке точки начала звучания в окне **Edit Envelope** для звука, размещенного на монтажной линейке в рабочей среде. Так, если значение параметра равно 5, звук начнет сразу воспроизводиться с шестой секунды.

Параметр *зацикливание* позволяет указать, сколько раз звук должен быть последовательно воспроизведен. По умолчанию значение этого параметра принимается равным единице. Если звук воспроизводится более одного раза, то воспроизведение всех повторяющихся звуков начинается с позиции, заданной параметром *смещение*.

Необходимо отметить, что метод `start()` запускает воспроизведение только динамически присоединенных или загруженных звуков. Его действие не распространяет-

ся на звуки, помещенные на монтажную линейку в рабочей среде. Следующий код присоединяет звук и запускает его трехкратное воспроизведение, начиная со второй секунды:

```
var s:Sound = new Sound(mc);
s.attachSound("soundID");
s.start(2, 3);
```

Остановка воспроизведения

Для остановки воспроизведения звука используется метод `stop()`. С помощью этого метода можно останавливать как звуки, размещенные на монтажной линейке в рабочей среде, так и звуки, присоединенные или загруженные динамически. Результат вызова метода `stop()` зависит от того, вызывается он глобальным или локальным объектом `Sound`, а также от того, передается ли ему при этом параметр. Необходимо отметить, что метод `stop()` только останавливает воспроизведение запущенных звуков, но не предотвращает их повторный запуск. В общем случае синтаксис вызова метода `stop()` имеет следующий вид:

```
звуковой объект.stop(идентификатор связи)
```

Параметр *идентификатор связи* задает идентификатор связи динамически присоединенного звука, воспроизведение которого необходимо остановить. Данный параметр является необязательным и может быть опущен.

Рассмотрим различные варианты вызова метода `stop()` и производимые им действия.

- Если метод `stop()` вызывается глобальным объектом `Sound` без передачи ему параметра, выполняется остановка всех звуков, воспроизводящихся в документе. Это относится как к звукам, размещенным на монтажной линейке в рабочей среде (событийным и потоковым), так и к звукам, присоединенным или загруженным динамически.

  ```
  var s:Sound = new Sound();
  s.stop(); //Остановка всех запущенных звуков в фильме
  ```

- Если метод `stop()` вызывается глобальным объектом `Sound` с передачей ему в качестве параметра идентификатора связи присоединенного данным объектом звука, выполняется остановка только присоединенного звука.

  ```
  var s:Sound = new Sound();
  s.attachSound("soundID");
  s.stop("soundID"); //Остановка только звука с идентификатором soundID;
  ```

- Если метод `stop()` вызывается локальным объектом `Sound` без передачи ему параметра, результатом являются следующие действия.

  - Выполняется остановка всех воспроизводимых событийных (Event) звуков, расположенных на монтажной линейке клипа, звуковое содержимое которого представлено объектом `Sound`.

- Воспроизведение потоковых (Stream) звуков, расположенных на монтажной линейке клипа, звуковое содержимое которого представлено объектом `Sound`, прекращается только до того момента, пока воспроизводящая головка не перейдет к следующему кадру монтажной линейки этого клипа. При входе в новый кадр воспроизведение потоковых звуков клипа возобновляется.
- Выполняется остановка всех динамически присоединенных и загруженных звуков, прикрепленных к клипу, звуковое содержимое которого представлено объектом `Sound`. Если с одним и тем же клипом ассоциировано несколько объектов `Sound`, то вне зависимости от того, какой из их вызвал метод `stop()`, воспроизведение всех звуков клипа будет остановлено.

```
var s1:Sound = new Sound(mc);
s1.attachSound("soundID1");
var s2:Sound = new Sound(mc);
s2.attachSound("soundID2");
mc.onPress = function() {
    s1.start();
    s2.start();
};
stopBut.onPress = function() {
    s1.stop(); /* Остановка всех звуков на монтажной линейке клипа mc,
    а также прикрепленных к нему звуков soundID1 и soundID2 */
};
```

- Если метод `stop()` вызывается локальным объектом `Sound` с передачей ему в качестве параметра идентификатора связи присоединенного данным объектом звука, результатом являются следующие действия.
  - Выполняется остановка всех воспроизводимых событийных (Event) звуков, расположенных на монтажной линейке клипа, звуковое содержимое которого представлено объектом `Sound`.
  - Выполняется остановка динамически присоединенного звука с идентификатором связи, переданным методу `stop()` в качестве параметра.

Для остановки воспроизведения всех запущенных звуков в фильме вместо вызова метода `stop()` глобальным объектом `Sound` можно воспользоваться глобальной функцией `stopAllSounds()`. Данная функция останавливает воспроизведение всех запущенных звуков, размещенных на монтажной линейке в рабочей среде, а также звуков, динамически присоединенных и загруженных из внешних файлов. Как и в случае метода `stop()`, в результате применения функции `stopAllSounds()` остановка потоковых звуков выполняется только до того момента, пока воспроизводящая головка не перейдет к следующему кадру, в котором присутствует данный звук. При входе в новый кадр воспроизведение потокового звука возобновится.

Обработка окончания звука

В классе `Sound` имеется обработчик события `onSoundComplete`, которое генерируется после окончания звука. Событие генерируется только при окончании динамически присоединенных или загруженных звуков. С его помощью нельзя определить момент окончания звуков, размещенных на монтажной линейке в рабочей среде. Для контроля события `onSoundComplete` необходимо определить соответствующий метод обработчика для объекта `Sound`, осуществившего присоединение или загрузку звука.

В листинге 19.1 приведен пример использования обработчика события `onSoundComplete` для контроля окончания звука и его повторного запуска. Здесь приведена функция `loop()`, осуществляющая присоединение и бесконечное зацикливание звука. Функция принимает два параметра. Параметр `sound` — это строка, содержащая идентификатор связи присоединяемого из библиотеки звука, а параметр `offset` является необязательным и позволяет задать начальную позицию воспроизведения.

Листинг 19.1. Бесконечное зацикливание звука

```
function loop(:String, offset:Number):Void {
        s = new Sound(this);
        s.attachSound(sound);
        s.start(offset);
        s.onSoundComplete = function() {
                s.start(offset);
        };
}
```

Определение длительности и текущей позиции воспроизведения звука

Объект `Sound` обладает свойством, позволяющим определить длительность присоединенного или загруженного данным объектом звука. Это свойство называется `duration` и применяется только для чтения. Свойство `duration` хранит длительность звука в миллисекундах, причем его нельзя использовать для определения длительности звуков, размещенных на монтажной линейке в рабочей среде. Следующий код выводит в панель **Output** длительность динамически присоединенного звука в минутах и секундах с точностью до секунды (миллисекунды отбрасываем):

```
var s:Sound = new Sound();
s.attachSound("soundID1");
trace(Math.floor(s.duration/60000) + ":" +↵
Math.floor((s.duration % 60000)/1000));
```

Для определения текущей позиции воспроизведения звука используется свойство `position` объекта `Sound`. Данное свойство используется только для чтения и хранит

количество миллисекунд, отделяющих текущую позицию воспроизведения от начала звука. Свойство position позволяет определить текущую позицию воспроизведения только для звуков, присоединенных динамически или загруженных из внешнего файла. Для звуков, размещенных на монтажной линейке в рабочей среде, данное свойство принимает значение undefined. С помощью данного свойства можно создать кнопку паузы, последовательное нажатие которой приводит к запуску и остановке воспроизведения звука. Приведенный в листинге 19.2 сценарий демонстрирует создание кнопки воспроизведения/паузы с именем but для динамически присоединенного звука с идентификатором связи soundID. При остановке воспроизведения в переменной offset сохраняется текущая позиция звука. При последующем запуске звук запускается не с самого начала, а с позиции, в которой звук находился в момент остановки. Если звук закончился, нажатие кнопки but приводит к возобновлению воспроизведения звука сначала. Окончание звука фиксируется с помощью обработчика onSoundComplete. Для того чтобы внешний вид кнопки изменялся в зависимости от состояния звука, на монтажной линейке клипа but расположены два кадра, содержащие изображение кнопки во время паузы и воспроизведения. Эти кадры снабжены метками _play и _pause соответственно.

Листинг 19.2. Создание кнопки воспроизведения/паузы присоединенного звука

```
var s:Sound = new Sound(this);
s.attachSound("soundID");
but.stop();
but.onRelease = function() {
        if (!splay) {// Если воспроизведение остановлено
                /* При первом нажатии значение offset - не определено,
                поэтому звук запускается сначала */
                s.start(offset);
                splay = true;
                this.gotoAndStop("_pause");
        }
        else {// Если воспроизведение выполняется
                s.stop("soundID");
                splay = false;
                offset = s.position / 1000;
                this.gotoAndStop("_play");
        }
};
/* Если звук закончился, переключаем флаг, устанавливаем позицию
воспроизведения в начало и меняем вид клипа but */
s.onSoundComplete = function() {
        splay = false;
```

```
        offset = 0;
        but.gotoAndStop("_play");
};
```

Загрузка внешнего звукового файла

Встраивание звука в документ далеко не всегда оказывается приемлемым на практике. Препятствием этому может служить чрезмерное увеличение объема конечного файла и трудности при обновлении звукового сопровождения. Постановка задачи может требовать воспроизведения звука, расположенного в сети по заданному адресу, или звука, локально размещенного в файловой системе пользователя. Класс `Sound` располагает набором программных средств, позволяющих осуществлять динамическую загрузку цифрового звука, а также контролировать процесс его загрузки и момент ее окончания. Flash поддерживает динамическую загрузку файлов в формате MPEG1 Layer 3, при этом поддерживаются и более поздние версии этого формата MPEG2 и MPEG 2, 5, ориентированные на применение при низкой скорости передачи данных. Flash поддерживает кодирование MP3, использующее как постоянную (Constant bitrate), так и переменную (Variable bitrate) скорость передачи данных.

Загрузка звука в потоковом и событийном режимах

Динамическая загрузка MP3-файла осуществляется с помощью метода `loadSound()`. Данный метод позволяет реализовать два варианта загрузки: в потоковом режиме и в событийном режиме. При загрузке звука в потоковом режиме его воспроизведение может быть запущено еще до окончания его полной загрузки. При загрузке в событийном режиме звук начнет воспроизводиться только после того, как он полностью загрузится на компьютер пользователя. Метод `loadSound()` имеет следующий синтаксис:

звуковой объект`.loadSound(`*путь*`, `*режим*`)`

Здесь *звуковой объект* — это объект `Sound`, инициирующий загрузку звука из внешнего файла. Данный объект представляет загруженный звук и используется в дальнейшем для контроля его параметров. Важно отметить, что если с помощью объекта *звуковой объект* выполнялось присоединение звука из библиотеки документа, то в результате загрузки внешнего звукового файла присоединенный звук будет удален. После загрузки внешнего звукового файла объектом `Sound` он уже не может выполнять присоединение звука из библиотеки.

Параметр *путь* — это строка, задающая путь к MP3-файлу, который требуется загрузить. Звуковой файл может быть загружен с применением протоколов HTTP, FTP или file:// (для локально расположенных файлов). Если звуковой файл находится в том же домене, что и Flash-фильм, можно указать относительный путь.

Параметр *режим* — это булево значение, определяющее режим загрузки звукового файла. Если параметр *режим* принимает значение `true`, выполняется загрузка в по-

токовом режиме. Если параметр *режим* принимает значение `false`, выполняется загрузка в событийном режиме.

Важно иметь в виду, что при использовании потокового режима загрузки воспроизведение звука начинается автоматически, как только загрузится определенный объем информации звукового файла. Время между началом загрузки порции потокового звука и началом его воспроизведения называется *временем буферизации звука*. По умолчанию величина этого параметра составляет 5 секунд. Это означает, что при потоковом режиме загрузки воспроизведение звука начнется только после того, как будет загружена порция звука длительностью пять секунд. Если по причине нестабильности соединения или из-за медленного канала происходит прерывание воспроизведения, оно возобновляется только после загрузки звукового фрагмента, длительность которого равна времени буферизации.

Для того чтобы задать время буферизации потоковых звуков (как размещенных на монтажной линейке вручную, так и динамически загруженных в потоковом режиме), используется глобальное свойство `_soundbuftime`. Данное свойство в качестве значения принимает число, задающее время буферизации в секундах для всех потоковых звуков, используемых в фильме.

В следующем сценарии выполняется динамическая загрузка звука в потоковом режиме с временем буферизации, составляющим 10 секунд. Кнопки `stopBut` и `playBut` используются для остановки и запуска звука соответственно.

```
_soundbuftime = 10
var s:Sound = new Sound(this);
s.loadSound("Ebony_Samba.mp3", true);
stopBut.onPress = function() {
       s.stop();
       pos = s.position;
};
playBut.onPress = function() {
       s.start(pos/1000);
};
```

Для того чтобы воспроизведение звука, загружаемого в потоковом режиме, не начиналось автоматически, нужно после вызова метода `loadSound()` вызвать метод `stop()`.

```
var s:Sound = new Sound(this);
s.loadSound("BWV 856.mp3", true);
s.stop(); // Останавливаем звук по умолчанию
```

Контроль состояния загрузки звука

Класс `Sound` предоставляет возможности контроля состояния загрузки звука. Для этих целей используются методы, названия и принципы применения которых совпадают с методами, применяемыми для контроля состояния загрузки клипов. Метод

getBytesTotal() возвращает объем загружаемого в объект Sound звука в байтах. Метод getBytesLoaded() возвращает загруженный на момент его вызова объем звукового файла в байтах. Оба метода вызываются с помощью объекта Sound, инициировавшего загрузку. Данные методы можно применять вне зависимости от используемого режима загрузки (потоковый или событийный).

Примечание

Файлы MP3, загружаемые в событийном или потоковом режимах, кэшируются браузером, что существенно ускоряет их повторную загрузку.

Отметим, что при использовании потокового режима загрузки свойство duration принимает значение, указывающее продолжительность загруженного фрагмента звука в миллисекундах. При событийной загрузке значение свойства duration остается равным нулю до тех пор, пока звук полностью не загрузится.

В листинге 19.3 приведен сценарий графического предзагрузчика для звука, загружаемого в потоковом режиме. Помимо процента загруженной информации мы будем выводить продолжительность загруженного фрагмента звука. При создании данного предзагрузчика используется подход, аналогичный рассмотренному в листинге 12.16.

Листинг 19.3. Предварительный загрузчик потокового звука

```
var s:Sound = new Sound(this);
s.loadSound("Ebony_Samba.mp3", true);
var sndTotal:Number = s.getBytesTotal();
// Функция для преобразования миллисекунд в минуты и секунды
function convert(s:Number):String{
return (Math.floor(s/60000) + ":" + Math.floor((s % 60000)/1000));
}
this.onEnterFrame = function() {
        var sndLoaded:Number = s.getBytesLoaded();
        if (sndLoaded < sndTotal) {
                var sndStatus = Math.round(sndLoaded / sndTotal * 100);
                statusBar._xscale = sndStatus;
                // Выводим состояние загрузки в процентах
                statusTxt.text = sndStatus + "%";
                // Выводим загруженную продолжительность
        durationTxt.text = convert(s.duration);
        }
        else {
                statusTxt.text = "";
```

```
            delete sndTotal;
            delete this.onEnterFrame;
        }
};
```

Проверка окончания загрузки звука

Для определения момента окончания загрузки внешнего звукового файла можно воспользоваться обработчиком события `onLoad`. Это событие генерируется объектом `Sound` в момент завершения загрузки или в результате ее прекращения по тем или иным причинам. Обработка данного события выполняется с помощью метода обработчика, которому может быть передан параметр, определяющий успешность выполнения загрузки. Метод обработчика события `onLoad` имеет следующий синтаксис:

```
звуковой объект.onLoad = function(success){
блок кода;
};
```

Параметр *success* принимает булево значение `true`, если звук успешно загрузился, и значение `false`, если звук загрузиться не смог. Данный параметр передается методу обработчика события в момент его вызова вне зависимости от того, был ли указан формальный параметр при его объявлении или нет. Причиной неудачи при загрузке звука может служить неверное указание пути к звуковому файлу или недоступность сервера, на котором он расположен.

Примечание

Обработчик `onLoad()` должен быть определен перед вызовом метода `loadSound()`.

В следующем примере выполняется динамическая загрузка звука в событийном режиме. Воспроизведение звука запускается автоматически, как только он полностью загрузится. В случае невозможности загрузки в панель **Output** выводится соответствующее сообщение.

```
var s:Sound = new Sound(this);
s.onLoad = function(success:Boolean) {
        if (success) {
                s.start();
        }
        else {
                trace("Ошибка при загрузке файла");
        }
};
s.loadSound("Ebony_Samba.mp3", false);
```

Поддержка ID3-тегов

Формат MP3 предполагает возможность хранить в файле не только данные, позволяющие воспроизвести аудиозапись, но и текстовую информацию с описанием ее содержания, характеристик и различных параметров. Эти данные могут быть размещены вначале или в конце MP3-файла и называются ID3-тегами. По мере развития технологии ID3-тегов были разработаны несколько версий, ставших широко используемыми. Flash поддерживает ID3-теги версий ID3 1.0, ID3 1.1, ID3 2.3 и ID3 2.4.

ID3-тег первой версии располагается в конце файла и имеет фиксированный размер, составляющий 128 байтов. Этот тег включает информацию о названии песни (30 байтов), исполнителе (30 байтов), альбоме (30 байтов), годе выпуска (4 байта), содержит комментарии (30 байтов) и цифровой код жанра (1 байт)[1].

ID3-тег второй версии располагается в начале файла и позволяет сохранять значительно больший объем информации, описывающей содержимое файла. Такой тег состоит из отдельных элементов, называемых фреймами. Каждый фрейм описывает определенную характеристику содержимого файла. Вторая версия ID3-тегов позволяет расширять набор стандартных тегов без опасности некорректного прочтения, поскольку анализатор тегов игнорирует неизвестную ему информацию. Теги второй версии поддерживают кодировку Unicode, что дает возможность применения кириллицы и, в частности, создания комментариев на нескольких языках. Flash распознает только теги, использующие кодировку UTF-8.

Flash позволяет выполнять чтение ID3-тегов в присоединенных или динамически загруженных MP3-файлах. Flash не дает возможности осуществлять запись и редактирование тегов, для этого необходимо использовать специальное программное обеспечение. Доступ к тегам выполняется с помощью объекта `id3`, представляющего собой свойство объекта `Sound`. При этом используется следующий синтаксис:

```
Sound.id3.тег
```

Здесь *`тег`* — это свойство, хранящее значение соответствующего ID3-тега.

В табл. 19.1 приведен перечень свойств объекта `id3`, соответствующих поддерживаемым в Flash тегам версии ID3v2.0.

Таблица 19.1. *Свойства, соответствующие ID3v2.0-тегам*

Свойство	Описание	Свойство	Описание
`COMM`	Комментарий	`TOFN`	Имя исходного файла (в некоторых ситуациях на имя файла накладываются технологические ограничения и оно не может быть произвольным, поэтому имя распространяемой версии файла может отличаться от оригинального)

[1] Оставшиеся три байта используются для создания подписи, обозначающей местоположение ID-тега в файле. — *Прим. автора.*

Таблица 19.1 *(продолжение)*

Свойство	Описание	Свойство	Описание
TALB	Название альбома/фильма/шоу	TOLY	Автор стихов/текста
TBPM	Кол-во тактов в минуту. Рассчитывается как частное от количества тактов на длительность аудиофайла	TOPE	Первый исполнитель
TCOM	Композитор	TORY	Год первого исполнения
TCON	Тип содержимого	TOWN	Владелец файла/лицензии
TCOP	Информация об авторских правах	TPE1	Солист/ведущий исполнитель
TDAT	Дата создания	TPE2	Группа/оркестр/аккомпанемент
TDLY	Интервал между композициями	TPE3	Дирижер
TENC	Кодер или организация, выполнившая кодировку	TPE4	Автор обработки/интерпретации / ремикса
TEXT	Автор стихов/текста	TPOS	Номер части (например, 1/2 — первая часть из двух)
TFLT	Тип файла (например, MPEG /2.5)	TPUB	Издатель, студия записи
TIME	Время создания	TRCK	Номер дорожки
TIT1	Категория музыкального произведения (например, фортепианный концерт, сейшн и т. д.)	TRDA	Дата записи
TIT2	Название произведения	TRSN	Название интернет-радиостанции
TIT3	Данные о произведении (например, op.8 или performed at the Royal Albert Hall)	TRSO	Владелец интернет-радиостанции
TKEY	Тональность (например, A (ля), B (си), Eb (ми-бемоль), D# (ре-диез))	TSIZ	Размер аудиофайла в байтах
TLAN	Языки, используемые в записи; указываются трехзначным кодом в соответствии со стандартом ISO-639-2	TSRC	Код ISRC (международный стандарт записи)

Таблица 19.1 *(окончание)*

Свойство	Описание	Свойство	Описание
TLEN	Время звучания	TSSE	Оборудование, его параметры настройки и программное обеспечение, использованные при оцифровке и кодировании звука
TMED	Источник, из которого был получен звук (например, TV — телевидение, RAD — радио, CD — компакт-диск, DVD)	TYER	Год записи
TOAL	Оригинальное название альбома/фильма/шоу	WXXX	Интернет-ссылка, связанная с данным файлом

Поскольку ID3-теги первой версии (ID3v1.0) отличаются от тегов второй версии, в Flash для их чтения применяются свойства, отличные от приведенных выше. Это также связано и с тем, что поддержка первой версии тегов появилась еще в шестой версии, в то время как вторая версия тегов начала поддерживаться только с выходом седьмой версии Flash. Для обеспечения обратной совместимости в случае отсутствия в файле тегов первой версии и при наличии соответствующих тегов второй версии Flash копирует свойства, соответствующие ID3-тегам второй версии, в свойства, соответствующие первой версии. Таким образом, если в сценарии используются свойства, читающие теги первой версии, они будут корректно работать даже с MP3-файлами, использующими более позднюю версию тегов. В табл. 19.2 представлен перечень свойств объекта `id3`, соответствующих тегам первой версии, и парные им свойства для второй версии тегов из числа свойств, приведенных в табл. 19.1.

Таблица 19.2. *Свойства для чтения ID3v1.0-тегов и соответствующие им свойства для чтения ID3v2.0-тегов*

ID3 1.0 тег	ID3 2.0 тег	Описание
comment	COMM	Комментарий
album	TALB	Название альбома
genre	TCON	Жанр
songname	TIT2	Название песни
artist	TPE1	Имя исполнителя
track	TRCK	Номер дорожки
year	TYER	Год записи

Следующий код присоединяет звук в формате MP3 и выводит в панель **Output** название исполняемой композиции:

```
var s:Sound = new Sound(this);
s.attachSound("sound");
trace(s.id3.songname);
```

Как было сказано ранее, в зависимости от версии ID3-тегов, они могут находиться в начале или в конце файла. Это означает, что информация о содержимом MP3-файла может поступить либо в самом начале его загрузки (при использовании второй версии), либо только в конце (при использовании первой версии). Для того чтобы отследить момент поступления информации о загрузке тегов, можно воспользоваться обработчиком события `onID3`, которое генерируется после окончания загрузки тегов. Для применения соответствующего метода обработчика используется следующий синтаксис:

```
звуковой объект.onID3 = function(){
блок кода;
};
```

Отметим, что если MP3-файл содержит теги обеих версий, то при его загрузке событие `onID3` будет сгенерировано дважды.

Приведенный ниже код создает текстовые поля и выводит в них перечень всех имеющихся в динамически загружаемом файле ID3-тегов. Вывод тегов осуществляется только после окончания их загрузки.

```
var s:Sound = new Sound(this);
// Координата первого текстового блока по Y
var initY:Number = 0;
/* Сохраняем ссылку на текущую монтажную линейку, чтобы в дальнейшем
обратиться к ней из тела метода обработчика onID3() */
var _this = this;
s.onID3 = function() {
       for (tag in s.id3) {
       _this.createTextField(tag + "Txt", initY, 20, initY * 15, 0, 0);
       initY++;
       _this[tag + "Txt"].autoSize = true;
       _this[tag + "Txt"].text = tag + ": " + s.id3[tag];
       }
};
s.loadSound("Ebony_Samba.mp3", true);
```

Управление уровнем звука

Класс `Sound` предоставляет возможность динамически управлять уровнем звука или громкостью звукового содержимого Flash-фильма. Для этих целей используются методы `setVolume()` и `getVolume()`, позволяющие соответственно устанавливать уровень звука и получать текущее значение громкости звукового содержимого, представляемого конкретным объектом `Sound`.

Метод `setVolume()` имеет следующий синтаксис:

звуковой объект`.setVolume(`*громкость*`)`

Параметр *громкость* задает устанавливаемое значение громкости звука в процентах от максимальной громкости и принимает значения в диапазоне от 0 до 100.

Метод `getVolume()` не принимает параметров и возвращает текущее значение громкости, установленное для вызвавшего его объекта `Sound`. По умолчанию значение громкости составляет 100%.

Данные методы могут использоваться для управления громкостью как звуков, размещенных на монтажной линейке в рабочей среде, так и присоединенных или загруженных динамически. При этом необходимо отметить, что установка громкости для объекта `Sound`, ассоциированного с клипом, будет применена ко всем звукам, размещенным на монтажной линейке этого клипа, а также на линейках вложенных в него клипов. Установка громкости для глобального объекта `Sound` приведет к применению этого значения ко всем звукам, используемым в документе.

Примечание

Установка уровня звука для объекта `Sound` не приводит к выравниванию значений громкости представляемого им звукового содержимого. Если звуки имеют разную громкость, то соотношение их уровней сохраняется и после изменения значения громкости. Метод `setVolume()` просто изменяет громкость всех звуков, представляемых объектом `Sound`, на одно и то же значение.

В качестве примера применения возможностей метода `setVolume()` приведем функцию `echo()`, присоединяющую звук и запускающую его воспроизведение с эффектом эха. Наша функция принимает три параметра:

- `soundID` — строка, задающая идентификатор связи звука, который должен быть присоединен;
- `repeat` — количество повторов звука (0 — отсутствие эха);
- `delay` — временной интервал в миллисекундах между воспроизведениями звука.

Принцип действия данной функции заключается в следующем. Создается пустой клип, выполняющий роль "контейнера" для звука. Затем в зависимости от заданного количества повторов эха для каждого из них создается индивидуальный объект `Sound`. Это необходимо сделать в связи с тем, что громкость эха должна постепенно уменьшаться (то есть при каждом повторном запуске звука его уровень должен становиться ниже), однако громкость звука, запущенного раньше него, меняться не должна. Если бы мы периодически запускали звук, присоединенный

к одному-единственному объекту Sound, то изменение громкости всякий раз отражалось бы на всех звуках. Далее необходимо инициировать периодический запуск звука для каждого объекта Sound требуемое количество раз и с заданным интервалом, постепенно понижая громкость каждого последующего звука. Для этого используется внутренняя функция setEcho(), периодический вызов которой выполняется с помощью функции setInterval(). Реализация функции echo() представлена в листинге 19.4.

Листинг 19.4. Создание эффекта эха

```
function echo(soundID:String, repeat:Number, delay:Number):Void {
// Создаем клип, к которому будет прикреплен звук
this.createEmptyMovieClip("soundHolder", this.getNextHighestDepth());
              for (var i = 0; i < repeat; i++) {
              // Создаем объект Sound для каждого повтора звука
              this["s" + i] = new Sound(soundHolder);
              // Присоединяем звук
              this["s" + i].attachSound(soundID);
       }
       s0.start(); // Запускаем исходный звук
       var counter = 0; // Счетчик количества воспроизведений
       // Функция, выполняющая повторение звука
       function setEcho(obj:Object) {
              // Запускаем повторный звук
              obj["s" + counter].start();
              /* Понижаем громкость звука эха для каждого повтора
              пропорционально его номеру */
              obj["s" + counter].setVolume(100 - 100 / repeat *
              counter);
              // Обновляем счетчик количества воспроизведений
              counter == (repeat - 1) ? clearInterval(sInt) : counter++;
       }
       // Выполняем периодический запуск звука с заданным интервалом
       sInt = setInterval(setEcho, delay, this);
}
// Запускаем звук с применением эха при нажатии левой кнопки мыши
this.onMouseDown = function() {
       echo("sound", 5, 10);
};
```

Управление балансом каналов

Наряду с управлением уровнем громкости звука класс `Sound` позволяет осуществлять контроль баланса каналов. Баланс каналов или *панорамирование* (panning) — это параметр, отвечающий за распределение уровня громкости по каналам (динамикам) звуковоспроизводящего устройства. По умолчанию соотношение громкостей левого и правого каналов устанавливается равным единице (то есть громкости равны), однако в некоторых ситуациях возникает необходимость в изменении этого соотношения. Управление балансом каналов реализуется с помощью метода `setPan()`, имеющего следующий синтаксис:

```
звуковой_объект.setPan(баланс)
```

Здесь *звуковой_объект* — это объект `Sound`, к звуковому содержимому которого применяется метод `setPan()`. Следует отметить, что данный метод, как и метод `setVolume()`, оказывает воздействие на все звуки, представляемые объектом `Sound`. Эти методы отчасти похожи, поскольку оба управляют характеристикой громкости, однако принципиальное различие заключается в том, что метод `setVolume()` в равной степени изменяет громкость левого и правого каналов, а метод `setPan()` задает распределение громкостей для левого и правого каналов. Применение метода `setVolume()` не влияет на распределение громкостей по каналам, а просто увеличивает или уменьшает уровень для каждого канала на одинаковую величину. В то время как метод `setPan()` позволяет изменять громкость одного канала за счет другого.

Параметр *баланс* — это число, которое задает распределение уровня звука между каналами. Данный параметр принимает значения в диапазоне от −100 до 100. Отрицательные значения соответствуют смещению в сторону левого канала, положительные значения соответствуют смещению в сторону правого канала. Нулевое значение параметра *баланс* соответствует равномерному распределению уровня по каналам. Если параметр *баланс* принимает значение −100, звучит только левый канал. Если параметр *баланс* принимает значение 100, звучит только правый канал. Если значение данного параметра меньше −100, он устанавливается равным значению −200 − *баланс*. Если значение данного параметра оказывается больше 100, он устанавливается равным 200 − *баланс*.

Выполнение следующего кода приведет к тому, что при воспроизведении всех звуков в фильме будет задействован только левый канал:

```
var globalSound:Sound = new Sound();
s.setPan(-100);
```

Для определения текущего состояния баланса можно воспользоваться методом `getPan()`, возвращающим числовое значение в диапазоне от −100 до 100. Поскольку баланс одинаковым образом устанавливается для всех звуков, представляемых объектом `Sound`, метод `getPan()` не принимает параметров и возвращает значение баланса для вызвавшего его объекта.

В листинге 19.5 приведен сценарий, реализующий стандартный элемент управления балансом каналов с помощью горизонтального ползунка (рис. 19.1). При полном отключении одного из каналов загорается индикация, представленная двумя клипами с именами `left` и `right` для левого и правого каналов соответственно.

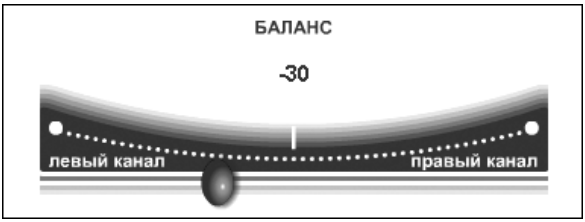


Рис. 19.1. Элемент управления балансом каналов

Листинг 19.5. Управление балансом каналов

```
var s:Sound = new Sound(this);
s.loadSound("Ebony_Samba.mp3", true);
s.start();
// Создаем объекты Color, отвечающие за цвет клипов-индикаторов
var leftCol:Color = new Color (left);
var rightCol:Color = new Color (right);
// Управление ползунком
scroller.onPress = function() {
       this.startDrag(false, slideBar._x, slideBar._y,
       slideBar._x+slideBar._width, slideBar._y);
       // Выключаем индикацию
       leftCol.setRGB(0xFFFFFF);
       rightCol.setRGB(0xFFFFFF);
       this.onMouseMove = function(){
       /* Устанавливаем значение баланса исходя из положения ползунка.
       Точка регистрации полосы slideBar расположена на ее левой границе.
       Точка регистрации ползунка scroller - в его центре */
       s.setPan( -100 + Math.round(100/slideBar._width*2*(scroller._x -
       slideBar._x)));
       // Выводим значение баланса
       panTxt.text = s.getPan();
       };
};
// При отпускании мыши прекращаем перетаскивание ползунка
scroller.onMouseUp = function() {
       this.stopDrag();
       // Если один из каналов полностью отключен, включаем индикацию
       var pan = s.getPan();
       if(pan <=-100) leftCol.setRGB(0xFF0000);
```

```
        if(pan >=100) rightCol.setRGB(0xFF0000);
        delete this.onMouseMove;
};
```

Управление выводом каналов

Рассмотренный выше метод `setPan()` позволяет управлять распределением уровня громкости по каналам, однако не дает возможности контролировать вывод сигнала для каждого канала в отдельности. Стереосигнал состоит из двух независимых друг от друга звуковых составляющих: сигнала левого канала и сигнала правого канала, которые в режиме стереофонического воспроизведения выводятся на разные динамики. В классе `Sound` имеется метод с названием `setTransform()`, позволяющий управлять выводом сигналов для левого и правого каналов на левый и правый динамики. Так, например, с его помощью можно сигнал левого канала вывести одновременно и на левый, и на правый динамик, а сигнал правого канала вообще отключить. Также можно заставить разные звуки воспроизводиться в разных каналах или, наоборот, смешать левый и правый сигнал и выводить их в обоих динамиках, имитируя монофоническое звучание.

Метод `setTransform()` представляет собой своеобразный аналог метода `Color.setTransform()`, используемого для цветовой трансформации объекта `Color`, и имеет следующий синтаксис:

звуковой объект`.setTransform(`*объект трансформации*`)`

Здесь *звуковой объект* — это объект `Sound`, к содержимому которого будет применено действие метода `setTransform()`. Действие данного метода распространяется на все звуковое содержимое, контролируемое объектом *звуковой объект*, так же, как и действие методов `setVolume()` и `setPan()`. Эффект применения метода `setTransform()` определяется значениями свойств объекта *объект трансформации*, переданного ему в качестве параметра.

Параметр *объект трансформации* — это объект `Object`, обладающий следующим набором свойств:

- `ll` — значение в процентах (от 0 до 100), определяющее, какая часть сигнала левого канала будет выведена на левый динамик;
- `lr` — значение в процентах (от 0 до 100), определяющее, какая часть сигнала правого канала будет выведена на левый динамик;
- `rr` — значение в процентах (от 0 до 100), определяющее, какая часть сигнала правого канала будет выведена на правый динамик;
- `rl` — значение в процентах (от 0 до 100), определяющее, какая часть сигнала левого канала будет выведена на правый динамик.

Примечание

Задание свойствам объекта трансформации значений, превышающих 100%, приведет к некоторому увеличению громкости, появлению шумов и искажению звука.

Параметр *объект трансформации* может быть задан с помощью литерала объекта, идентификатором или выражением, возвращающим ссылку на соответствующий объект трансформации.

По умолчанию сигнал левого канала выводится на левый динамик, а сигнал правого — на правый, в связи с этим объект трансформации по умолчанию имеет следующий вид:

```
{ll:100, lr:0, rr:100, rl:0}
```

Приведем несколько примеров, иллюстрирующих результат применения метода `setTransform()`.

- Для того чтобы поменять местами каналы (левый будет звучать в правом динамике, а правый — в левом), нужно воспользоваться следующим объектом трансформации:

  ```
  {ll:0, lr:100, rr:0, rl:100}
  ```

- Для того чтобы отключить правый канал и выводить в оба динамика только левый, объект трансформации должен иметь следующий вид:

  ```
  {ll:100, lr:0, rr:0, rl:100}
  ```

- Для того чтобы в два раза уменьшить громкость звука (не перемещая каналы), используется объект следующего вида:

  ```
  {ll:50, lr:0, rr:50, rl:0}
  ```

- Для того чтобы сместить баланс на 50% в сторону правого канала, объект трансформации должен выглядеть так:

  ```
  {ll:50, lr:0, rr:100, rl:0}
  ```

- Для того чтобы получить монофоническое звучание, нужно смешать каналы с помощью объекта следующего вида:

  ```
  {ll:100, lr:100, rr:100, rl:100}
  ```

Примечание

Как видно из приведенных примеров, метод `setTransform()` может успешно выполнять функции методов `setVolume()` и `setPan()`. Следует иметь в виду, что применение метода `setPan()` отражается на значениях свойств объекта трансформации, в то время как применение метода `setVolume()` не изменяет значений этих свойств.

В листинге 19.6 приведен код, реализующий функциональность, которая позволяет вживую поэкспериментировать с возможностями метода `setTransform()`. Для этих целей применяются четыре текстовых поля с именами `llTxt`, `lrTxt`, `rrTxt` и `rlTxt`, предназначенные для ввода значений свойств объекта трансформации (рис. 19.2.).

При нажатии на кнопку `applyBut` указанные параметры вступают в силу.

Листинг 19.6. Управление выводом звуковых каналов на динамики

```
var s:Sound = new Sound(this);
s.loadSound("Ebony_Samba.mp3", true);
```

```
var sTransformObj:Object = new Object();
applyBut.onPress = function() {
        sTransformObj.ll = Number(llTxt.text);
        sTransformObj.lr = Number(lrTxt.text);
        sTransformObj.rr = Number(rrTxt.text);
        sTransformObj.rl = Number(rlTxt.text);
        s.setTransform(sTransformObj);
};
```



Рис. 19.2. Интерфейс для управления выводом каналов

Для получения значений свойств объекта трансформации, установленных для конкретного объекта `Sound`, можно воспользоваться методом `getTransform()`. Данный метод не принимает параметров и возвращает объект трансформации, содержащий четыре свойства `ll`, `lr`, `rr`, и `rl`.

Для того чтобы вывести в панель **Output** перечень свойств объекта трансформации и их значений, можно воспользоваться следующим кодом:

```
for (var i in s.getTransform()){
        trace(i + ": " + s.getTransform()[i]);
}
```

Пример программного управления звуком. Интерактивная перкуссия

В качестве примера, демонстрирующего возможности программного управления звуком, рассмотрим создание проекта, имитирующего секцию перкуссионных инструментов. Наша секция будет состоять из пяти инструментов: cowbell, ведущий барабан, бонги (в количестве двух) и тарелка. Внешний вид конечного проекта представлен на рис. 19.3. Управление инструментами осуществляется с помощью клавиатуры (цифры основной клавиатуры с 1 по 5 включительно). Громкостью звучания каждого инструмента можно управлять независимо от остальных. Громкость

всех инструментов устанавливается с помощью единственного регулятора. Чтобы определить, на громкость какого инструмента влияет изменение положения регулятора громкости, в нижней части экрана расположены кнопки, соответствующие инструментам. Щелчок на кнопке делает представляемый ею инструмент текущим и устанавливает для него громкость в соответствии с положением регулятора. Изменение положения регулятора громкости сразу же отражается на громкости текущего инструмента. Нажатие на кнопку также вызывает звучание соответствующего ей инструмента. Звучание инструмента визуализируется с помощью анимации, которая заключается в появлении графического изображения, на этом инструменте. Кнопка с надписью "MP3" позволяет загрузить внешний звуковой файл, мелодии которого можно аккомпанировать. После загрузки файла данная кнопка исполняет роль кнопки запуска и остановки звука. Во время воспроизведения внешнего звукового файла выполняется анимация динамиков, которая останавливается при остановке звучания.



Рис. 19.3. Интерактивная перкуссия

Для реализации описанной функциональности создается шесть объектов `Sound` (пять — для инструментов и один — для загружаемого из внешнего файла звука). Этим объектам мы присваиваем имена `s0`, `s1`, ..., `s5`. Создание отдельных объектов для каждого звука необходимо, поскольку мы хотим управлять этими звуками индивидуально. Звуки инструментов присоединяются соответствующими объектами из библиотеки документа.

Динамики представлены двумя экземплярами одного и того же анимированного клипа и имеют имена `mcDyn1` и `mcDyn2`. При запуске воспроизведения загруженного звука их анимация запускается с произвольного кадра с тем, чтобы избежать синхронного движения, которое выглядит менее естественно. Вначале анимация динамиков остановлена.

Для управления инструментами с клавиатуры используются возможности класса `Key`. При нажатии клавиши проверяется ее виртуальный код и, если он совпадает с

кодом одной из заданных клавиш, выполняется запуск воспроизведения звука. При этом клип slDrum, визуализирующий звучание инструмента, позиционируется на нем. Это позволяет оживить инструменты, вызвав визуально выраженную реакцию на нажатие клавиши.

Кнопки, определяющие, какой инструмент является текущим (то есть для него можно регулировать громкость), имеют имена but0, but1, ..., but4 в соответствии с именами соответствующих объектов Sound. Номер текущего инструмента (совпадающий с номером кнопки) сохраняется в переменной curChoice. Эта переменная используется при указании целевого объекта Sound, для которого устанавливается громкость. Для отображения текущего инструмента нажатая кнопка подсвечивается. Подсветка реализуется с помощью клипа highlight, координаты которого устанавливаются равными координате подсвечиваемой кнопки.

Регулировка громкости текущего звука (инструмента или загруженного аудиофайла) выполняется с помощью ползункового регулятора аналогично тому, как это делалось в листинге 18.7. Установленное регулятором значение громкости применяется к текущему инструменту или загруженному звуку.

Кнопка с надписью "MP3" имеет имя loadSoundBut и при первом нажатии осуществляет загрузку внешнего звукового файла, а при последующих нажатиях выполняет попеременную остановку и запуск его воспроизведения. При повторном нажатии воспроизведение продолжается с той позиции, в которой звук находился в момент остановки (аналогично тому, как это было сделано в листинге 19.2).

Сценарий, реализующий описанную функциональность, представлен в листинге 19.7.

Листинг 19.7. Интерактивная перкуссия

```
//ИНИЦИАЛИЗАЦИЯ
/* Создаем объекты Sound для каждого инструмента (s0-s4) и для
последующей загрузки внешнего звука (s5) */
for (var i = 0; i < 6; i++) {
        this["s" + i] = new Sound(this["clip" + i]);
        if(i < 5){// Присоединяем звуки для инструментов
        this["s" + i].attachSound("sound" + i);
        }
}
/* Изначально отключаем анимацию динамиков, поскольку звук еще не
загружен */
mcDyn1.stop();
mcDyn2.stop();
// Создаем объект-приемник событий объекта Key
var obj:Object = new Object();
Key.addListener(obj);
// Отключаем видимость клипа, визуализирующего звучание инструмента
slDrum._visible = false;
```

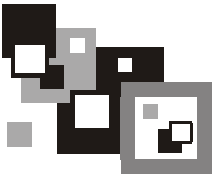
```
// УПРАВЛЕНИЕ ИНСТРУМЕНТАМИ С КЛАВИАТУРЫ
obj.onKeyDown = function() {
        //Включаем видимость клипа, визуализирующего звучание инструмента
        slDrum._visible = true;
        if (Key.isDown(49)) {// При нажатии клавиши <1>...
                s0.start(0.01); //...запускаем звук
                // Позиционируем клип на звучащем инструменте
                slDrum._x = clip0._x;
                slDrum._y = clip0._y;
        }
        if (Key.isDown(50)) {
                s1.start(0.01);
                slDrum._x = clip1._x;
                slDrum._y = clip1._y;
        }
        if (Key.isDown(51)) {
                s2.start(0.01);
                slDrum._x = clip2._x;
                slDrum._y = clip2._y;
        }
        if (Key.isDown(52)) {
                s3.start(0.01);
                slDrum._x = clip3._x;
                slDrum._y = clip3._y;
        }
        if (Key.isDown(53)) {
                s4.start(0.01);
                slDrum._x = clip4._x;
                slDrum._y = clip4._y;
        }
};
// При отпускании кнопки, если ни одна из указанных клавиш не нажата,
// отключаем отображение клипа, визуализирующего звучание инструмента
obj.onKeyUp = function(){
        if(!Key.isDown(49)&&!Key.isDown(50)&&!Key.isDown(51)&&↵
        !Key.isDown(52)&&!Key.isDown(53)){
        slDrum._visible = false;
        }
};
```

```
/* ВЫБОР ТЕКУЩЕГО ИНСТРУМЕНТА С ПОМОЩЬЮ КНОПОК И УСТАНОВКА ДЛЯ НЕГО
ТЕКУЩЕГО ЗНАЧЕНИЯ ГРОМКОСТИ */
// Переменная, хранящая номер текущего инструмента, для которого
// устанавливается громкость
var curChoice:Number = 0;
for (var i = 0; i < 5; i++){
// При нажатии каждой из кнопок...
this["but" + i].onPress = function(){
       // ...подсветка нажатой кнопки
       highlight._x = this._x;
       highlight._y = this._y;
       curChoice = Number(this._name.substr(-1));
       /* ...устанавливаем громкость соответствующего инструмента на
       основании текущего положения ползунка регулятора громкости */
       this._parent["s"+this._parent.curChoice].setVolume(
       Math.round(100/slideBar._width*(scroller._x - slideBar._x)));
       /* ... запускаем воспроизведение звука соответствующего
       инструмента */
       this._parent["s"+this._parent.curChoice].start(0.01);
       slDrum._visible = true;
       /*... позиционируем клип, визуализирующий звучание на звучащем
       инструменте */
       slDrum._x = this._parent["clip"+this._parent.curChoice]._x;
       slDrum._y = this._parent["clip"+this._parent.curChoice]._y;
       };
this["but" + i].onRelease = function(){
       slDrum._visible = false;
       };
}
// РЕГУЛИРОВКА ГРОМКОСТИ С ПОМОЩЬЮ ПОЛЗУНКА
scroller.onPress = function() {
       this.startDrag(false, slideBar._x, slideBar._y,↳
       slideBar._x+slideBar._width, slideBar._y);
       this.onMouseMove = function(){
              /* Устанавливаем значение громкости звука, соответствующего
              подсвеченной кнопке, меняя положение ползунка регулятора
              scroller */
              this._parent["s"+this._parent.curChoice].setVolume(↳
              Math.round(100/slideBar._width*(scroller._x -↳
              slideBar._x)));
       };
};
```

```
// При отпускании мыши прекращаем перетаскивание ползунка регулятора
scroller.onMouseUp = function() {
	this.stopDrag();
	delete this.onMouseMove;
};
// ЗАГРУЗКА ЗВУКА И ЕГО ВИЗУАЛИЗАЦИЯ
// При нажатии кнопки loadSoundBut запускаем загруженный звук, если он не
// звучит, или останавливаем его звучание при повторном нажатии.
loadSoundBut.onPress = function(){
	// Подсвечиваем кнопку loadSoundBut
	highlight._x = this._x;
	highlight._y = this._y;
	curChoice = 5; // Текущий выбор
	// Устанавливаем громкость загруженного звука в соответствии
	// с положением регулятора громкости
	s5.setVolume( Math.round(100/slideBar._width*(scroller._x -↳
	slideBar._x)));
	if(!playing){ //Если звук не звучит...
		s5.start(curPosition/1000); // ...запускаем его
		playing = true; // переключаем флаг
		// Если звук уже загружен...
		if(loadStatus){
		/* Визуализируем его, включая анимацию динамиков. Каждый
		динамик запускается со случайного кадра, так чтобы они
		двигались асинхронно */
		mcDyn1.gotoAndPlay(Math.ceil(Math.random()*↳
		mcDyn1._totalframes));
		mcDyn2.gotoAndPlay(Math.ceil(Math.random()*↳
		mcDyn2._totalframes));
		}
	}
	else { // Если звук звучит...
		s5.stop(); // Останавливаем его
		curPosition = s5.position;
		playing = false; // Переключаем флаг
		mcDyn1.gotoAndStop(1);
		mcDyn2.gotoAndStop(1);
	}
	// Если звук еще не загружен, загружаем его в событийном режиме
	if(!loadStatus) s5.loadSound("Ebony_Samba.mp3", false);
};
```

```
// По окончании загрузки звука...
s5.onLoad = function(success){
if(success){
       this.start(); // Запускаем его
       // Визуализируем его, включая анимацию динамиков
       mcDyn1.gotoAndPlay(Math.ceil(Math.random()*mcDyn1._totalframes));
       mcDyn2.gotoAndPlay(Math.ceil(Math.random()*mcDyn2._totalframes));
       loadStatus = true; // Переключаем флаг
}
else songTxt.text = "Ошибка при загрузке звука !";
};
/* После загрузки ID3-тегов выводим информацию о названии композиции и
исполнителе... */
s5.onID3 = function(){
       songTxt.text = this.id3.songname;
       artistTxt.text = this.id3.artist;
};
```

Глава 20



Управление процессом загрузки внешних документов и изображений. Взаимодействие между SWF-документами

Сложные системы имеют тенденцию противопоставлять
себя своим же функциям.

Принцип Шательера

В настоящей главе мы рассмотрим возможности контроля загрузки внешних документов, предоставляемые новым классом `MovieClipLoader`[1]. Также здесь будут описаны возможности организации взаимодействия между разными SWF-документами, реализуемые с помощью класса `LocalConnection`. В заключение главы мы рассмотрим связанные с этими темами вопросы политики безопасности, которые имеют место для проигрывателя Flash Player v7.0.

Класс *MovieClipLoader*

В *главе 12* мы уже познакомились с базовыми возможностями ActionScript, позволяющими осуществлять загрузку и выгрузку внешних документов. К числу таких программных инструментов относятся методы `loadMovie()` и `unloadMovie()`, позволяющие загружать внешние документы на монтажные линейки клипов и удалять их оттуда, а также глобальная функция `loadMovieNum()`, выполняющая загрузку на заданный уровень стека документов. Эти средства вполне эффективны и часто применяются на практике, однако их использование в ряде случаев приводит к существенным трудностям, связанным с отсутствием адекватных возможностей контроля хода загрузки. Перечислим наиболее существенные недостатки "традиционных" возможностей:

- отсутствие простых способов создания универсального предзагрузчика, позволяющего контролировать одновременную загрузку нескольких документов;

[1]Класс `MovieClipLoader` появился в арсенале встроенных программных средств ActionScript с выходом седьмой версии Flash — Macromedia Flash MX 2004. — *Прим. автора.*

- отсутствие контроля этапов загрузки. Используя традиционные методы, нельзя определить момент, соответствующий началу загрузки, и момент инициализации загруженного документа;
- отсутствие простых способов получения информации о причине невозможности или прекращения загрузки.

Класс `MovieClipLoader` позволяет устранить все перечисленные трудности и обеспечивает возможность детального контроля процесса загрузки внешних файлов в формате SWF и JPEG. В этом разделе будут описаны все возможности данного класса и приведен практический пример их применения.

Создание экземпляра класса

Механизм использования класса `MovieClipLoader` основан на модели трансляции событий. Для ее реализации создается объект `MovieClipLoader`, который может транслировать различные события, связанные с ходом загрузки, объектам, зарегистрированным в качестве его приемников. С помощью объекта `MovieClipLoader` выполняется вызов методов класса `MovieClipLoader`, позволяющих инициировать или прекратить процесс загрузки, а также получить сведения о ее состоянии.

Создание объекта `MovieClipLoader` выполняется с помощью оператора `new` и конструктора `MovieClipLoader()`, не принимающего параметров:

```
var mcl:MovieClipLoader = new MovieClipLoader();
```

Один и тот же объект `MovieClipLoader` может инициировать загрузку любого количества внешних документов. При этом ход загрузки для каждого из них можно контролировать с помощью объектов-приемников, получающих сведения от осуществляющего загрузку объекта `MovieClipLoader`. Вместе с тем при необходимости можно для загрузки каждого документа использовать индивидуальный объект `MovieClipLoader`.

Удаление объекта `MovieClipLoader` во время загрузки внешнего документа не приведет к ее прекращению, но сделает невозможным вызов методов этого объекта или обработку его событий.

Методы класса *MovieClipLoader*

Встроенные методы класса `MovieClipLoader` позволяют инициировать или прекратить процесс загрузки, получить сведения о ее состоянии, а также зарегистрировать объект на прием сообщений о событиях от объекта `MovieClipLoader` и исключить его из списка объектов-приемников.

Инициирование загрузки документа

Для инициирования загрузки внешнего документа используется метод `loadClip()`. С его помощью можно осуществить загрузку внешнего SWF-документа или файла в формате JPEG. Данный метод совмещает в себе назначение метода `loadMovie()` и глобальной функции `loadMovieNum()`, позволяя осуществлять загрузку в заданный клип или на заданный уровень стека документов. Напомним, что Flash не поддер-

живает загрузку JPEG-файлов, использующих режим **Progressive**, поэтому от их применения следует отказаться.

Метод `loadClip()` имеет следующий синтаксис:

загрузчик`.loadClip(`*URL*`, `*целевой клип*`)`

Здесь *загрузчик* — это объект `MovieClipLoader`.

Параметр *URL* — это строка, содержащая путь к загружаемому файлу. Допускается применение как относительных, так и абсолютных ссылок.

Параметр *целевой клип* — это ссылка на клип или уровень документа, в который будет загружен внешний файл. Если загруженный файл должен быть помещен в клип, данный параметр представляет собой ссылку на этот клип. Если загруженный файл должен быть помещен на уровень стека документа, данный параметр задается целым положительным числом, указывающим номер уровня.

Метод `loadClip()` возвращает булево значение `true` в случае успешной отсылки запроса по заданному URL. В противном случае возвращаемым значением является `false`.

Результат загрузки внешнего файла в клип или на уровень стека с помощью метода `loadClip()` в точности соответствует результатам применения `loadMovie()` или `loadMovieNum()` подробно описанным в *главе 12*. В частности, при загрузке внешнего файла в клип все содержимое его монтажной линейки (включая и размещенные на ней сценарии) удаляется. Методы обработчиков событий, определенные для этого клипа, также удаляются. При загрузке внешнего файла его левый верхний угол оказывается в точке регистрации принимающего клипа. При загрузке внешнего документа на уровень стека документов его левый верхний угол совпадает с левым верхним углом рабочей области.

Прекращение загрузки или выгрузка документа

Для того чтобы прервать загрузку внешнего документа или выгрузить полностью загруженный документ из клипа или с уровня стека, применяется метод `unloadClip()`. Вызов этого метода в процессе загрузки приведет к ее остановке. Если метод вызывается после окончания загрузки документа, выполняется его выгрузка. В результате выгрузки внешнего документа принимающий его клип не удаляется, но он лишается всего своего наполнения (то есть остается пустым). Метод `unloadClip()` имеет следующий синтаксис:

загрузчик`.unLoadClip(`*целевой клип*`)`

Параметр *целевой клип* — это ссылка на клип или номер уровня стека документов, из которого требуется выгрузить загруженный в него внешний документ.

Следующий код позволяет начать загрузку документа на первый уровень стека документов при нажатии на кнопку `loadBut` и прекратить ее (либо выгрузить загруженный документ) при нажатии на кнопку `unloadBut`.

```
var mcl:MovieClipLoader = new MovieClipLoader();
loadBut.onPress = function() {
```

```
        mcl.loadClip("content/clip.swf", 1);
};
unloadBut.onPress = function() {
        mcl.unloadClip(1);
};
```

Получение информации о ходе загрузки

Для получения информации о количестве загруженной информации в классе `MovieClipLoader` имеется метод `getProgress()`. По сути, этот метод является аналогом методов `getBytesLoaded()` и `getBytesTotal()` класса `MovieClip`. Метод `getProgress()` имеет следующий синтаксис:

загрузчик`.getProgress(`*целевой клип*`)`

Параметр *целевой клип* — это ссылка на клип или номер уровня стека документов, в который загружается внешний файл.

Возвращаемым значением метода `getProgress()` является объект со следующими двумя свойствами:

- `bytesLoaded` — содержит информацию о количестве загруженных в *целевой клип* байтов от объема внешнего файла;
- `bytesTotal` — содержит информацию об объеме загружаемого в *целевой клип* документа в байтах.

Данный метод не является единственной возможностью получения информации о ходе загрузке. Для этих целей также может быть использован обработчик события `onLoadProgress`, описание которого будет приведено в следующем подразделе. Характерной особенностью метода `getProgress()` является то, что он может быть вызван по требованию в любой момент в процессе загрузки, в то время как событие `onLoadProgress` является системным и генерируется по мере поступления новых данных с сервера. Кроме того, метод `getProgress()` можно применять при тестировании локально расположенного файла в режиме имитации загрузки (**Simulate Download**), а обработчик `onLoadProgress()` функционирует только при реальной загрузке удаленного файла.

Пример, демонстрирующий применение данного метода, представлен в листинге 20.1.

Обработка событий

Основным преимуществом класса `MovieClipLoader` является наличие в нем набора обработчиков событий, позволяющих контролировать этапы и состояние процесса загрузки.

Генератором событий, отражающих ход загрузки, является объект `MovieClipLoader`. Обработка событий объекта `MovieClipLoader` выполняется с помощью объекта-приемника (или нескольких объектов), зарегистрированного на получение сообщений о соответствующих событиях. Регистрация объектов-приемников и их отключение

реализуется с помощью стандартных методов addListener() и removeListener(), определенных в классе MovieClipLoader. Таким образом, для получения возможности обработки методов объекта MovieClipLoader необходимо выполнить три типичные для модели "источник-приемник" процедуры:

1. Создать экземпляр класса MovieClipLoader — объект-источник.
2. Создать объект-приемник (объект типа Object) и зарегистрировать его на получение сообщений о событиях от объекта MovieClipLoader.
3. Определить в объекте-приемнике методы обработчиков событий класса MovieClipLoader.

Примечание

Необходимо иметь в виду, что обработчики событий класса MovieClipLoader могут быть использованы только по отношению к документам, загружаемым с помощью метода loadClip(). При загрузке с помощью loadMovie() и loadMovieNum() эти обработчики не действуют.

В этом разделе будут рассмотрены все события класса MovieClipLoader в порядке, соответствующем реальному ходу загрузки.

Начало загрузки. Событие *onLoadStart*

Событие onLoadStart генерируется в случае, если метод loadClip() начал загружать внешний документ. При обнаружении данного события функции, которая присваивается свойству onLoadStart объекта-приемника, в качестве параметра передается ссылка на объект (клип или уровень), в который загружается внешний документ. Определение метода onLoadStart() выполняется с применением следующего синтаксиса:

```
объект-приемник.onLoadStart = function(целевой объект){
блок кода;
...
};
```

Параметр *целевой объект* не является обязательным, то есть при определении метода обработчика формальный параметр можно не указывать. Тем не менее он все равно передается функции при трансляции события. Применение данного параметра может оказаться чрезвычайно полезным при загрузке нескольких разных файлов, ход которой контролируется с помощью одного и того же объекта-приемника. В подобной ситуации успешное начало загрузки любого из внешних документов будет сопровождаться наступлением события onLoadStart, которое повлечет за собой выполнение кода обработчика. Между тем зачастую необходимо выполнить те или иные действия в зависимости от того, в какой именно контейнер (клип или уровень) загружается предназначенный ему файл. Например, при параллельной загрузке четырех клипов для каждого можно вывести свой предзагрузчик. Именно для этих целей в основном и используется параметр *целевой объект*, указывающий инструкциям обработчика события, с каким объектом связано обнаруженное событие.

Данный параметр является универсальным и передается методам обработчиков всех событий класса MovieClipLoader.

Событие onLoadStart может быть использовано для выполнения действий, связанных с началом загрузки. Чаще всего его используют для вывода изображения предзагрузчика и выполнения визуализации процесса загрузки, например, запуска анимации, продолжающейся до окончания (или прерывания) загрузки. Как известно, при загрузке внешнего документа в клип исходное содержимое клипа-контейнера удаляется. Это обстоятельство необходимо иметь в виду, поскольку при обнаружении события onLoadStart клип-контейер уже оказывается лишенным всего своего содержимого. В частности, это означает, что при обращении к свойствам _width и _height объекта *целевой объект* из тела метода onLoadStart() будет получено значение 0. Выполнить какое-либо действие с загружаемым в клип-контейнер (или на уровень) содержимым можно будет только после обнаружения события onLoadInit, описанного далее.

Следует отметить, что после обнаружения события onLoadStart для клипа, в который осуществляется загрузка, можно определить любой метод обработчика события клипа. Это довольно важное обстоятельство, поскольку, как известно, в результате загрузки в клип внешних документов его методы обработчиков перестают функционировать. Следовательно, если в процессе загрузки (или после ее окончания) клип-контейнер должен выполнять обработку событий, соответствующие методы обработчиков можно определить непосредственно при обнаружении события onLoadStart.

Такой подход применяется в листинге 20.1, демонстрирующем создание простого внешнего предзагрузчика. Принцип его работы заключается в том, что при обнаружении события onLoadStart (то есть в начале загрузки) для клипа-контейнера определяется метод обработчика события onEnterFrame, который периодически проверяет состояние загрузки и визуализирует эту информацию. Информация о состоянии загрузки определяется с помощью описанного ранее метода getProgress(). Контроль состояния загрузки можно было бы осуществить и более простым способом, используя метод обработчика onLoadProgress(), однако этот подход не позволяет выполнять тестирование загрузки локально расположенного файла.

В приведенном далее листинге загрузка осуществляется в клип с именем holder. Предзагрузчик представлен клипом с именем preloader, который содержит на своей монтажной линейке клип statusBar, представляющий полосу состояния, и два текстовых поля statusTxt и infoTxt — для вывода процента загрузки и информации о загруженном объеме соответственно. Подобный предзагрузчик можно применить для контроля загрузки файла, содержащего большой объем информации, который должен быть загружен перед первым кадром. При этом важно, чтобы габариты (размер в пикселах) файла-предзагрузчика совпадали с габаритами загружаемого документа.

Примечание

Для локального тестирования предзагрузчиков необходимо использовать команду **View>Simulate Download** среды тестирования.

Листинг 20.1. Предзагрузчик с возможностью локального тестирования

```
// Создаем объект MovieClipLoader
var mcl:MovieClipLoader = new MovieClipLoader();
// Создаем объект-приемник
var mclListener:Object = new Object();
// Регистрируем mclListener на получение сообщений от mcl
mcl.addListener(mclListener);
// Начальный масштаб полосы состояния загрузки
preloader.statusBar._xscale = 1;
// Инициируем загрузку документа в клип holder
mcl.loadClip("content/clip1.swf", holder);
// В начале загрузки…
mclListener.onLoadStart = function(target) {
       target.onEnterFrame = function() {
              // Локальный объект с информацией о состоянии загрузки
              var progress:Object = mcl.getProgress(this);
              var loadStatus:Number = Math.round(progress.bytesLoaded /
              progress.bytesTotal * 100);
              // Если информация о загруженном объеме поступила...
              if (progress.bytesLoaded) {
              // Изменяем ширину полосы состояния
              this._parent.preloader.statusBar._xscale = loadStatus;
              // Выводим процент загрузки
              this._parent.preloader.statusTxt.text = loadStatus + "%";
              // Выводим данные о загрузке
              this._parent.preloader.infoTxt.text = "Загружено " +
              Math.round(progress.bytesLoaded / 1024) + "Кб из " +
              Math.round(progress.bytesTotal / 1024);
              }
              // Если загрузка окончена...
              if (loadStatus == 100) {
                     /* Выгружаем содержимое предзагрузчика и заканчиваем
                     его работу */
                     mcl.unloadClip(preloader);
                     delete this.onEnterFrame;
              }
       };
};
```

Поступление данных. Событие *onLoadProgress*

Для контроля хода загрузки в классе `MovieClipLoader` имеется обработчик события `onLoadProgress`. Данное событие периодически генерируется в ходе процесса загрузки при получении и сохранении на диске очередной порции загружаемой информации. Следует иметь в виду, что обработчик этого события функционирует только при реальной загрузке удаленного файла. Это означает, что его работу невозможно оценить, загружая локально расположенный документ. С помощью данного события можно не только получать известие о загрузке новых данных, но также определять состояние загрузки.

Определение метода `onLoadProgress()` имеет следующий синтаксис:

```
объект-приемник.onLoadProgress = function(целевой объект, загружено,↵
общий объем){
блок кода;
...
};
```

Параметр *целевой объект* имеет то же назначение, что и у метода `onLoadStart()`.

Параметр *загружено* — это число, соответствующее объему информации в байтах, загруженному в *целевой объект* на момент обнаружения данного события.

Параметр *общий объем* — это число, соответствующее суммарному объему информации в байтах, загружаемому в *целевой объект*.

Параметры не являются обязательными, однако даже при отсутствии формальных параметров фактические параметры передаются методу обработчика и могут быть использованы его инструкциями с помощью объекта `arguments`.

Далее приводится простой образец применения данного события для вывода в панель **Output** информации о ходе загрузки.

```
mclListener.onLoadProgress = function(target, loaded, total) {
        // Вычисляем состояние загрузки в процентах
        var loadStatus = Math.round(loaded / total * 100);
        trace("В клип " + target + " загружено " + loadStatus + "%");
};
```

Окончание загрузки. Событие *onLoadComplete*

Событие `onLoadComplete` генерируется после того, как внешний файл будет полностью загружен и записан на диск. Данное событие можно использовать при необходимости выполнения определенных действий, связанных с окончанием загрузки, например, для того, чтобы прекратить анимацию, выполняющуюся на протяжении загрузки, и скрыть предзагрузчик. Следует иметь в виду, что данное событие гене-

рируется до инициализации загруженного документа, то есть до того, как его переменные, свойства и методы станут доступны. Это означает, что из обработчика события `onLoadComplete` нельзя вызвать функцию, объявленную в загруженном документе, запустить его анимацию, задать размер и т. д. Поэтому данное событие обычно используется для выполнения каких-то действий в загружающемся фильме. Управление загружаемым документом осуществляется при обнаружении события `onLoadInit`, описанного далее.

Необходимо отметить, что событие `onLoadComplete` корректно работает только при загрузке удаленного файла. Это связано с тем, что данное событие генерируется после того, как загружаемый документ полностью записан на диск, поэтому имитация загрузки локально расположенных файлов приводит к преждевременному обнаружению этого события. При тестировании загрузки файлов, расположенных локально, событие `onLoadComplete` генерируется сразу же после вызова метода `loadClip()`, то есть еще до окончания имитируемой загрузки внешнего файла.

Определение метода `onLoadProgress()` имеет следующий синтаксис:

```
объект-приемник.onLoadComplete = function(целевой объект){
блок кода;
…
};
```

Параметр *целевой объект* является необязательным и имеет то же назначение, что и у метода `onLoadStart()`.

Инициализация загруженного документа. Событие *onLoadInit*

Событие `onLoadInit` генерируется после инициализации загруженного клипа, то есть после того, как был выполнен весь программный код в его первом кадре (сценарий основной монтажной линейки и сценарии клипов, расположенных в первом кадре). В обработчик данного события можно поместить инструкции, управляющие клипом (или содержимым уровня стека), в который была выполнена загрузка внешнего документа. Так, например, обработчик события может позиционировать клип-контейнер, придать ему требуемый размер, запустить его анимацию.

Определение метода `onLoadInit()` имеет следующий синтаксис:

```
объект-приемник.onLoadInit = function(целевой объект){
блок кода;
…
};
```

Параметр *целевой объект* является необязательным и имеет то же назначение, что и у метода `onLoadStart()`.

Ошибка при загрузке. Событие *onLoadError*

Событие `onLoadError` генерируется в случае невозможности выполнения загрузки или в результате ее прерывания. С помощью данного события можно не только установить факт прекращения загрузки, но и определить его причину. Определение метода `onLoadError()` имеет следующий синтаксис:

```
объект-приемник.onLoadError = function(целевой объект, код ошибки){
блок кода;
...
};
```

Параметр *целевой объект* является необязательным и имеет то же назначение, что и у метода `onLoadStart()`.

Параметр *код ошибки* — это строка, указывающая причину ошибки загрузки, которая передается методу обработчика при трансляции события. Данный параметр может принимать только два фиксированных значения:

- `"URLNotFound"` — данное значение возвращается в случае, если Flash не смог начать загрузку указанного файла. В этой ситуации вызов метода `loadClip()` не приводит к возникновению ни события `onLoadStart`, ни события `onLoadComplete`. Причиной этого может служить неверный URL загружаемого документа или отсутствие ответа сервера;
- `"LoadNeverCompleted"` — данное значение возвращается, если загрузка была успешно начата, однако указанный документ не смог загрузиться. В этом случае вызов метода `loadClip()` привел к обнаружению события `onLoadStart`, однако событие `onLoadComplete` не произошло. Такая ситуация может иметь место в результате сбоя в обмене данными с сервером, при перегрузке сервера или в результате разрыва интернет-соединения.

Отметим, что вызов метода `unloadClip()` во время загрузки внешнего документа приводит к прекращению загрузки, сопровождаемому возникновением события `onLoadError`. В этом случае параметр *целевой объект*, передаваемый методу обработчика, представляет не ссылку на объект, в который выполнялась загрузка, а ссылку на уровень стека документа. Возможно, причиной этого является ошибка в Flash Player. При этом второму параметру метода обработчика передается строка `"LoadNeverCompleted"`. Такая ситуация, однако, имеет место только при загрузке удаленного файла. При локальном тестировании вызов метода `unloadClip()` вообще не приводит к обнаружению события `onLoadError`.

В теле обработчика события `onLoadError` обычно размещают инструкции, которые должны быть выполнены при возникновении ошибки в загрузке, например, код, выводящий сообщение о невозможности выполнения загрузки.

Ниже приведен код, демонстрирующий модель применения обработчиков событий класса `MovieClipLoader`.

```
var mcl:MovieClipLoader = new MovieClipLoader();
var mclListener:Object = new Object();
```

```
mcl.addListener(mclListener);
mcl.loadClip("content/clip.swf", mc);
mclListener.onLoadStart = function(target) {
       trace("Началась загрузка в клип " + target);
};
mclListener.onLoadProgress = function(target, loaded, total) {
       trace("В клип " + target + " Загружено " + Math.round(loaded /↳
       total * 100 / 1024) + "Кб из " + total / 1024);
};
mclListener.onLoadComplete = function(target) {
       trace("Загрузка в клип" + target + " закончена");
};
mclListener.onLoadInit = function(target) {
       trace("Загруженный документ инициализировался");
       // Позиционируем клип-контейнер с загруженным документом
       target._x = 0;
       target._y = 0;
};
mclListener.onLoadError = function(target, errorCode) {
       switch (errorCode) {
       case "URLNotFound" :
              trace("ОШИБКА! URL не найден.");
              break;
       case "LoadNeverCompleted" :
              trace("ОШИБКА! Загрузка прервана.");
              break;
       }
};
```

Пример управления загрузкой нескольких документов

Существенным преимуществом класса `MovieClipLoader` является возможность создания универсального кода предзагрузчика, позволяющего контролировать параллельную или последовательную загрузку нескольких документов. Программный код, реализующий такой предзагрузчик, представлен в листинге 20.2. Приведенный предзагрузчик позволяет выполнять одновременную загрузку четырех документов и обладает следующей функциональностью (рис. 20.1).

- Загрузка документа начинается при нажатии на одну из четырех кнопок. Во время загрузки документа соответствующая ему кнопка становится неактивной.

- Процесс загрузки каждого документа визуализируется с помощью анимации и появляющейся во время загрузки панели предзагрузчика. Предзагрузчик выводит информацию о проценте и объеме загруженной информации. Для каждого документа используется собственный предзагрузчик.
- В случае невозможности выполнения или прекращения загрузки появляется информационная панель с соответствующим сообщением. Щелчок по этой панели закрывает ее, после чего кнопки, позволяющие инициировать загрузку, снова активизируются.

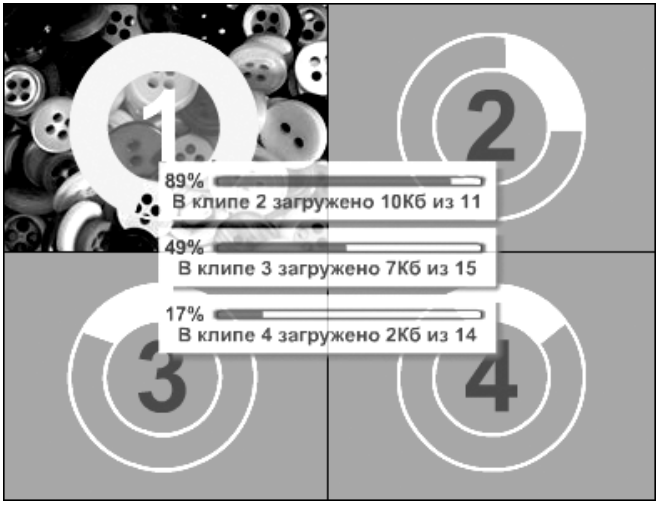


Рис. 20.1. Управление загрузкой нескольких документов

Приведенный программный код снабжен подробными комментариями, поэтому ограничимся только кратким описанием основных элементов, задействованных в программе.

Загрузка внешних документов выполняется в клипы-контейнеры с именами `holder1`, ..., `holder4`. Размер каждого контейнера в точности соответствует размеру загружаемого документа, точки регистрации контейнеров находятся в левом верхнем углу. Таким образом, загруженные документы точно позиционируются в предоставленной им области.

На протяжении процесса загрузки над областью, в которую будет помещен загружаемый документ, появляется анимированный клип с идентификатором связи `lCircle`, который динамически присоединяется из библиотеки документа. При прерывании загрузки анимация присоединенного экземпляра прекращается.

Загрузка документов инициируется с помощью пустых (невидимых) кнопок с именами `but1`, ..., `but4`, расположенных над соответствующими клипами-контейнерами. На протяжении загрузки эти кнопки отключаются с помощью свойства `enabled` во избежание выполнения запуска повторной загрузки.

Графический предзагрузчик представляет собой клип с идентификатором связи `preloader`. Этот клип динамически присоединяется для каждого загружаемого документа во время выполнения загрузки и удаляется при ее прекращении или завершении. Клип содержит два текстовых поля — `statusTxt` и `infoTxt` — для вывода процента загрузки и объема загруженной информации соответственно. На его монтажной линейке также находится клип `statusBar`, представляющий собой стандартную полосу состояния.

Панель, появляющаяся при прекращении (или невозможности) загрузки, представляет собой клип с идентификатором связи `loadFailed`, который так же, как и предзагрузчик, динамически присоединяется в соответствующий момент для каждого загружаемого документа.

В приведенном сценарии обработка всех событий, связанных с загрузкой документов, осуществляется с помощью единственного объекта-приемника `mclListener`. Для определения, с каким именно клипом-контейнером связан текущий вызов метода-обработчика событий, можно воспользоваться передаваемым ему при вызове параметром `target`. Поскольку имена всех элементов, участвующих в реализации предзагрузчика, унифицированы (в конце их имен имеется порядковый номер), это дает возможность создать универсальный код, выполняющий необходимые действия для любого загружаемого в клип-контейнер документа вне зависимости от того, происходит ли их загрузка последовательно или параллельно. Для определения номера клипа, с которым связано событие загрузки в теле метода обработчика, достаточно вырезать последний символ (номер) из имени объекта, передаваемого параметром `target`.

Поскольку событие `onLoadProgress` действует только при загрузке удаленного файла, в листинг 20.2 включен фрагмент кода (указан в комментариях), позволяющий выполнить локальное тестирование представленного предзагрузчика. Для того чтобы это сделать, необходимо раскомментировать соответствующий код. Для выполнения реальной загрузки нужно поместить загружаемые файлы на удаленный сервер, сохранив их взаимное расположение. В этом случае фрагмент кода для локального тестирования не требуется (его необходимо удалить или закомментировать).

Листинг 20.2. Управление загрузкой нескольких документов

```
// *ИНИЦИАЛИЗАЦИЯ*
// Создаем объект MovieClipLoader
var mcl:MovieClipLoader = new MovieClipLoader();
// Создаем объект-приемник и регистрируем его на прием сообщений от mcl
var mclListener:Object = new Object();
mcl.addListener(mclListener);
// Сохраняем ссылку на текущую линейку для обращения к ней из методов
// обработчиков
var _this:Object = this;
// Функция для вывода сообщения о прекращении или прерывании загрузки
// *НАЧАЛО ЗАГРУЗКИ*
```

```
mclListener.onLoadStart = function(target) {
        // Сохраняем номер клипа, в который осуществляется загрузка
        var n = Number(String(target).substr(-1));
        /* Отключаем кнопки, так чтобы во время загрузки исключить
        возможность ее повторного выполнения */
        // Выводим клип с анимацией
        _this.attachMovie("lCircle", "lCircle" + n, n, {_x:target._x +
        98.5, _y:target._y + 75});
        // Выводим панель предзагрузчика
        _this.attachMovie("preloader", "preloader" + n, n + 4, {_x:204,
        _y:80 + 40 * (n - 1)});
        // Установка начальной ширины полосы состояния загрузки
        _this["preloader" + n].statusBar._xscale = 1;
        _this["preloader" + n].infoTxt.text = "Загрузка клипа " + n + "
        началась...";
/*//-------------Код для локального тестирования---------------
target.onEnterFrame = function() {
var progress:Object = mcl.getProgress(this);
var loadStatus:Number = Math.round(progress.bytesLoaded
/progress.bytesTotal * 100);
progress.bytesTotal();
// Если информация о загруженном объеме поступила...
if (progress.bytesLoaded) {
this._parent["preloader" + n].statusBar._xscale = loadStatus;
this._parent["preloader" + n].statusTxt.text = loadStatus + "%";
this._parent["preloader" + n].infoTxt.text = "В клипе " + n +
" загружено "+ Math.round(progress.bytesLoaded / 1024) + "Кб" + " из " +
Math.round(progress.bytesTotal / 1024);
}
if (loadStatus == 100) {
delete this.onEnterFrame;
}
};
//---------------Конец кода для локального тестирования-------- */
};
// *ПРИ ПОСТУПЛЕНИИ НОВЫХ ДАННЫХ В ХОДЕ ЗАГРУЗКИ*
mclListener.onLoadProgress = function(target, loaded, total) {
        var n = String(target).substr(-1);
        // Вычисляем состояние загрузки в процентах
        var loadStatus = Math.round(loaded / total * 100);
```

```
        // Обновляем полосу состояния загрузки
        _this["preloader" + n].statusBar._xscale = loadStatus;
        // Выводим процент загрузки
        _this["preloader" + n].statusTxt.text = loadStatus + "%";
        // Выводим информацию о загрузке
        _this["preloader" + n].infoTxt.text = "В клипе " + n + "загружено
        " + Math.round(loaded / 1024) + "Кб" + " из " + Math.round(total
        / 1024);
};
// *ПО ОКОНЧАНИИ ЗАГРУЗКИ*
mclListener.onLoadComplete = function(target) {
        var n = String(target).substr(-1);
        _this["preloader" + n].infoTxt.text = "Клип " + n + " загружен";
};
// *ПОСЛЕ ВЫПОЛНЕНИЯ СЦЕНАРИЯ ПЕРВОГО КАДРА ЗАГРУЖЕННОГО КЛИПА*
mclListener.onLoadInit = function(target) {
        var n = String(target).substr(-1);
        // Удаляем клип с анимацией
        _this["lCircle" + n].removeMovieClip();
        // Удаляем панель предзагрузчика
        _this["preloader" + n].removeMovieClip();
};
// *ОБРАБОТКА ОШИБКИ ПРИ ЗАГРУЗКЕ*
mclListener.onLoadError = function(target, errorCode) {
        var n = Number(String(target).substr(-1));
        // Выводим панель сообщения о прекращении загрузки
        _this.attachMovie("loadFailed", "loadFailed" + n, n + 4, {_x:204,
        _y:80 + 40 * (n - 1)});
        // Останавливаем клип с анимацией
        _this["lCircle" + n].stop();
        // В зависимости от причины прекращения загрузки выводим
        // сообщение
        switch (errorCode) {
        case "URLNotFound" :
        _this["loadFailed" + n].infoTxt.text = "ОШИБКА! URL не найден.";
                break;
        case "LoadNeverCompleted" :
        _this["loadFailed" + n].infoTxt.text = errorCode + " " +
        arguments[0] + "ОШИБКА! Загрузка прервана.";
```

```
                break;
        }
        // Скрываем панель сообщения при нажатии на нее
        _this["loadFailed" + n].onPress = function() {
                this.removeMovieClip();
        // Вновь активизируем кнопки для возможности повторной загрузки
                _this["but" + n].enabled = true;
        // Удаляем клип с анимацией, присутствующий во время загрузки
                _this["lCircle" + n].removeMovieClip();
        };
};
// *ИНИЦИИРОВАНИЕ ЗАГРУЗКИ ВНЕШНЕГО ДОКУМЕНТА ПРИ НАЖАТИИ НА КНОПКУ*
for (var i = 1; i < 5; i++) {
// При нажатии на кнопку...
this["but" + i].onPress = function() {
        var n = this._name.substr(-1);
        //Инициируем выполнение загрузки соответствующего документа
        mcl.loadClip("content/clip" + n+ ".swf", _this["holder" + n]);
                this.enabled = false;
        };
}
```

Вопросы политики безопасности, связанные с загрузкой внешних SWF-документов

Flash-фильм может осуществлять загрузку внешних SWF-документов вне зависимости от их фактического местоположения в сети. С загруженными документами можно выполнять все стандартные операции, не требующие доступа к их содержимому, например, масштабирование, поворот, изменение прозрачности, цветовых характеристик и т. д. Тем не менее политика безопасности, поддерживаемая проигрывателем Flash Player 7, существенно ограничивает возможности взаимодействия с загруженными документами. Это ограничение выражается в следующих характерных особенностях.

- По умолчанию один документ может получить доступ к объектам и переменным другого документа, только если оба документа загружены с одного и того же домена. Это означает, что файл, с **www.wherever.com**, загруженный в документ, расположенный на **www.anywhere.com**, не сможет получить доступ к его переменным и функциям, равно как и принимающий файл не сможет обратиться к переменным загруженного документа.

- По умолчанию один документ может получить доступ к объектам и переменным другого документа, только если оба документа загружены с применением одного и того же протокола (HTTP или HTTPS). Это означает, что файл с **http://www.wherever.com** не сможет получить доступ к переменным и функциям документа с **https://www.wherever.com**, равно как и принимающий файл не сможет обратиться к переменным загруженного документа (даже несмотря на идентичность их доменов).

Следует отметить, что подобные правила применяются при использовании любых средств загрузки внешних документов (`MovieClip.loadMovie()`, `loadMovieNum()`, `MovieClipLoader.loadClip()`).

Для того чтобы разрешить доступ к объектам и переменным документа, полученного с другого домена, можно воспользоваться методом `allowDomain()` объекта `security`, принадлежащего классу `System`. Данный метод имеет следующий синтаксис:

```
System.security.allowDomain(домен1, домен2, ... доменN)
```

Параметры *домен1*, *домен2*, ... *доменN* — это строки, указывающие имена доменов, файлам с которых разрешается получать доступ к объектам и переменным документа, использующего данный метод.

Имя домена, передаваемое в качестве параметра, может включать указание протокола и также может быть задано с применением IP-адреса. Ниже приводятся допустимые варианты задания имен:

```
"wherever.com"
"http://wherever.com"
"http://65.57.83.12"
```

При использовании проигрывателя Flash Player 7 параметры *домен1*, *домен2*, ... *доменN* должны содержать точные имена доменов (домены **www.wherever.com** и **wherever.com** будут считаться разными).

Необходимо иметь в виду, что если документ *A* разрешает доступ к своим объектам и переменным документу *B*, то это не значит, что документ *A* автоматически получает доступ к переменным документа *B*. Для открытия такого двустороннего доступа необходимо, чтобы оба документа содержали вызов метода `allowDomain()`, разрешающего их взаимодействие.

Следующий код позволяет получить доступ к объектам и переменным текущего документа всем файлам из доменов **www.wherever.com** и **www.anywhere.com**:

```
System.security.allowDomain("www.wherever.com", "www.anywhere.com");
```

Для того чтобы разрешить доступ к объектам и переменным документам с любого домена, нужно передать методу `allowDomain()` пустую строку:

```
System.security.allowDomain("");
```

Для того чтобы разрешить доступ к файлам, использующим защищенный протокол HTTPS, документам, использующим незащищенный протокол HTTP, необходимо использовать метод `allowInsecureDomain()`. Данный метод имеет следующий синтаксис:

```
System.security.allowInsecureDomain(домен1, домен2, ... доменN))
```

Параметры *домен1*, *домен2*, ... *доменN* — это строки, указывающие точные имена доменов файлам, с которых разрешается получать доступ к объектам и переменным документа, использующего данный метод.

Отметим, что для обеспечения доступа к содержимому документа, использующему защищенный протокол, со стороны документа, расположенного в том же домене, но с применением незащищенного протокола, необходимо вызвать данный метод, передав ему в качестве параметра точное имя домена.

Класс *LocalConnection*. Взаимодействие между SWF-документами

Наряду с возможностью загрузки в проигрыватель внешних SWF-документов ActionScript позволяет организовать взаимодействие между Flash-фильмами, которые одновременно воспроизводятся в разных окнах проигрывателей на одном компьютере. Эта возможность может оказаться чрезвычайно полезной, например, в случае, когда в разных частях Web-страницы требуется разместить два разных Flash-ролика, в одном из которых отображается динамическое содержимое (например, фотогалерея), а в другом находится панель управления этим содержимым. Другой пример, требующий обеспечения взаимодействия между фильмами, — создание обучающей программы для двухмониторной системы, когда на один монитор должна выводиться навигационная панель, служебные элементы и вспомогательная информация, а на другой — содержимое текущего раздела.

Взаимодействие между документами может быть реализовано с помощью класса `LocalConnection`, причем взаимодействовать между собой могут любые два Flash-документа вне зависимости от того, воспроизводятся ли они на HTML-странице в браузере, в автономном проигрывателе или в среде тестирования Flash.

Механизм использования класса `LocalConnection` довольно прост, однако организация взаимодействия между двумя файлами требует размещения соответствующего программного кода в обоих файлах, что придает всей процедуре некоторую запутанность. В связи с этим мы сначала рассмотрим процесс взаимодействия в общих чертах, а затем приведем более подробное описание программного инструментария, используемого для реализации этого процесса.

Механизм взаимодействия

Во взаимодействии, реализуемом с помощью класса `LocalConnection`, всегда принимают участие два документа. Один документ, отправляющий данные, мы будем называть *отправителем* (sending document), другой документ, получающий данные, — *получателем* (receiving document). В принципе, один и тот же документ может одновременно и отправлять данные, и получать данные от другого документа. В этом случае он является отправителем для первого документа и получателем — для второго. Обмен информацией может выполняться даже в пределах одного-единственного документа, однако такая возможность вряд ли имеет практическое применение. Два документа также могут осуществлять взаимный обмен информаци-

ей, при этом каждый документ в различные моменты может быть и отправителем, и получателем информации.

Для получения возможности передавать или принимать информацию в документе должен быть создан экземпляр класса `LocalConnection`. Объект `LocalConnection` создается и в документе-получателе, и в документе-отправителе. Для создания объекта применяются оператор `new` и не принимающий параметров конструктор. В общем случае при этом используется следующий синтаксис[2]:

```
// Создание объекта LocalConnection в документе-отправителе
var lcSender = new LocalConnection();
// Создание объекта LocalConnection в документе-получателе
var lcReceiver = new LocalConnection();
```

В приведенном примере объекты `LocalConnection` в разных документах сохраняются в переменных с разными идентификаторами. Это не является обязательным. Ссылки на объект `LocalConnection` в документе-отправителе и в документе-получателе могут сохраняться в переменной с одним и тем же именем. Разные имена просто делают код более понятным и логичным.

Взаимодействие между файлами выполняется путем установки между ними соединения с заданным именем. Файл-получатель открывает соединение, с помощью которого ему будут передаваться данные от файла-отправителя. Тот, в свою очередь, подключается к соединению с заданным именем, открытому файлом-получателем, и отправляет ему данные, инициируя вызов объявленных в этом файле методов. В общих чертах этот процесс состоит из следующих этапов.

1. В документе-получателе для объекта `LocalConnection` создается набор методов, каждый из которых может быть вызван при получении соответствующего распоряжения от файла-отправителя.
2. Документ-получатель открывает соединение, используя метод `connect()`, вызванный созданным в нем объектом `LocalConnection`. Открытие соединения — это процедура, подготавливающая файл-получатель к приему данных от файла-отправителя. При вызове данного метода ему в качестве параметра передается имя открываемого соединения. Таким образом, объект-получатель будет реагировать только на обращения отправителя, отправленные посредством соединения с этим именем.
3. В файле-отправителе вызывается метод `send()` объекта `LocalConnection`. Данный метод осуществляет вызов метода, объявленного в файле-получателе, используя для этого соединение, открытое получателем.

---

[2] В приведенных примерах при создании объекта `LocalConnection` намеренно не используется строгая типизация, поскольку в исходной версии Flash v7.0 встроенный класс `LocalConnection` был объявлен без атрибута `Dynamic`, что не позволяло добавлять свойства его экземплярам. Вследствие этого попытка добавления метода объекту `LocalConnection` приводила к ошибке компиляции. Очевидно, это явилось следствием ошибки, допущенной разработчиками, которая была исправлена в вышедшем позднее обновлении Flash v7.2, под названием Ellipsis. На страницах этой книги строгая типизация объекта `LocalConnection` не используется во избежание возможных ошибок компиляции. — *Прим. автора*.

Описанный механизм взаимодействия в виде схемы представлен на рис. 20.2.

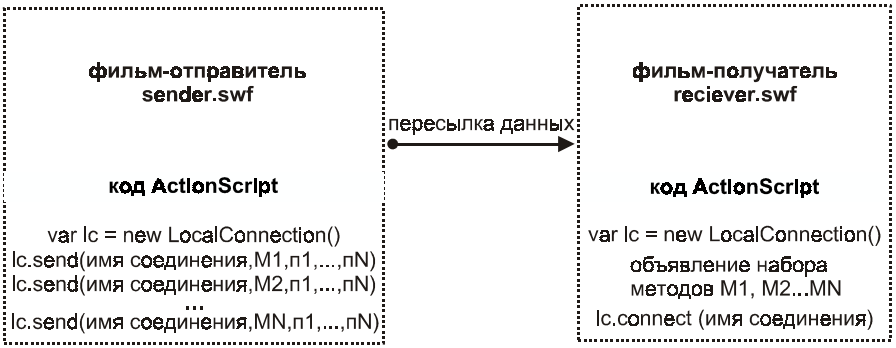


Рис. 20.2. Схема взаимодействия SWF-документов посредством класса `LocalConnection`

Инструментарий класса *LocalConnection*

В данном разделе будет описано применение методов и обработчиков событий класса `LocalConnection`.

Объявление методов объекта *LocalConnection* в документе-получателе

Взаимодействие между документами выражается в том, что файл-отправитель вызывает один или несколько методов, объявленных в файле-получателе. Для того чтобы документ-получатель мог реагировать на принимаемые от документа-отправителя сообщения, для созданного в нем объекта `LocalConnection` необходимо объявить соответствующие методы. При этом используется следующий синтаксис:

```
lc.метод = function(п1, п2, ..., пN){
блок кода;
...
};
```

Здесь *lc* — это объект `LocalConnection`, созданный в файле-получателе.

метод — это произвольное имя метода. Параметры *п1, п2, ..., пN* не являются обязательными, но могут быть указаны в случае, если документ-отправитель вызывает соответствующий метод в документе-получателе, передавая ему параметры. В случае отсутствия формальных параметров доступ к фактическим параметрам из тела метода может быть получен с помощью объекта `arguments`.

Следует отметить, что если при создании объекта LocalConnection использовалась строгая типизация, то объявление метода должно выполняться с помощью оператора [] *(см. сноску на стр. 745)*.

```
lc["метод"] = function(п1, п2, ..., пN){
блок кода;
...
};
```

Ниже приводится код документа-получателя, создающий объект LocalConnection и добавляющий этому объекту метод showMessage(), который выводит переданный ему в виде строки параметр в динамически созданное текстовое поле msgTxt.

```
var lcReciever = new LocalConnection();
lcReciever.showMessage = function(msg:String):Void {
        _root.createTextField("msgTxt", 1, 0, 0, 0, 0);
        msgTxt.autoSize = true;
        msgTxt.text = msg;
};
```

Открытие соединения. Метод *connect()*

Для реализации взаимодействия между файлами необходимо, чтобы файл-получатель находился в состоянии готовности к приему передаваемых ему файлом-отправителем сообщений. Для того чтобы привести его в такое состояние, используется метод connect(), который вызывается с помощью объекта LocalConnection с применением следующего синтаксиса:

```
lc.connect(имя соединения)
```

Здесь *lc* — это объект LocalConnection, созданный в файле-получателе.

Параметр *имя соединения* — это строка, указывающая имя соединения, посредством которого файл-отправитель обращается к файлу-получателю. Получатель может принимать данные только по соединению, к которому он подключен с помощью метода connect(). Необходимо отметить, что один объект-получатель может быть подключен только к одному соединению. В строке *имя соединения* запрещается использовать двоеточие, причины этого будут объяснены далее.

Метод connect() возвращает значение true, если выполнены оба следующих условия:

- на момент вызова метода connect() объект *lc* не является подключенным к другому соединению;
- на момент вызова метода connect() отсутствует другое открытое соединение с тем же самым значением параметра *имя соединения*, открытое другим объектом LocalConnection.

Если хотя бы одно из этих условий оказывается не выполнено, метод connect() возвращает значение false. Наличие двух одновременно открытых соединений с

одинаковыми именами приведет к тому, что данные от файла-отправителя будет получать только получатель, первым открывший соединение.

Отправка данных получателю

Для того чтобы файл-отправитель смог инициировать вызов метода, объявленного в файле-получателе, нужно вызвать метод `send()` объекта `LocalConnection` в файле-отправителе. Метод `send()` вызывает метод, объявленный в файле-получателе, используя открытое им соединение, и позволяет передать вызываемому методу набор параметров. Вызов метода `send()` имеет следующий синтаксис:

lc.send(*имя соединения*, *метод*, *п1*, *п2*, ..., *пN*)

Здесь *lc* — это объект `LocalConnection`, созданный в файле-отправителе.

Параметр *имя соединения* — это строка, задающая имя соединения, используемого для передачи данных. Эта строка должна совпадать с именем соединения, указанного при вызове метода `connect()` в файле-получателе.

Параметр *метод* — это строка, задающая имя метода, объявленного в файле-получателе, который требуется вызвать. Следует обратить внимание на то, что данный параметр (как и предыдущий) задается строкой и не должен содержать оператора вызова функции `()`.

Остальные параметры *п1*, *п2*, ..., *пN* — это фактические параметры, которые должны быть переданы методу `метод` при его вызове. Если метод файла-получателя не принимает параметров, они просто не указываются.

Метод `send()` возвращает булево значение `true`, если его вызов является синтаксически корректным. В противном случае метод `send()` возвращает значение `false`.

Примечание

Следует иметь в виду, что объем информации, передаваемой методом `send()`, ограничен, поэтому, если в результате его вызова получается значение `false` в то время как синтаксис является корректным, можно попробовать разбить передачу данных на два этапа.

Важно понимать, что значение `true`, полученное в результате вызова метода `send()`, не означает, что файлу-отправителю удалось успешно подключиться к соединению и отправить данные файлу-получателю. Значение `true` свидетельствует только о синтаксической корректности вызова метода `send()`. Так, например, если метод `send()` пытается передать данные по несуществующему соединению, то в результате его вызова (при отсутствии синтаксических ошибок) будет получено значение `true`, однако передачи данных не произойдет.

Для того чтобы определить, произошла ли передача данных, инициированная методом `send()`, можно воспользоваться обработчиком события `onStatus` класса `LocalConnection`. Событие `onStatus` генерируется объектом `LocalConnection` файла-отправителя, если метод `send()` возвращает значение `true` (то есть если его вызов является синтаксически корректным).

Определение метода обработчика события `onStatus` имеет следующий синтаксис:

```
lc.onStatus = function(информационный объект){
блок кода;
};
```

Параметр `информационный объект` — это объект, который содержит свойство `level`, позволяющее определить, удалось ли объекту `LocalConnection` файла-отправителя осуществить отправку инструкций получателю.

Свойство `level` может принимать одно из двух строковых значений. В зависимости от этого значения могут быть выполнены те или иные инструкции в теле метода обработчика.

- Значение `"status"` присваивается свойству `level` в случае успешной отправки инструкций получателю. Это, однако, не означает, что данные инструкции были получены и исполнены файлом-получателем. Так, например, инструкции отправителя могут быть проигнорированы, если файл-получатель расположен в другом домене и не допускает взаимодействия с файлами, полученными с домена, на котором расположен файл-отправитель. Инструкции отправителя также не будут выполнены, если у объекта `LocalConnection` файла-получателя отсутствуют вызываемые отправителем методы. Таким образом, значение просто свидетельствует о том, что метод `send()` был успешно выполнен, однако не дает никаких гарантий, что его выполнение отразилось на содержимом документа-получателя.
- Значение `"error"` присваивается свойству `level`, если вызов метода `send()` является синтаксически корректным, однако не может быть выполнен. Причиной этого может быть отсутствие открытого соединения с соответствующим именем или слишком большой объем передаваемых данных (что чаще всего маловероятно).

Как мы видим, применение обработчика `onStatus` не дает возможности точно определить, был ли вызван целевой метод в файле-получателе. Для решения этой проблемы необходимо в случае успешного вызова метода в файле-получателе отправить обратно сообщение файлу-отправителю (для чего в файле-отправителе должны быть определены соответствующие методы). Пример организации подобного взаимодействия будет представлен далее в листинге 20.4.

В качестве иллюстрации возможностей класса `LocalConnection` приведем следующий пример. Имеется два SWF-документа. В одном из них (документ-отправитель) находится изображение, в другом (документ-получатель) отображается увеличенный фрагмент изображения, над которым в данный момент в документе-отправителе находится курсор, и изображение рамки, указывающей увеличиваемую область (рис. 20.3). Подобный пример можно применить при создании виртуальной галереи или интерактивной карты.

В документе-отправителе (s.swf) находится экземпляр растрового изображения, уменьшенный по сравнению с оригиналом в 4 раза. При перемещении мыши над изображением перемещается рамка, представленная клипом. В окне документа-получателя (r.swf) отображается фрагмент оригинального (укрупненного) изображения соответствующий области, расположенной внутри рамки `frame`. В документе-

отправителе также имеется текстовое поле с именем `levelTxt`, в которое выводится информация о состоянии соединения.



Рис. 20.3. Интерактивный навигатор

Принцип работы данного примера заключается в том, что при перемещении рамки `frame` в документе-отправителе на основании ее координат вычисляются координаты увеличенного изображения. Эти значения передаются документу-получателю в качестве параметров метода `setMove()` с помощью соединения с именем `moveConnection`. В документе-получателе клип `pic`, содержащий увеличенное в 4 раза изображение, смещается на расстояние, соответствующее изменению координат рамки в фильме-отправителе. Точка регистрации клипа `frame` находится в его центре, а точка регистрации клипа `pic` находится в его левом верхнем углу.

В листинге 20.3 приведен код, реализующий описанную функциональность (в листинге представлены сценарии обоих документов).

Листинг 20.3. Интерактивный навигатор

```
//---------------Сценарий документа-отправителя----------------------
var lc = new LocalConnection();
/* Функция, выполняющая перемещение рамки и пересылку значений смещений
увеличенного изображения файлу-получателю */
function followMouse() {
        var x = frame._x = this._xmouse;
        var y = frame._y = this._ymouse;
        var parX = x - frame._width / 2;
        var parY = y - frame._height / 2;
        lc.send("moveConnection", "setMove", parX, parY);
}
frame.onPress = function() {
        followMouse();
```

```
        this._parent.onMouseMove = function() {
                followMouse();
                lc.onStatus = function(infoObj) {
                        if (infoObj.level == "status") {
                                levelTxt.text = "Соединение...";
                                updateAfterEvent();
                        }
                        else if (infoObj.level == "error") {
                                levelTxt.text = "Соединение отсутствует";
                        }
                };
        };
};
frame.onRelease = function() {
        delete this._parent.onMouseMove;
};
//--------------Сценарий документа-получателя-------------------------
var lc = new LocalConnection();
lc.setMove = function(parX, parY) {
        pic._x = -parX*4;
        pic._y = -parY*4;
};
lc.connect("moveConnection");
```

Закрытие соединения

Для того чтобы закрыть соединение, то есть сделать так, чтобы принимающий объект `LocalConnection` перестал ожидать передачи инструкций от объекта-отправителя, можно воспользоваться методом `close()`. Данный метод имеет следующий синтаксис:

```
lc.close()
```

Здесь *`lc`* — это объект `LocalConnection`, созданный в файле-получателе.

Применение данного метода приводит к удалению соединения, открытого вызовом метода `connect()`. Это дает возможность открытия нового соединения объектом *`lc`* или открытия нового соединения с тем же именем другим объектом `LocalConnection`.

Пример организации взаимодействия документов

Как уже говорилось, единственная возможность, позволяющая в файле-отправителе определить успешность вызова методов в файле-получателе, заключается в отправке отправителю ответного сообщения от получателя. При этом оба документа выполняют функции и отправителя, и получателя. Во избежание путаницы мы будем по-прежнему называть получателем документ, который первым получает запрос на выполнение инструкций, а отправителем — документ, первым отправляющий такой запрос. Очевидно, что для реагирования на инструкции, отправленные получателем в результате выполнения его методов, в файле-отправителе должен быть объявлен соответствующий метод, определяющий ответные действия. Отправка инструкций получателю и отправка ответных инструкций отправителю должны выполняться с помощью разных соединений, однако в обоих документах для приема и отправки данных может быть использован один и тот же объект `LocalConnection`.

Рассмотрим несложный пример подобного взаимодействия. На рис. 20.4 представлена схема Web-страницы, содержащей два SWF-документа. Документ recievingMovie.swf содержит покадровую анимацию, а документ sendingMovie.swf содержит панель управления (контроллер) этой анимацией. Нажатие кнопки на панели управления должно отразиться на состоянии анимации. В то же время, получив команду от отправителя, файл с анимацией отправляет ему сообщение, отражающее выполненные действия. Это сообщение выводится в текстовое поле `statusTxt`, расположенное на панели навигации.

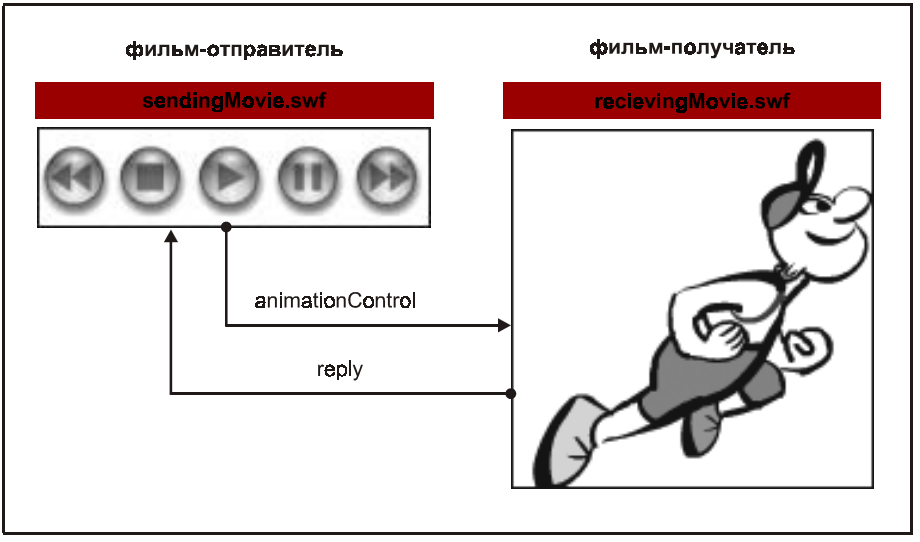


Рис. 20.4. Пример организации взаимодействия между файлами

Для вызова методов объекта-получателя, управляющих анимацией, используем соединение с именем `animationControl`. Передача данных от получателя обратно от-

правителю для вывода сообщения в текстовое поле выполняется с помощью соединения с именем reply.

Сценарии, реализующие взаимодействие, приведены в листинге 20.4.

Листинг 20.4. Организация взаимодействия между файлами

```
//СЦЕНАРИЙ ДОКУМЕНТА-ПОЛУЧАТЕЛЯ (resievingMovie.swf)
//Создание объекта LocalConnection
lc = new LocalConnection();
/*Объявление методов, управляющих воспроизведением текущего фильма и
осуществляющих отсылку ответа обратно в документ с панелью управления*/
lc.stopAnim = function() {
       runner.stop();
       /*"reply" - имя соединения, "showMessage" - метод, объявленный в
       документе с анимацией, осуществляющий вывод информации в текстовое
       поле, "Stop" - параметр, передаваемый методу "showMessage" для
       вывода в текстовое поле*/
       lc.send("reply", "showMessage", "Stop");
};
lc.playAnim = function() {
       runner.play();
       lc.send("reply", "showMessage", "Play");
};
lc.toBeginning = function() {
       runner.gotoAndStop(1);
       lc.send("reply", "showMessage", runner._currentframe);
};
lc.forward = function() {
       runner.nextFrame();
       lc.send("reply", "showMessage", runner._currentframe);
};
lc.backward = function() {
       runner.prevFrame();
       lc.send("reply", "showMessage", runner._currentframe);
};
/*Открытие соединения с именем "animationControl" для управления
анимацией*/
lc.connect("animationControl");
/*При щелчке по бегуну очищаем текстовое поле в документе с панелью
управления*/
runner.onPress = function() {
```

```
        lc.send("reply", "showMessage", "");
};
//СЦЕНАРИЙ ДОКУМЕНТА-ОТПРАВИТЕЛЯ (sendingMovie.swf)
//Создание объекта LocalConnection
var lc = new LocalConnection();
//Контроль панели управления воспроизведением
pauseBut.onPress = function() {
        /* Отправка объекта с названием animationControl для вызова метода
        stopAnim */
        lc.send("animationControl", "stopAnim");
};
playBut.onPress = function() {
        lc.send("animationControl", "playAnim");
};
stopBut.onPress = function() {
        lc.send("animationControl", "toBeginning");
};
fwBut.onPress = function() {
        lc.send("animationControl", "forward");
};
rwBut.onPress = function() {
        lc.send("animationControl", "backward");
};
/*Объявление метода, выводящего в текстовое поле сообщение, передаваемое
данному методу в качестве параметра/
lc.showMessage = function(msg:String) {
        statusTxt.text = msg;
};
/*Установление соединения с названием "reply", осуществляющего отсылку
ответа обратно в документ, отправляющий команды управления анимацией*/
lc.connect("reply");
```

Вопросы политики безопасности, связанные с взаимодействием документов

Политика безопасности, поддерживаемая проигрывателем Flash Player 7, по умолчанию не позволяет организовывать взаимодействие между файлами, загруженными с разных доменов. Взаимодействие между документами по умолчанию разрешается только при условии полной идентичности их доменов. Это означает, что файл-

отправитель сможет вызвать метод, объявленный в документе-получателе, загруженном из домена **www.wherever.com**, только в том случае, если отправитель также загружен из домена **www.wherever.com**. Полная идентичность доменов означает, что документ, загруженный из домена **www.wherever.com**, не сможет получить доступа к методам документа, загруженного из домена **wherever.com**[3]. Для того чтобы разрешить взаимодействие между файлами, расположенными в разных доменах, необходимо воспользоваться обработчиком события `allowDomain` класса `LocalConnection`.

Кроме ограничения, связанного с идентичностью доменов, в Flash Player 7 имеется ограничение идентичности протоколов. Данное ограничение выражается в том, что файл, загруженный с сайта, использующего протокол HTTP, не сможет взаимодействовать с файлом, загруженным с сайта по защищенному протоколу HTTPS. Для преодоления этого ограничения можно воспользоваться обработчиком события `allowInsecureDomain` класса `LocalConnection`.

Доменная безопасность

Рассмотрим механизм, в соответствии с которым Flash Player проверяет условие идентичности доменов.

При открытии соединения документом-получателем строка параметра, переданная методу `connect()`, разрешается проигрывателем Flash Player в строку, состоящую из имени домена, на котором расположен файл-получатель, и имени соединения, разделенных точкой с запятой. Так, например, если файл-получатель, загруженный с домена **www.somewhere.com**, открыл соединение с именем `myConnection`, то Flash Player автоматически сформирует строку следующего вида:

```
somewhere.com:myConnection
```

Если отрывающий соединение файл расположен локально, вместо имени домена используется слово **localhost**, и сформированная строка имеет следующий вид:

```
localhost:myConnection
```

Аналогичная процедура выполняется и в документе-отправителе. При вызове метода `send()` к имени соединения, указанному в качестве параметра, присоединяется имя домена, в котором находится файл-отправитель. После этого выполняется сравнение полученных строк для документа-получателя и документа-отправителя, и в случае их идентичности Flash Player разрешает взаимодействие между файлами.

В большинстве случаев взаимодействующие друг с другом файлы расположены в одном и том же домене, поэтому описанный механизм проверки идентичности доменов позволит таким файлам свободно взаимодействовать. Тем не менее, если требуется организовать взаимодействие между файлами с разных доменов, то сформи-

[3] В предыдущей версии Flash Player 6 требования идентичности доменов были менее строгими. Для разрешения взаимодействия между документами достаточно было выполнения условия идентичности родительских доменов (то есть доменов более высокого уровня). Так, в 6-ой версии файл, загруженный с **www.wherever.com**, мог получить доступ к файлу, загруженному с **www.movie.wherever.com**, в то время как в Flash Player 7 такая возможность по умолчанию отсутствует. — *Прим. автора*.

рованные проигрывателем строки не будут идентичными и, соответственно, взаимодействие между ними будет запрещено.

Для того чтобы разрешить взаимодействие между файлами с разных доменов, необходимо сделать так, чтобы имя домена не присоединялось к имени соединения. Для этих целей можно воспользоваться одним из следующих подходов.

- В документе-отправителе и в документе-получателе в начале имени соединения необходимо поместить символ нижнего подчеркивания (например, `_myConnection`). В результате этого Flash Player не будет добавлять имя домена к имени соединения. Таким образом, если в файле-отправителе и в файле-получателе будет указано одно и то же значение имени соединения, строки, сформированные при проверке идентичности доменов, будут одинаковыми. Для того чтобы проигрыватель разрешил осуществлять взаимодействие таким файлам в документе-получателе, необходимо определить метод обработчика события `allowDomain`, указывающий домены, файлам, с которых будет разрешен доступ к методам принимающего объекта `LocalConnection`. При использовании данного подхода перемещение принимающего файла в другой домен не приведет к утрате возможности взаимодействия (то есть при этом не нужно изменять код в файле-отправителе).
- В качестве имени соединения в документе-отправителе при вызове метода `send()` необходимо указать строку, включающую имя домена, в котором будет размещен данный файл, и имя соединения, разделенные двоеточием (например, `wherever.com:myConnection`). При этом Flash Player также не будет вносить изменений в эту строку. Отметим, что выполнить подобную операцию в файле-получателе нельзя. Для того чтобы проигрыватель разрешил осуществлять взаимодействие между файлами в документе-получателе, необходимо определить метод обработчика события `allowDomain`, указывающий домены, файлам, с которых будет разрешен доступ к методам принимающего объекта `LocalConnection`. При использовании данного подхода перемещение принимающего файла в другой домен приведет к необходимости изменения кода в файле-отправителе.

Для реализации взаимодействия документов, полученных из различных доменов, недостаточно идентичности строк из имени домена и соединения, сформированных проигрывателем. В файле-получателе необходимо также явно указать, документам с каких доменов разрешено осуществлять вызов методов объекта-получателя. Для того чтобы Flash Player разрешил взаимодействие между файлами, загруженными из разных доменов, в документе-получателе необходимо определить метод обработчика события `allowDomain`. Данное событие генерируется при получении запроса на вызов методов принимающего объекта `LocalConnection` от файла-отправителя. С помощью обработчика `allowDomain` можно указать отправляющим файлам, с каких доменов разрешается вызывать методы принимающего объекта `LocalConnection`. Определение метода `allowDomain()` имеет следующий вид:

```
объект-получатель.allowDomain = function(домен отправителя) {
блок кода, возвращающий true или false;
};
```

Параметр *домен отправителя* — это строка, указывающая домен, в котором находится файл-отправитель. Данный параметр является необязательным. Его нужно использовать для того, чтобы указать конкретные домены файлам, с которых разрешается вызывать методы объекта *объект-получатель*. Следует отметить, что при публикации для Flash Player 6 параметр *домен отправителя* возвращает имя наддомена, в то время как при публикации для Flash Player 7 параметр *домен отправителя* возвращает точное имя домена. Так, если документ загружен из домена **www.wherever.com**, то в первом случае параметр будет представлять собой строку:

```
"wherever.com",
```

а во втором:

```
"www.wherever.com"
```

Код, помещенный внутрь фигурных скобок, должен возвращать булево значение `true` или `false`. Если возвращаемым значением является `true`, Flash Player разрешает вызов методов файла-получателя. Если возвращаемым значением является `false`, Flash Player запрещает вызов методов файла-получателя, и взаимодействие не реализуется.

Для определения имени домена, в котором расположен SWF-документ, может быть использован метод `domain()` класса `LocalConnection`, имеющий следующий синтаксис:

```
lc.domain()
```

Здесь *lc* — это объект *LocalConnection*.

Данный метод возвращает строку, содержащую имя домена, в котором расположен файл, содержащий объект *lc*. Для локально расположенных файлов данный метод возвращает строку `"localhost"`. При публикации для Flash Player 7 строка, возвращаемая данным методом, представляет точный домен, из которого был получен файл (например, **www.wherever.com**). При публикации для Flash Player 6 данная строка содержит наддомен (например, **wherever.com**).

По умолчанию файлы из одного домена могут взаимодействовать без ограничений. Этой ситуации соответствует следующий код определения обработчика `allowDomain`.

```
my_lc.allowDomain = function (sendingDomain) {
return (sendingDomain == this.domain());
};
```

Следующий пример позволяет вызывать методы объекта-получателя всем файлам, загруженным из домена **www.wherever.com** и из того же домена, на котором находится получатель:

```
my_lc.allowDomain = function (sendingDomain) {
return(sendingDomain==www.wherever.com || sendingDomain== this.domain());
};
```

Для того чтобы разрешить вызывать методы объекта-получателя файлам из любого домена, нужно просто поместить в тело метода обработчика код, возвращающий значение `true`:

```
my_lc.allowDomain = function () {
return true;
};
```

Безопасность протоколов

Для того чтобы Flash Player разрешил вызывать методы файла-получателя, загруженного по защищенному протоколу HTTPS, с помощью файла-отправителя, загруженного по протоколу HTTP, необходимо воспользоваться обработчиком события `allowInsecureDomain()`. Определение метода данного обработчика аналогично определению метода обработчика `allowDomain()`.

```
объект-получатель.allowInsecureDomain = function(домен отправителя) {
блок кода, возвращающий true или false;
};
```

Параметр *домен отправителя* — это строка, указывающая домен, в котором находится файл-отправитель.

Код, помещенный в тело метода, должен возвращать булево значение, на основании которого Flash Player разрешит или запретит взаимодействие между файлами.

Метод `allowInsecureDomain` вызывается, когда файл-получатель, загруженный по защищенному протоколу, получает запрос на вызов методов от документа-отправителя, загруженного по протоколу HTTP.

В Flash Player 6 не было ограничений, связанных с требованием идентичности протоколов. В связи с этим при взаимодействии файлов, опубликованных для Flash Player 6, проверка идентичности протоколов выполняться не будет, даже если фильмы воспроизводятся в Flash Player 7.

Следующий код разрешает доступ к файлу, загруженному по защищенному протоколу, файлам, загруженным по незащищенному протоколу, только в том случае, если они находятся в одном и том же домене.

```
lc.allowInsecureDomain = function(sendingDomain:String) {
return (sendingDomain == this.domain());
};
```

Глава 21

Класс *SharedObject*. Хранение информации на стороне клиента

Любознательный читатель наверняка интересовался назначением красивой цветной панели, появляющейся при выполнении команды **Settings...** контекстного меню проигрывателя Flash Player 7 (рис. 21.1). Название вкладки **Local Storage** (Локальное хранение данных) и предложение указать допустимый объем дискового пространства, используемого для хранения данных, недвусмысленно свидетельствуют о наличии возможности сохранять информацию из Flash-фильма на машине клиента.



Рис. 21.1. Панель настроек **Flash Player 7**. Вкладка **Local Storage**

Такая возможность действительно существует и, начиная с шестой версии Flash (Flash MX), для ее реализации применяется встроенный класс `SharedObject`. В настоящей главе мы подробно изучим инструментарий класса `SharedObject` и рассмотрим вопросы, связанные с хранением информации на стороне клиента.

Понятие об объекте *SharedObject*

Сохранение информации из Flash-фильма на диске пользователя осуществляется с помощью объекта `SharedObject`. Все рассмотренные нами до сих пор объекты создавались в процессе воспроизведения Flash-фильма и уничтожались при закрытии окна проигрывателя. В отличие от них объект `SharedObject`, будучи создан в про-

цессе воспроизведения фильма, может быть в дальнейшем сохранен на диске компьютера, на котором воспроизводится фильм, и продолжит свое существование даже после того, как окно проигрывателя будет закрыто. При следующем запуске фильма его сценарий сможет "отыскать" сохраненный на компьютере пользователя объект `SharedObject` и считать сохраненные в нем данные.

Этот механизм напоминает применение cookies — текстовых файлов, создаваемых браузером, в которых хранится информация о недавно посещенных ресурсах. Объекты `SharedObject` сохраняются проигрывателем в текстовых файлах, имеющих расширение sol.

Ограничения локального хранения информации

Сохранение информации на стороне клиента имеет два существенных ограничения, введенных с целью обеспечения безопасности.

- Пользователь в любой момент может ограничить объем дискового пространства, предоставляемого его компьютером для хранения всех объектов `SharedObject`, созданных файлами из одного домена. Пользователь также всегда может удалить все хранимые на его компьютере объекты `SharedObject`.
- По умолчанию объекты `SharedObject`, созданные SWF-документами, загруженными из одного домена, недоступны SWF-документам, загруженным из другого домена. Аналогично, локально размещенные файлы по умолчанию не могут обращаться к объектам `SharedObject`, созданным другими файлами.

По умолчанию Flash Player может использовать не более 100 Кбайт дискового пространства для файлов, осуществляющих локальное хранение информации, полученных из одного домена. Если объем сохраняемой информации превышает данное значение, на экран выводится панель настроек проигрывателя, предлагающая пользователю разрешить (**Allow**) увеличение выделенного дискового пространства или запретить (**Deny**) его увеличение (рис. 21.2). В последнем случае информация сохранена не будет.

При желании пользователь может сам определить, какой объем информации он разрешает сохранять файлам с определенного домена на своем компьютере. Для этого достаточно щелкнуть правой кнопкой мыши внутри окна проигрывателя и из появившегося контекстного меню выбрать команду **Settings...**. В результате выполнения этой команды появится панель, представленная на рис. 21.1.

Примечание

Панель настроек проигрывателя выводится только при условии, что окно Flash Player имеет размеры не менее чем 215×138 пикселов. Это ограничение имеет место и для вывода панели по требованию пользователя, и для автоматического вывода. Если размеры окна проигрывателя оказываются меньше указанных значений, команда **Settings...** в контекстном меню проигрывателя будет неактивной. Также вывода панели не произойдет и при нехватке места на диске, в результате чего информация не сможет быть сохранена.

При необходимости можно программным образом вывести пользователю панель с настройками, позволяющими задать допустимый лимит, используемый для локаль-

ного хранения информации. Для этого можно воспользоваться статическим методом `System.showSettings()`, передав ему в качестве параметра число 1:

```
System.showSettings(1);
```



Рис. 21.2. Сообщение о нехватке дискового пространства, отведенного для локального хранения информации

Числовой параметр указывает, какая именно вкладка панели **Settings** будет выведена в результате вызова данного метода. Кроме вкладки **Local Storage**, которой соответствует значение параметра 1, здесь имеются вкладки **Privacy** (0), **Microphone** (2) и **Camera** (3). Эти вкладки позволяют пользователю разрешить или запретить доступ к установленным у него микрофону и камере и настроить параметры данного оборудования.

На вкладке **Local Storage** с помощью горизонтального слайдера можно установить допустимый объем, предоставляемый для локального хранения информации файлами из того же домена, в котором находится текущий документ. Установка ползунка в значение **None** (крайнее левое положение) запретит локальное хранение информации и приведет к удалению всех хранящихся в настоящий момент на диске объектов `SharedObject`, созданных документами из того же домена, откуда был получен текущий документ. Установка ползунка в значение **Unlimited** (крайнее правое положение) полностью снимет ограничение на объем, позволив проигрывателю локально сохранять всю необходимую информацию из данного домена. Установка флажка **Never Ask Again** приведет к запрету локального сохранения информации файлами из того же домена, из которого был получен текущий документ. При этом панель, сообщающая о необходимости предоставления дополнительного пространства на диске, выводиться не будет и, соответственно, в этой ситуации класс `SharedObject` окажется совершенно бесполезным.

Все SOL-файлы, используемые для хранения содержимого объектов `SharedObject`, сохраняются проигрывателем в папке с названием Flash Player, местоположение которой зависит от используемой операционной системы. Так, путь к этой папке в Windows 98 будет иметь примерно следующий вид:

C:\WINDOWS\ApplicationData\Macromedia\Flash Player

В операционной системе Windows 2000 или Windows XP путь будет выглядеть так:

C:\Documents and Settings\user\ApplicationData\Macromedia\Flash Player

Для того чтобы избежать конфликтов, связанных с тем, что объекты `SharedObject`, создаваемые разными документами, могут иметь одинаковые имена, проигрыватель сохраняет их не непосредственно в корневую папку Flash Player, а формирует структуру вложенных папок. Эта структура в точности отражает местоположение файла, сохраняющего данные в доменной структуре (если файл загружен по сети) или в файловой системе (если файл расположен локально). Путь к SOL-файлу, хранящему содержимое объекта `SharedObject`, созданного Flash-фильмом, относительно каталога Flash Player в точности соответствует URL или локальному пути этого фильма. Так, например, SOL-файл, созданный фильмом, загруженным с **www.wherever.com/movies/m1.swf**, будет сохранен в папке Flash Player в каталоге, путь к которому будет иметь следующий вид: **www.wherever.com/movies/m1.swf**.

В самой нижней для этой структуры папке (m1.swf) будет сохранен файл с расширением sol, имя которого будет соответствовать имени объекта `SharedObject`, указанному при его создании.

Для локально расположенных файлов (то есть файлов, хранящихся в файловой системе компьютера пользователя) в папке Flash Player создается папка с именем localhost, в которой формируется структура вложенных папок, отражающая местоположение Flash-фильма, сохраняющего данные в файловой системе. Так, например, файл, расположенный по адресу **C:\FlashWorks\Projects\Cartoons\go\part1.swf**, будет сохранять содержимое объекта `SharedObject` в папке, расположенной в каталоге localhost по следующему адресу:

FlashWorks\Projects\Cartoons\go\part1.swf

Тот факт, что каждый объект обладает собственным именем и по умолчанию сохраняется в каталоге, соответствующем создающему его файлу, позволяет избежать конфликтов имен и исключает возможность обращения к объектам `SharedObject`, созданным одним файлом со стороны других файлов. Тем не менее при необходимости можно сохранить объект `SharedObject` в заданной папке (в пределах каталога, соответствующего домену создавшего его файла и расположенного в папке Flash Player), сделав его доступным другим SWF-документам.

Вопросы безопасности локального хранения информации

В соответствии с новыми правилами политики безопасности, вступившими в силу с выходом седьмой версии Flash Player, файлы локальных объектов сохраняются в каталоге, имя которого в точности соответствует имени домена, из которого был получен файл, выполняющий локальное сохранение информации. Тем не менее в шестой версии Flash Player использовались другие правила политики безопасности, в соответствии с которыми локальные объекты сохранялись в каталоге, имя которого формировалось на основе имени домена верхнего уровня, а не точно того домена, из которого был получен записывающий их документ. Так, при использовании правил политики безопасности Flash Player 6 файлы локальных объектов, созданных документами, расположенными в доменах **www.aaa.index.com**, **www.bbb.index.com** и **www.index.com**, будут сохранены в одной и той же папке с именем index.com. Это означает, что файлы, расположенные в одном и том же родительском домене, могут использовать общие локальные объекты. По умолчанию

документы, созданные для Flash Player 7, используют правила точного соответствия доменов, а документы, созданные для Flash Player 6, используют правила совпадения только доменов верхнего уровня.

Для обеспечения обратной совместимости с документами, опубликованными для Flash Player 6, можно программным образом приказать проигрывателю Flash Player 7 применять правила, используемые в шестой версии. Для этих целей применяется статическое свойство `System.exactSettings`, принимающее булево значение. Если данное свойство принимает значение `true`, проигрыватель будет использовать правила точного соответствия доменов. Если значение данного свойства установить в `false`, проигрыватель будет использовать правила, соответствующие политике безопасности, применяемой в Flash Player 6. В этом случае все локальные объекты, созданные файлами из одного и того же родительского домена, будут помещены в один и тот же каталог. В основном значение `false` свойству `System.exactSettings` следует присваивать в документах, которые публикуются для Flash Player 7, но должны иметь доступ к локальным объектам, созданным документами, опубликованными для Flash Player 6. Такая необходимость возникает, например, при обновлении сайта, когда разработчик желает использовать новые возможности Flash, но в то же время обеспечить взаимодействие с остальным, необновленным содержимым проекта.

При необходимости использования свойства `System.exactSettings` присвоить ему требуемое значение необходимо в первом кадре и до выполнения всех операций, связанных с доступом к локально сохраненной информации и настройкам ее хранения. Так, например, после вызова метода `SharedObject.getLocal()` изменить значение свойства `System.exactSettings` будет невозможно.

Необходимо отметить, что если в проигрывателе одновременно находятся несколько документов, в которых используются разные значения свойства `System.exactSettings`, то для всех этих документов будут действовать одинаковые правила безопасности, определяемые тем, какой из загруженных документов последним выполнил соответствующую установку.

Создание объекта *SharedObject*

Процедура создания объекта `SharedObject` существенно отличается от традиционного способа создания экземпляров большинства объектов ActionScript. Это связано со спецификой данного объекта, а именно с тем, что его содержимое хранится на диске пользователя и должно быть доступно создавшему его документу.

Для создания объекта `SharedObject` применяется метод `getLocal()` класса `SharedObject`. Принцип работы этого метода заключается в следующем. При выполнении метод проверяет, имеется ли в соответствующем каталоге на диске пользователя SOL-файл, хранящий запрашиваемый объект `SharedObject`. Если такой файл существует, то метод `getLocal()` возвращает ссылку на хранящийся в нем объект `SharedObject`. Если файл не найден, метод `getLocal()` создает новый объект `SharedObject` и возвращает ссылку на него. Метод `getLocal()` является статическим и имеет следующий синтаксис:

`SharedObject.getLocal(`*имя*`, `*путь*`)`

Возвращаемая данным методом ссылка на объект SharedObject, как обычно, сохраняется в переменной, с помощью которой в дальнейшем можно осуществлять доступ к свойствам этого объекта и вызывать его методы. Если метод getLocal() не смог ни отыскать, ни создать объект SharedObject, он возвращает значение null. Такая ситуация может иметь место при задании параметрам этого метода некорректного (или неверного) значения.

Отметим, что метод getLocal() не сохраняет объект SharedObject на диск. Он либо считывает данные из SOL-файла, либо создает текущий объект SharedObject, который в дальнейшем может быть сохранен.

- Параметр *имя* — это строка, задающая имя локально сохраняемого объекта. Данное значение используется при задании имени SOL-файла, в котором сохраняется объект SharedObject. Так, если в качестве первого параметра методу getLocal() передается строка "mySharedObject", то данный метод возвратит ссылку на объект, сохраненный в файле mySharedObject.sol или, если такого файла не существует, создаст новый объект, который в дальнейшем будет сохранен на диске пользователя в файле с именем mySharedObject.sol. Строка, передаваемая этим параметром, не может содержать пробелы, а также следующие символы:

  ~ % & \ ; : " ' , < > ? #

  Если в качестве значения данного параметра передается пустая строка (""), вызов метода не дает результата. Если параметр не является строкой, он преобразуется в строку в соответствии с правилами преобразования типов, используемыми в ActionScript.

  Параметр *имя* является необязательным. В случае его отсутствия метод getLocal() выполнит обращение или создаст объект, который будет сохранен в файле с именем undefined.sol.

- Параметр *путь* — это строка, задающая путь относительно папки с именем домена, из которого был получен SWF-документ (или относительно папки localhost — для локально хранящихся фильмов), вложенной в папку Flash Player, по которому расположен (или будет сохранен) SOL-файл, содержащий объект SharedObject. Необходимо четко понимать, что задаваемый путь обязательно должен отражать доменную или файловую структуру, в которой находится SWF-документ, выполняющий вызов метода getLocal(). Данный параметр, как и предыдущий, является необязательным. Если параметр *путь* отсутствует, то путь к файлу, содержащему объект SharedObject относительно папки с именем, соответствующим имени домена, или относительно папки localhost, представляет собой точную копию адреса SWF-документа (рис. 21.3). Как альтернативу можно, используя параметр *путь*, в качестве каталога для размещения SOL-файла указать любой каталог, входящий в URL, или путь SWF-документа, осуществляющего вызов метода getLocal(). При этом Flash выполнит обращение к объекту (или в дальнейшем сохранит его) в папке, расположенной по указанному адресу, относительно папки с именем домена (или относительно папки localhost). Для того чтобы SOL-файл был сохранен непосредственно в папке, расположенной в каталоге с именем домена (или в localhost — для локальных файлов), необходимо в качестве параметра *путь* передать строку "/". Рассмотрим несколько примеров.

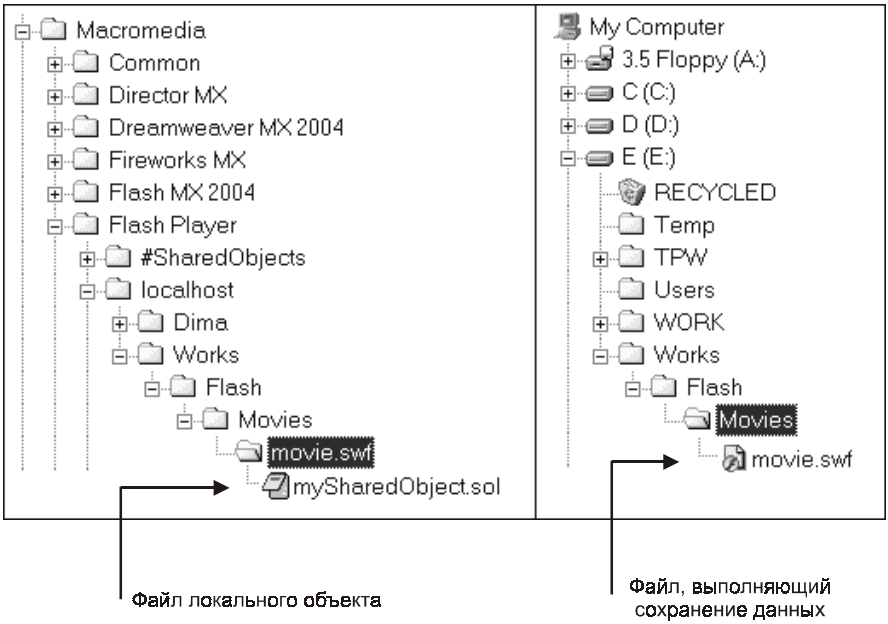


Рис. 21.3. Взаимосвязь местонахождения файла, осуществляющего локальное сохранение данных, и расположения созданного им SOL-файла

Допустим, документ, создающий локальный объект, расположен по адресу **www.wherever.com/movies/fun.swf**.

- Если метод `getLocal()` будет вызван без передачи ему параметра *путь*:

  ```
  var so:SharedObject = SharedObject.getLocal("soObj");
  ```

  В этом случае объект `SharedObject` будет сохранен в каталоге Flash Player по следующему пути:

  wherever.com/movies/fun.swf/soObj.sol

- Если метод `getLocal()` будет вызван с передачей ему следующего пути:

  ```
  var so:SharedObject = SharedObject.getLocal("soObj", "/movies");
  ```

  В этом случае объект `SharedObject` будет сохранен в каталоге Flash Player по следующему пути:

  wherever.com/movies/soObj.sol

- Если метод `getLocal()` будет вызван с передачей ему следующего пути:

  ```
  var so:SharedObject = SharedObject.getLocal("soObj", "/");
  ```

  В этом случае объект `SharedObject` будет сохранен в каталоге Flash Player по следующему пути:

  wherever.com/soObj.sol

Для локально размещенных файлов ситуация аналогичная. Так, пусть документ, создающий локальный объект, расположен по адресу **G:\Dima\Flash\Works\extra.swf**.

- Если метод `getLocal()` будет вызван без передачи ему параметра *путь*, объект `SharedObject` будет сохранен в каталоге **Flash Player** по следующему пути:

  localhost\Dima\Flash\Works\extra.swf\soObj.sol

- Если метод будет вызван с передачей ему следующего пути:

  ```
  var so:SharedObject = SharedObject.getLocal("soObj", "/Dima");
  ```

  В этом случае объект `SharedObject` будет сохранен в каталоге Flash Player по следующему пути:

  localhost\Dima\soObj.sol

Передача параметров методу `getLocal()` является необходимой в том случае, если к объектам `SharedObject`, созданным одним документом, необходимо получить доступ из другого документа, а также для того, чтобы сделать доступность объекта `SharedObject` менее зависимой от местоположения использующего его SWF-документа.

Так, для того чтобы сделать объект `SharedObject` доступным для всех SWF-документов, необходимо передать методу `getLocal()` в качестве второго параметра строку `"/"`. Правда, в этом случае появляется опасность возникновения конфликтов, связанных с наличием нескольких объектов `SharedObject` с одинаковыми именами. В ситуации, когда разные SWF-документы сохраняют в одном и том же каталоге объекты `SharedObject` с одинаковыми именами, они помещаются в один и тот же SOL-файл, включающий свойства обоих объектов. В такой ситуации при удалении свойства объекта `SharedObject` одним документом другой документ уже больше не может получить к нему доступ.

Следует помнить, что при вызове метода `getLocal()` без параметров дальнейшее перемещение SWF-документа приведет к тому, что он уже не сможет получить доступ к сохраненному им объекту `SharedObject`. Если же при вызове был задан параметр *путь*, то SWF-документ можно безболезненно перемещать на всем отрезке дерева каталогов, лежащем ниже уровня, имя которого соответствует имени каталога, в котором сохранен объект `SharedObject`.

Запись и чтение данных в объекте *SharedObject*

У объекта `SharedObject`, полученного в результате вызова метода `getLocal()`, имеется свойство с именем `data`. Это свойство представляет собой объект, свойства которого используются для локального сохранения данных и позволяют считывать данные из локально сохраненного объекта `SharedObject`.

При создании нового объекта `SharedObject` его объект `data` является пустым, то есть не содержит свойств. Для того чтобы поместить данные в объект `SharedObject`,

необходимо создать в объекте data соответствующие свойства и поместить в них эти данные:

```
var so:SharedObject = SharedObject.getLocal("so");
so.data.prop1 = "Flash";
so.data.prop2 = true;
so.data.prop3 = [10, 20, 30, 40, 50];
so.data.prop4 = {x:100, y:200};
```

В объекте data можно сохранять данные основных типов: Number, String, Boolean и Object. Локально не могут быть сохранены функции, а также клипы, кнопки и текстовые блоки. Следует помнить, что при сохранении данных их необходимо присваивать не свойству data, а созданному в нем свойству (как это сделано выше). Так, следующее предложение будет проигнорировано интерпретатором:

```
so.data = value;
```

Само по себе присваивание значения свойству объекта data не приводит к автоматическому сохранению информации на диске пользователя. В результате этой операции данные только помещаются в текущий объект SharedObject. Сохранение же этого объекта в SOL-файл выполняется в следующих ситуациях:

- при закрытии окна проигрывателя;
- в результате вызова метода flush();
- в результате удаления текущего объекта SharedObject.

Для извлечения данных из локально сохраненного объекта SharedObject достаточно просто считать значение соответствующего свойства объекта data. Так, если в соответствующем каталоге у пользователя имеется файл so.sol, то следующий код выведет в панель **Output** перечень свойств и значений, хранящихся в этом объекте:

```
for(var i in so.data){
       trace(i + ": " + so.data[i]);
}
```

Сценарий, представленный в листинге 21.1, при каждом запуске фильма выводит число, показывающее, сколько раз этот документ открывался на компьютере пользователя. Здесь в объекте data создается свойство openCounter, в котором сохраняется число, увеличивающееся на единицу при каждом открытии документа. Сохранение локального объекта на диске происходит автоматически при закрытии окна проигрывателя (при условии, что пользователь позволяет локально сохранять информацию и для ее сохранения имеется достаточно свободного места).

Листинг 21.1. Локальный счетчик посещений

```
var so:SharedObject = SharedObject.getLocal("soObj");
if (so.data.openCounter == undefined) {
       // Если значение свойства не определено - первое открытие
```

```
        so.data.openCounter = 1;
}
else {
        // Если свойство уже имеет значение, увеличиваем его на единицу
        so.data.openCounter++;
}
this.createTextField("counterTxt", 1, 0, 0, 0, 0);
counterTxt.autoSize = true;
counterTxt.text = so.data.openCounter;
```

Объект `SharedObject` допускает возможность создания свойств, в которых могут храниться определенные значения. Однако свойства, добавленные к объекту `SharedObject`, будут доступны экземпляру локального объекта только на протяжении работы фильма. Они не сохраняются на диске пользователя и удаляются после окончания работы фильма. Создание свойств для объекта `SharedObject` имеет смысл только тогда, когда в нем требуется сохранить какую-то информацию, которая не должна быть записана на диск пользователя.

```
var so:SharedObject = SharedObject.getLocal("soObj");
so.prop1 = "value"; // Свойство, которое не будет записано на диск
so.data.prop1 = "value"; // Свойство, которое будет сохранено на диске
```

Сохранение объекта *SharedObject*

Как уже было сказано, текущий локальный объект автоматически сохраняется в SOL-файл при закрытии окна Flash Player. Сохранения не происходит только в случае, если этому препятствуют параметры настроек проигрывателя, заданные пользователем.

Удаление текущего локального объекта также автоматически приводит к сохранению его на диске. Удаление объекта `SharedObject` может быть выполнено с помощью оператора `delete` или в результате удаления из проигрывателя документа или клипа, содержащего переменные со ссылками на этот объект.

Основной проблемой, связанной с автоматическим сохранением, является то, что объем, выделенный для локального хранения информации на диске пользователя, может оказаться меньше объема, необходимого для сохранения текущего локального объекта. В этом случае автоматическое сохранение выполняться не будет, и при этом Flash Player не будет выводить панель с предупреждающим сообщением (см. рис. 21.2).

Для выполнения записи текущего локального объекта на диск можно воспользоваться методом `flush()`. Как следует из его названия, при вызове данный метод выполняет сохранение объекта в SOL-файл. Метод `flush()` имеет следующий синтаксис:

локальный объект`.flush(`*объем*`)`

Здесь *локальный объект* — это объект `SharedObject`, который сохраняется на диск.

Параметр *объем* — это число, указывающее, какое количество байтов необходимо зарезервировать на диске пользователя для сохранения текущего локального объекта. Как уже говорилось, по умолчанию для локального сохранения информации файлами из определенного домена отводится 100 Кбайт дискового пространства. Если предоставленного пространства оказывается недостаточно, Flash Player выводит панель с соответствующим сообщением (рис. 21.2) и предложением увеличить разрешенный лимит. Для того чтобы избежать многократного вывода этой панели, можно вызвать метод `flush()`, передав ему в качестве параметра объем, который потенциально может потребоваться для хранения информации. Если указанное значение превышает установленный пользователем лимит, то Flash Player выведет панель с сообщением. Если пользователь разрешит увеличить предоставленный объем, то в дальнейшем сообщение о необходимости увеличения объема выводиться не будет до тех пор, пока размер файла локального объекта вновь не превысит значения, указанного параметром *объем*.

Метод `flush()` возвращает булево значение или строку `"pending"` в зависимости от успешности выполнения записи локального объекта.

- Значение `true` возвращается, если пользователь не запретил локальное сохранение информации файлам из данного домена и при наличии достаточного для сохранения разрешенного объема дискового пространства. Если метод `flush()` был вызван с параметром *объем*, то значение `true` возвращается только в том случае, если разрешенный объем дискового пространства составляет значение не меньшее, чем значение параметра *объем*.
- Значение `false` возвращается, если пользователь запретил сохранение информации файлам из данного домена (установив флажок **Never Ask Again** в панели настроек **Local Storage**). Значение `false` также возвращается, если Flash не может осуществить запись локального объекта на диск (например, если диск недоступен или на нем нет места).
- Значение `"pending"` возвращается, если разрешенный для сохранения локальных объектов объем оказывается меньше, чем требуется для сохранения текущего локального объекта (или меньше, чем требует параметр *объем* метода `flush()`). Получение данного значения всегда сопровождается выводом панели с предложением разрешить увеличение объема. Если пользователь соглашается предоставить требуемое дисковое пространство, метод `flush()` вызывается повторно.

Если метод `flush()` возвращает значение `"pending"`, он уже не позволяет определить, разрешил пользователь увеличить объем или отказался от увеличения. Между тем узнать результат выбора пользователя бывает необходимо, например, для того, чтобы известить его о невозможности выполнения тех или иных операций при запрете увеличения объема.

Для того чтобы узнать результат выбора пользователя, нужно воспользоваться обработчиком события `onStatus` объекта `SharedObject`. Событие `onStatus` генерируется при выборе пользователем одной из двух опций **Allow** (Разрешить) или **Deny** (Запретить) в панели, представленной на рис. 21.2. Определение метода обработчика

события onStatus синтаксически аналогично определению одноименного метода для объекта LocalConnection и имеет следующий вид:

```
локальный объект.onStatus = function(информационный объект){
блок кода;
...
};
```

При вызове метода обработчика ему передается параметр *информационный объект*, содержащий два свойства code и level, строковые значения которых сообщают о результате вызова метода flush():

- если возвращаемым значением метода flush() является строка "pending", и пользователь не разрешил увеличить дисковое пространство (кнопка **Deny** в панели, выводимой Flash Player), свойство code будет иметь значение "SharedObject.Flush.Failed", а свойство level — значение "error";
- если возвращаемым значением метода flush() является строка "pending", и пользователь разрешил увеличить дисковое пространство (кнопка **Allow** в панели, выводимой Flash Player), свойство code будет иметь значение "SharedObject.Flush.Success", а свойство level — значение "status".

Ниже приводится фрагмент кода, иллюстрирующий проверку результата вызова метода flush(). В зависимости от того, удалось ли выполнить сохранение данных в текстовое поле, выводится соответствующее сообщение.

```
var so:SharedObject = SharedObject.getLocal("so");
this.createTextField("statusTxt", 1, 0, 0, 0, 0);
statusTxt.autoSize = true;
// Если flush() возвращает true или "pending"
if (so.flush(1000000)) {
        so.onStatus = function(iObj) {
        // При выводе панели с предложением об увеличении объема...
        if (iObj.level == "error") {
        statusTxt.text = "Недостаточно выделенного места на диске!";
        }
        if (iObj.level == "status") {
        statusTxt.text = "Сохранение данных успешно выполнено.";
        }
        };
}
// Если flush() возвращает false
else {
        statusTxt.text = "Сохранение невозможно!";
}
```

Примечание

Наряду с применением обработчика onStatus класса SharedObject, можно воспользоваться универсальным обработчиком onStatus класса System. С помощью этого обработчика можно контролировать различные аспекты состояния трех различных объектов: LocalConnection, NetStream и SharedObject. Если для одного из этих объектов не определен собственный метод onStatus(), при наличии метода обработчика onStatus класса System в соответствующей ситуации будет вызываться он. Методу System.onStatus() в качестве параметра также передается объект *информационный объект* с двумя свойствами: code и level. Применение обработчика класса System может оказаться полезным как универсальное средство обработки ошибок, связанных с функционированием указанных объектов. Свойство code позволяет идентифицировать объект, вызвавший наступление события onStatus.

Определение объема локального объекта

Для того чтобы определить объем в байтах, занимаемый текущим объектом SharedObject, можно воспользоваться методом getSize(), имеющим следующий синтаксис:

локальный объект.getSize()

Метод getSize() возвращает объем текущего объекта *локальный объект*, а не объем SOL-файла, записанного на диск. Так, например, если пользователь запретил локальное хранение информации, то данный метод будет возвращать объем локального объекта, созданного в фильме, однако этот объект не будет сохранен на диске.

Метод вычисляет объем локального объекта, последовательно перебирая все свойства, хранящиеся в его объекте data. Если число таких свойств велико, метод getSize() может выполняться довольно долго. В связи с этим применять этот метод стоит только тогда, когда в этом действительно есть необходимость.

Удаление локальных данных

Поскольку объем, предоставляемый для локального хранения информации, как правило, всегда ограничен, необходимо тщательно следить за неиспользуемыми свойствами объекта SharedObject и удалять их в случае ненадобности. Также следует удалять ставшие ненужными объекты SharedObject.

Для того чтобы удалить свойство объекта data, необходимо использовать оператор delete. При этом необходимо помнить, что соответствующее свойство будет удалено из SOL-файла только после его повторного сохранения. Проиллюстрируем это на примере:

```
//Создаем локальный объект с тремя свойствами
var so:SharedObject = SharedObject.getLocal("so");
so.data.prop1 = "A";
so.data.prop2 = "B";
```

```
so.data.prop3 = "C";
//Сохраняем созданный локальный объект в файле so.sol
so.flush();
```

После этого в том же самом файле заменим предыдущий код на следующий:

```
var so:SharedObject = SharedObject.getLocal("so");
delete so.data.prop1;
//Вновь сохраняем локальный объект в файле so.sol
so.flush();
```

Если теперь вывести список свойств сохраненного локального объекта, можно убедиться в том, что свойства `prop1` среди них уже нет (это можно также проверить, непосредственно открыв файл so.sol в любом текстовом редакторе).

```
for(var i in so.data){
trace(i);
}
```

В панель **Output** будет выведено: `prop3`, `prop2`.

Если подобным образом удалить все свойства локального объекта и любым способом выполнить его сохранение, соответствующий SOL-файл будет удален с диска.

Для того чтобы полностью удалить локальный объект с диска пользователя, наряду с описанным способом можно применить специальный метод `clear()`. Метод `clear()` вызывается объектом `SharedObject` и удаляет все свойства его объекта `data`. В результате его применения текущий объект `SharedObject` остается пустым (хотя у него по-прежнему имеется свойство `data`), а соответствующий ему SOL-файл удаляется с диска.

По всей видимости, в результате допущенной разработчиками ошибки вызов метода `clear()` может вызывать ошибку при компиляции. Для того чтобы избежать этой проблемы, можно вызывать этот метод не с помощью идентификатора, а используя оператор `[]`. Эта неполадка была исправлена в вышедшем позднее обновлении Flash v7.2.

```
so.clear(); // Может вызвать ошибку
so.["clear"](); // Работает
```

Пример применения объекта *SharedObject* для хранения пользовательских настроек интерфейса

В качестве примера, демонстрирующего возможности практического применения объекта `SharedObject`, рассмотрим создание формы, позволяющей пользователю задать параметры интерфейса, которые будут локально сохранены на его компьютере, и при очередной загрузке формы в ней будут отображаться настройки, заданные пользователем перед ее последним закрытием. Подобная задача может возникнуть

при разработке приложений, с которыми пользователь регулярно работает в течение продолжительного времени, например, обучающих или развлекательных программ. Пользователь может один раз задать удобные для него параметры интерфейса (например, цвет фона, размер текста и т. д.), чтобы в дальнейшем, обращаясь к приложению, не тратить время на повторную настройку интерфейса.

Для того чтобы не делать программный код чересчур громоздким и не уводить пример от его основного назначения, мы ограничимся лишь учебной моделью интерфейса, состоящего из следующих элементов:

- цветной фон;
- заголовки;
- массив текста;
- динамически загружаемое звуковое сопровождение;
- динамически загружаемая растровая графика.

Пользователь должен иметь возможность выбрать одну из предложенных ему цветовых схем, а также указать, нужно ли автоматически выполнять динамическую загрузку растровых изображений и звуков. Цветовая схема определяет сочетание цветов фона, текстовых заголовков, основного текста, а также фона основного текста. Кроме того, пользователь должен иметь возможность удалить со своего локального диска информацию о настройках. Внешний вид формы представлен на рис. 21.4.



Рис. 21.4. Форма для установки пользовательских настроек элементов интерфейса

Включение или отключение автоматической загрузки растровой графики и звука выполняется с помощью флажков. Установка флажка приводит к загрузке соответствующего элемента, снятие выгружает его. При очередном открытии документа загрузка элемента выполняется только в том случае, если соответствующей флаг был установлен перед последним закрытием.

Создание флажка — это довольно простая задача, однако поскольку на страницах этой книги флаги еще не применялись, скажем об этом несколько слов. Флаг представляет собой клип, монтажная линейка которого состоит из двух кадров. Первый кадр содержит изображение снятого флага (checked out) и снабжен меткой "out". Второй кадр содержит изображение установленного (checked in) флага и снабжен меткой "in". Клипам-флагам мы присвоили идентификаторы bitmapFlag и soundFlag. В соответствии с нашими целями флаги должны использоваться двояко. С одной стороны, при нажатии на флаг он должен устанавливаться или сбрасываться и выполнять соответствующее действие (загрузка или выгрузка внешнего файла). С другой стороны, если пользователь установил опцию, то при следующем открытии документа флаг должен автоматически установиться и выполнить возложенную на него функцию. Для того чтобы сделать программную модель флага более универсальной, мы отделим механизм его собственного функционирования (снятие и установка галочки при нажатии) от выполняемых им задач (загрузка и выгрузка того или иного объекта). Таким образом, если затем нам понадобится добавить в интерфейс еще несколько флагов, это можно будет сделать, не вмешиваясь в программный код уже имеющихся. Функционирование флага обеспечивается с помощью функции operateFlag(), которая принимает три параметра:

- flag — ссылка на экземпляр клипа, представляющего флаг;
- checkIn — функция, вызываемая при установке флага;
- checkOut — функция, вызываемая при снятии флага.

Естественно, в качестве параметров передаются только ссылки на функции (то есть их имена без оператора ()).

Код функции operateFlag() приведен в листинге 21.2 и использует стандартный алгоритм кнопки-флага (листинг 7.1).

Для принудительной установки и снятия флагов используются функции checkInFlag() и checkOutFlag() соответственно. В качестве параметров эти функции принимают ссылку на экземпляр клипа, представляющего флаг, и ссылку на функцию, которая должна быть выполнена.

Выбор и установка цветовой схемы осуществляются следующим образом. Каждой предложенной цветовой схеме сопоставляется уникальный номер от 1 до 5 включительно. Применение цветовой схемы осуществляет функция setScheme(). Эта функция принимает в качестве параметра числовой идентификатор схемы и задает цвета элементам формы, расположенным на сцене, выбирая их из созданных заранее массивов.

Кнопки, по нажатию на которые выполняется установка цветовой схемы, имеют идентификаторы, состоящие из слова scheme и индекса, соответствующего номеру выбранной схемы. Это позволяет легко в цикле обработать кнопки, снабдив их единообразным поведением.

Программный код, реализующий описанную функциональность, представлен в листинге 21.2.

Листинг 21.2. Пример хранения пользовательских настроек интерфейса

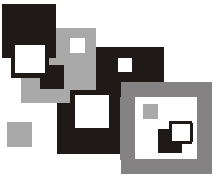
```
//*ИНИЦИАЛИЗАЦИЯ*
// Объект MovieClipLoader, выполняющий загрузку растрового изображения
var mcl:MovieClipLoader = new MovieClipLoader();
// Объект Sound, выполняющий динамическую загрузку звука
var s:Sound = new Sound(this);
s.onLoad = function() {
       s.start();
};
// Объект Color, отвечающий за цвет фона
var col:Color = new Color(main.bg);
/* Элементы цветовых схем (номер элемента массива это номер цветовой
схемы минус единица)*/
// цвета фона
var bgCol = [0xFFFFFF, 0xFFCC66, 0x6699CC, 0xCC0000, 0xAEFBBB];
// цвета заголовка
var headerCol = [0x000000, 0x993300, 0xFFFFFF, 0xFFCC66, 0x00AD22];
// Цвета фона текста
var textBlockCol = [0x666666, 0x8E4400, 0x333366, 0x990000, 0x5A8033];
// Цвета текста
var textBg = [0xFFFFFF, 0xFFEECC, 0xC8DAEC, 0xFFCC66, 0xFFFFCC];
// Функция, обеспечивающая функциональность флага при нажатии
function operateFlag(flag:Object, checkIn:Function, checkOut:Function) {
       flag.onPress = function() {
              if (!this.checkIn) {
                     checkIn();
                     this.nextFrame();
                     this.checkIn = true;
              }
              else {
                     checkOut();
                     this.prevFrame();
                     this.checkIn = false;
              }
       };
}
// Функция, принудительно устанавливающая флаг
function checkInFlag(flag:Object, checkIn:Function) {
```

```
        flag.gotoAndStop("in");
        flag.checkIn = true;
        checkIn();
}
// Функция, принудительно снимающая флаг
function checkOutFlag(flag:Object, checkOut:Function) {
        flag.gotoAndStop("out");
        flag.checkIn = false;
        checkIn();
}
// Функция, устанавливающая цветовую схему
function setScheme(num:Number):Void {// num - номер выбранной схемы
        curScheme._x = this["scheme" + num]._x;
        curScheme._y = this["scheme" + num]._y;
        num = num - 1;
        // Задаем цвет заголовка
        headerTxt.textColor = headerCol[num];
        // Задаем цвет текста
        blockTxt.textColor = textBlockCol[num];
        // Задаем цвет фона текста
        blockTxt.backgroundColor = textBg[num];
        // Задаем цвет фона
        col.setRGB(bgCol[num]);
}
/* Установка цветовой схемы при выборе опции и сохранение выбора
в локальном объекте */
for (i = 1; i < 6; i++) {
        this["scheme" + i].onPress = function() {
                so.data.scheme = Number(String(this).substr(-1));
                setScheme(Number(String(this).substr(-1)));
                so.flush();
        };
}
/* Функции, определяющие действия, выполняемые при включении/отключении
флагов соответствующих опций */
function bitmapOn() {
        mcl.loadClip("media/pic.jpg", holder);
        so.data.bitmap = true;
        so.flush();
}
```

```
function bitmapOff() {
	mcl.unloadClip(holder);
	so.data.bitmap = false;
	so.flush();
}
function soundOn() {
	if (!sloaded) {
		s.loadSound("media/Ebony_Samba.mp3", false);
		s.start();
		sloaded = true;
	}
	else {
		s.start();
	}
	so.data.sound = true;
	so.flush();
}
function soundOff() {
	s.stop();
	so.data.sound = false;
	so.flush();
}
// Активация флагов
operateFlag(bitmapFlag, bitmapOn, bitmapOff);
operateFlag(soundFlag, soundOn, soundOff);
// *ВОССТАНОВЛЕНИЕ ОПЦИЙ ИЗ ЛОКАЛЬНОГО ОБЪЕКТА ИЛИ УСТАНОВКА ЗНАЧЕНИЙ ПО
// УМОЛЧАНИЮ *
// Создание объекта SharedObject с именем colorScheme
var so:SharedObject = SharedObject.getLocal("colorScheme");
// Установка звуковых настроек
if (so.data.sound != undefined) {// Если установка этой опции
                                 // сохранена...
	if (so.data.sound) {// Если опция установлена в режим "включено"
		checkInFlag(soundFlag, soundOn());
	}
	else {// Если опция установлена в режим "выключено"
		checkOutFlag(soundFlag, soundOff());
	}
}
```

```
else {// Если установка этой опции не сохранена
        checkOutFlag(soundFlag, soundOff());
}
// Установка настроек растровой графики
if (so.data.bitmap != undefined) {
        if (so.data.bitmap) {
                checkInFlag(bitmapFlag, bitmapOn());
        }
        else {
                checkOutFlag(bitmapFlag, bitmapOff());
        }
}
else {
        checkOutFlag(bitmapFlag, bitmapOff());
}
// Установка цветовой схемы
if (so.data.scheme != undefined) {
        setScheme(so.data.scheme);
}
else {
        setScheme(1);
}
// Удаление локальных данных при нажатии на соответствующую кнопку
clearBut.onPress = function() {
        so["clear"]();
        checkOutFlag(soundFlag, soundOff());
        checkOutFlag(bitmapFlag, bitmapOff());
        setScheme(1);
};
```

Глава 22



Контроль параметров документа и проигрывателя. Информация о системе и оборудовании

Пришла нужда постучать по дереву — обнаруживаешь,
что мир состоит из алюминия и пластика.

Закон Флагга

Класс *Stage*. Контроль размеров и масштабирования окна фильма

Класс `Stage` располагает небольшим, но чрезвычайно полезным программным инструментарием, позволяющим получать информацию о размерах окна фильма и осуществлять контроль выравнивания и масштабирования фильма внутри окна проигрывателя Flash Player. Свойства и методы класса `Stage` являются статическими, то есть доступ к ним осуществляется непосредственно с помощью объекта `Stage` и не требует использования конструктора для создания экземпляра.

Управление масштабированием фильма

Воспроизведение Flash-фильма в автономном проекторе или на HTML-странице всегда выполняется внутри окна проигрывателя Flash Player. При этом пользователь может произвольно изменять размер окна проигрывателя, сжимая его или растягивая. Изменение размеров окна автономного проигрывателя выполняется непосредственно путем перемещения его границ с помощью курсора мыши. Изменение размеров окна проигрывателя, размещенного на HTML-странице, выполняется в одном из следующих случаев.

- Если размеры окна проигрывателя в атрибутах `width` и `height` тегов `<object>` и `<embed>` заданы в относительных единицах (процентах), то они будут изменяться при изменении размеров окна браузера.
- При изменении значений атрибутов `width` и `height` тегов `<object>` и `<embed>`. Изменения размеров окна проигрывателя вступают в силу только после обновления страницы (Refresh).

Изменение размеров окна проигрывателя может по-разному отразиться на содержимом фильма. Результат такого изменения определяется настройками масштабирования. При размещении Flash-фильма на HTML-странице настройки масштабирования задаются параметром (атрибутом) `scale` тега `<object>` (`<embed>`). Соответствующие настройки можно выполнить при публикации или вручную задать требуемое значение данному параметру в HTML-коде. При работе с автономным проектором режим масштабирования может быть установлен только программным образом.

Для установки режима масштабирования используется свойство `scaleMode`, обращение к которому выполняется с применением следующего синтаксиса:

```
Stage.scaleMode
```

В качестве значения данное свойство может принимать одно из следующих строковых значений: `"showAll"`, `"noBorder"`, `"exactFit"` и `"noScale"`. Присваивание данному свойству любого другого значения даст результат, аналогичный использованию значения `"showAll"`. Приведем описание перечисленных режимов масштабирования фильма внутри окна проигрывателя (рис. 22.1).

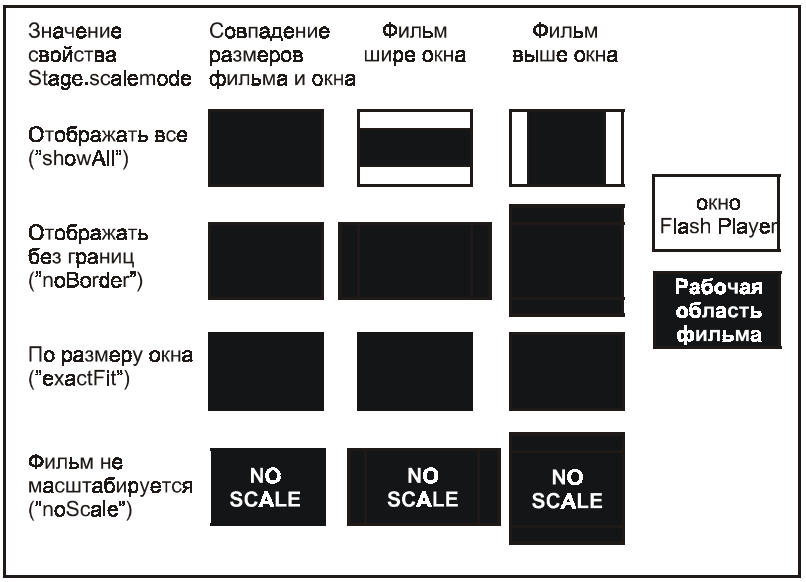


Рис. 22.1. Масштабирование фильма в окне проигрывателя средствами AS

- `showAll` — применение данного режима масштабирует фильм так, что рабочая область целиком размещается внутри окна проигрывателя. Если размер фильма меньше размера окна проигрывателя, то фильм пропорционально увеличивается. Если фильм больше окна проигрывателя, он пропорционально уменьшается до тех пор, пока рабочая область не будет полностью видна в окне проигрывателя. Если фильм и окно проигрывателя имеют разные пропорции (соотношение ши-

рины и высоты), то в результате применения данного режима в окне проигрывателя будет видна часть содержимого вспомогательной области. Режим `showAll` по умолчанию используется при воспроизведении фильма в автономном проекторе и на HTML-странице.

- `noBorder` — применение данного режима пропорционально масштабирует фильм так, что окно проигрывателя оказывается полностью заполнено содержимым рабочей области фильма. Если рабочая область фильма и окно проигрывателя имеют разные пропорции, часть содержимого рабочей области будет срезана границами окна проигрывателя. Таким образом, при использовании данного режима заполнение окна проигрывателя может выполняться за счет срезания части фильма.
- `exactFit` — применение данного режима масштабирует фильм так, что окно проигрывателя оказывается полностью заполнено содержимым рабочей области фильма. В отличие от режима `noBorder`, данный режим точно подгоняет размеры рабочей области фильма под размеры окна проигрывателя. Если рабочая область фильма и окно проигрывателя имеют разные пропорции, фильм масштабируется непропорционально. Таким образом, при использовании данного режима заполнение окна проигрывателя может выполняться за счет изменения пропорций содержимого фильма.
- `noScale` — данный режим отключает масштабирование фильма. При его использовании фильм отображается внутри окна проигрывателя со своими исходными размерами. Если фильм меньше окна, внутри окна проигрывателя будет видна часть вспомогательной области. Если фильм больше окна, в окне будет отображаться только часть содержимого рабочей области. Режим `noScale` по умолчанию используется при воспроизведении фильма в среде тестирования (однако не является режимом по умолчанию для автономного проектора и браузера). Для того чтобы задействовать данный режим при воспроизведении фильма на HTML-странице, достаточно выполнить одно из следующих действий.
  - В HTML-коде страницы присвоить параметру (атрибуту) `scale` тега `<object>` (`<embed>`) значение `"noscale"`. При этом применение свойства `scaleMode` уже не является обязательным.
  - В Flash-документе присвоить свойству `scaleMode` значение `"noScale"` и в HTML-коде страницы присвоить параметру (атрибуту) `scale` тега `<object>` (`<embed>`) значение `"showall"` (или вообще удалить данный параметр, так как значение `"showall"` используется по умолчанию).

Отметим, что при использовании режима `"noScale"` изменение размера окна проигрывателя приводит к наступлению события `onResize` объекта `Stage`.

Важным преимуществом свойства `scaleMode` является возможность динамического изменения режима масштабирования в нужный момент. Как только данному свойству присваивается новое значение, сразу же устанавливается соответствующий режим масштабирования. В листинге 22.1 приведен сценарий, устанавливающий режим масштабирования при нажатии на соответствующую кнопку. Название текущего режима выводится в динамическое текстовое поле с именем `txt`. Читатель может поработать с этим примером для того, чтобы на практике разобраться в различиях режимов масштабирования.

Листинг 22.1. Установка режима масштабирования фильма

```
txt.text = Stage.scaleMode;
showAllBut.onPress = function() {
       Stage.scaleMode = "showAll";
       txt.text = Stage.scaleMode;
};
exactFitBut.onPress = function() {
       Stage.scaleMode = "exactFit";
       txt.text = Stage.scaleMode;
};
noBorderBut.onPress = function() {
       Stage.scaleMode = "noBorder";
       txt.text = Stage.scaleMode;
};
noScaleBut.onPress = function() {
       Stage.scaleMode = "noScale";
       txt.text = Stage.scaleMode;
};
```

Необходимо иметь в виду, что при размещении Flash-фильма на HTML-странице изначально режим масштабирования устанавливается на основании значения параметра (атрибута) `scale` тегов `<object>` и `<embed>` вне зависимости от того, какое значение присвоено свойству `scaleMode`. Для того чтобы изменить режим масштабирования, прописанный в HTML-коде, нужно присвоить требуемое значение свойству `scaleMode` непосредственно в процессе воспроизведения фильма. Это можно сделать, либо нажав на соответствующую кнопку, либо организовав цикл (например, с применением обработчика `onEnterFrame`), выполняющий присваивание требуемого значения свойству `scaleMode` до тех пор, пока соответствующий режим масштабирования не будет установлен *(такой пример будет показан далее в разд. "Контроль размеров сцены и окна проигрывателя").*

Управление выравниванием фильма

Если размер рабочей области фильма и размер окна проигрывателя не совпадают, возникает вопрос, каким образом фильм будет позиционироваться в окне? Ответом на этот вопрос является свойство `align`, позволяющее задавать параметры выравнивания фильма относительно границ окна проигрывателя Flash Player. При размещении фильма на HTML-странице выравнивание может быть задано параметром (атрибутом) тега `<object>` (`<embed>`), который для обоих тегов носит одинаковое название `salign`. Тем не менее свойство `align` имеет два существенных преимущества. Во-первых, с его помощью можно управлять выравниванием фильма в авто-

номном проекторе. Во-вторых, его значение можно динамически изменять по ходу воспроизведения фильма, точно так же, как и значение свойства `scaleMode`.

Свойство `align` может принимать одно из фиксированных строковых значений, приведенных в табл. 22.1.

Таблица 22.1. *Значения свойства* `align`

Значение	Вертикальное выравнивание	Горизонтальное выравнивание
`"T"`	по верхней границе	по центру
`"B"`	по нижней границе	по центру
`"L"`	по центру	по левой границе
`"R"`	по центру	по правой границе
`"TL"`	по верхней границе	по левой границе
`"TR"`	по верхней границе	по правой границе
`"BL"`	по нижней границе	по левой границе
`"BR"`	по нижней границе	по правой границе

Для того чтобы отцентрировать фильм относительно границ окна проигрывателя, нужно в качестве значения присвоить свойству `align` любую строку, отличную от приведенных в табл. 22.1.

Результат применения свойства `align` схематично представлен на рис. 22.2.

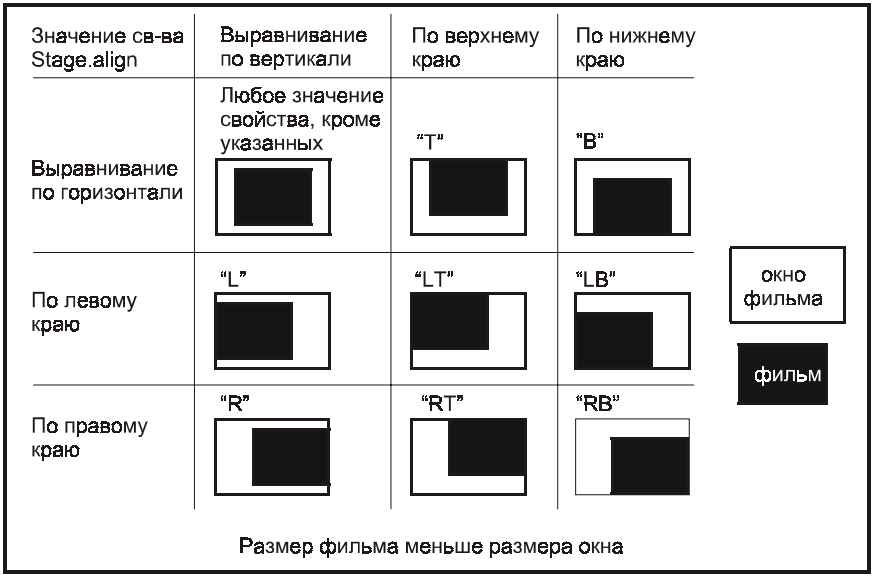


Рис. 22.2. Выравнивание фильма в окне проигрывателя средствами AS

В качестве иллюстрации возможностей свойства `align` приведем простой пример управления выравниванием фильма внутри окна проигрывателя. Фильм состоит из девяти кнопок, соответствующих разным вариантам выравнивания. Имена кнопок имеют форматы `but_TL`, `but_T`, `but_TR` и т. д. Приведенный в листинге 22.2 сценарий при отпускании кнопки устанавливает соответствующий режим выравнивания фильма. При установке требуемого режима строка, задающая значение свойству `align`, вырезается из имени кнопки с помощью метода `split()`. Кнопка, выполняющая центрирование фильма, имеет имя `but_C`. Для простоты обход кнопок выполняется с применением цикла `for...in`.

Листинг 22.2. Установка режима выравнивания фильма

```
Stage.scaleMode = "noScale";
for (i in this) {
       if (this[i] instanceof Button) {
              this[i].onRelease = function() {
                     Stage.align = this._name.split("_")[1];
              };
       }
}
```

Контроль размеров сцены и окна проигрывателя

Объект `Stage` обладает статическими свойствами `width` и `height`, позволяющими получить информацию о размерах рабочей области и размерах окна проигрывателя. Данные свойства доступны только для чтения. Это означает, что Flash не позволяет динамически управлять размером рабочей области и размером окна проигрывателя. Значения, возвращаемые свойствами `width` и `height`, зависят от текущего режима масштабирования, который задается либо с помощью свойства `scaleMode`, либо с помощью параметра (атрибута) `scale` тега `<object>` (`<embed>`) при размещении фильма на HTML-странице:

- если фильм использует режим масштабирования `noScale`, то свойства `width` и `height` соответственно возвращают ширину и высоту окна проигрывателя;
- если используется любой другой режим масштабирования, то свойства `width` и `height` возвращают исходные размеры рабочей области, заданные непосредственно в рабочей среде.

Следующий код выводит в панель **Output** размеры рабочей области документа:

```
trace("Ширина: " + Stage.width);
trace("Высота: " + Stage.height);
```

Изменение размеров окна проигрывателя приводит к изменению размеров фильма только при использовании режима `"noScale"`. Если применяется другой режим масштабирования, то изменения фактических размеров не происходит, фильм просто растягивается или сжимается вместе со всем своим содержимым.

Следует отметить, что при установке свойства scaleMode в значение "noScale" для фильма, размещенного на HTML-странице, непосредственно после загрузки свойства width и height принимают нулевые значения. Для того чтобы получить реальные значения этих свойств, необходимо обратиться к ним уже после загрузки фильма (то есть после того как фильм отобразится на странице). Удобнее всего для этих целей воспользоваться обработчиком события onResize.

Событие onResize генерируется объектом Stage в результате изменения размеров окна проигрывателя при условии, что текущим режимом масштабирования является режим "noScale". Данное событие оказывается очень полезным в ситуациях, требующих выполнения каких-либо действий в ответ на изменение размеров окна проигрывателя (например, смещение координат элементов интерфейса так, чтобы они оставались видимыми и не срезались границами окна).

Для обработки события onResize необходимо создать объект-приемник, зарегистрировать его на получение сообщений от объекта Stage и определить для него соответствующий метод обработчика события. Для регистрации объекта-приемника и ее отмены в классе Stage имеются два стандартных метода: addListener() и removeListener().

Приведенный ниже код выводит текущие размеры окна проигрывателя в динамические текстовые поля, расположенные в центре рабочей области. Для тестирования работы этого кода в браузере необходимо при публикации установить режим масштабирования **No scale** или **Default (Show all)**.

```
var resizeObj = new Object();
// Определяем координаты центра рабочей области
var centerX = Stage.width/2;
var centerY = Stage.height/2;
this.createTextField("widthTxt", 1, centerX, centerY, 0, 0);
this.createTextField("heightTxt", 2, centerX, centerY+20, 0, 0);
widthTxt.autoSize = heightTxt.autoSize = true;
Stage.scaleMode = "noScale"
Stage.addListener(resizeObj);
resizeObj.onResize = function(){
// Выводим новые размеры окна проигрывателя
widthTxt.text = Stage.width;
heightTxt.text = Stage.height;
};
```

При анализе приведенного кода необходимо обратить внимание на то, что установка режима масштабирования "noScale" выполняется после определения координат центра рабочей области с помощью свойств width и height. Это сделано из-за того, что при присваивании свойству scaleMode значения "noScale" в случае размещения фильма на HTML-странице в начальный момент значения свойств width и height оказываются равными нулю. Если бы присваивание значения "noScale" свойству scaleMode было выполнено в самом начале, то координаты центра фильма

были бы определены неверно. Если же свойство scaleMode принимает значение, отличное от "noScale", то в начальный момент свойства width и height будут иметь значения, соответствующие ширине и высоте рабочей области. Таким образом, в приведенном примере мы сначала нашли координаты точки центра рабочей области, а после этого, установив режим "noScale", определили метод обработчика onResize, с помощью которого осуществляется изменение значений в текстовых полях при изменении размеров окна проигрывателя. Описанная особенность имеет место только при работе с фильмом, размещенным на HTML-странице. В автономном проекторе подобной проблемы не существует.

Следует иметь в виду, что после загрузки фильма на HTML-страницу событие onResize генерируется автоматически. Таким образом, определив соответствующий метод обработчика (при условии использования режима "noScale"), можно сразу же при загрузке фильма получить информацию о размерах окна проигрывателя. При этом важно помнить, что размеры рабочей области фильма могут отличаться от размеров окна проигрывателя, и, что если в HTML-коде установлен режим масштабирования, отличный от "noscale" или "showall", метод обработчика onResize вызываться не будет.

Как уже было показано ранее, режим масштабирования может быть произвольно изменен в любой момент. Таким образом, вне зависимости от настроек масштабирования, заданных в HTML-коде, мы можем при желании установить любой требуемый режим. В листинге 22.1 демонстрировалось изменение режима масштабирования с помощью нажатия на соответствующую кнопку. Для того чтобы программным образом переопределить настройки масштабирования, заданные в коде HTML, можно воспользоваться следующим подходом. Определим метод обработчика onEnterFrame и в теле этого метода будем периодически присваивать свойству scaleMode значение "noScale" до тех пор, пока данное свойство не примет это значение. Как только данный режим масштабирования будет установлен, метод обработчика можно удалить. Этот подход делает масштабирование Flash-фильма независимым от параметров масштабирования, заданных с помощью HTML, поскольку масштабирование выполняется уже в процессе воспроизведения фильма. С другой стороны применение данного режима выполняется очень быстро, так что пользователь, скорее всего, не заметит переключения режима.

```
this.onEnterFrame = function() {
        if (Stage.scaleMode != "noScale") {
                Stage.scaleMode = "noScale";
        }
        else {
        delete this.onEnterFrame;
        }
};
var resizeObj:Object = new Object();
Stage.addListener(resizeObj);
resizeObj.onResize = function(){
```

```
widthTxt.text = Stage.width;
heightTxt.text = Stage.height;
};
```

В листинге 22.3 приведен пример сценария, который при изменении размеров окна проигрывателя смещает объекты, расположенные на сцене так, чтобы расстояние от каждого из них до центра рабочей области изменялось пропорционально изменению размеров окна. Это позволяет отображать данные объекты, не изменяя их размеры, при любых соотношениях ширины и высоты окна проигрывателя (рис. 22.3).

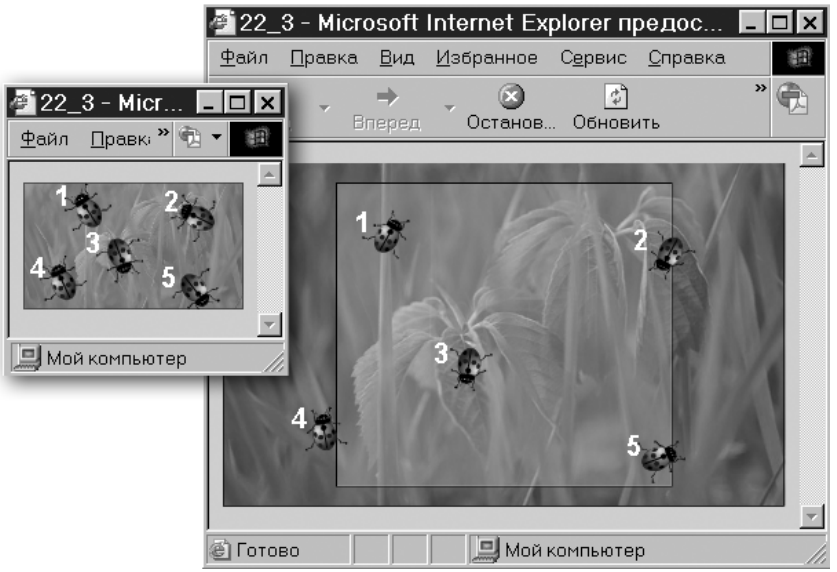


Рис. 22.3. Управление размещением объектом при масштабировании окна проигрывателя

Позиционируемые объекты представлены клипами с именами `mc1`, `mc2`, ..., `mc5`. Фоновое растровое изображение с ограничивающей рамкой — это клип с именем `frame`. Размер этого клипа меняется пропорционально изменению размеров окна проигрывателя (в то время, как все остальные элементы в фильме не меняют свой масштаб вне зависимости от режима масштабирования, заданного в HTML-коде).

Листинг 22.3. Управление размещением объекта при масштабировании окна проигрывателя

```
// Определяем координаты центра рабочей области
centerX = Stage.width / 2;
centerY = Stage.height / 2;
```

```
// Определяем исходный размер рабочей области
initW = Stage.width;
initH = Stage.height;
var resizeObj:Object = new Object();
// Устанавливаем режим "noScale"
this.onEnterFrame = function() {
        if (Stage.scaleMode != "noScale") {
                Stage.scaleMode = "noScale";
        }
        else {
        delete this.onEnterFrame;
        }
};
// Запоминаем начальные координаты клипов и расстояние от центра до клипа
for (i = 1; i < 6; i++) {
        this["mc" + i].initX = this["mc" + i]._x;
        this["mc" + i].initY = this["mc" + i]._y;
        this["mc" + i].distFromCenterX = centerX - this["mc" + i].initX;
        this["mc" + i].distFromCenterY = centerY - this["mc" + i].initY;
}
var resizeObj:Object = new Object();
Stage.addListener(resizeObj);
// При изменении размеров окна проигрывателя...
resizeObj.onResize = function() {
        // Вычисляем коэффициенты масштабирования окна по X и по Y
        var xRatio = Stage.width / initW;
        var yRatio = Stage.height / initH;
        for (i = 1; i < 6; i++) {
        // Позиционируем клипы
        _root["mc" + i]._x = centerX - _root["mc" + i].distFromCenterX *↵
        xRatio;
        _root["mc" + i]._y = centerY - _root["mc" + i].distFromCenterY *↵
        yRatio;
        }
        // Масштабируем рамку пропорционально масштабированию окна
        frame._xscale = 100 * xRatio;
        frame._yscale = 100 * yRatio;
};
```

Управление отображением меню проигрывателя

Читателю наверняка хорошо знакомо контекстное меню проигрывателя, которое появляется при щелчке правой кнопкой мыши внутри окна проигрывателя. Данное меню, в частности, содержит команды, позволяющие останавливать и запускать воспроизведение основной монтажной линейки документа, изменять масштаб просмотра и устанавливать степень сглаживания графики. В некоторых ситуациях Flash-дизайнер не хотел бы давать пользователю возможность применения этих команд, поскольку это может нарушить целостность восприятия и раскрыть некоторые особенности реализации, которые автор не хотел бы афишировать. Так, например, увеличив ролик, можно "раскрыть секреты" некоторых технических уловок. Далее в этой главе мы детально рассмотрим возможности, позволяющие управлять содержимым контекстного меню, а пока что опишем последнее неисследованное свойство объекта `Stage`, связанное именно с меню проигрывателя. Это свойство носит название `showMenu`, и с его помощью можно отключить все команды контекстного меню за исключением команды настроек **Settings...** и команды **About Macromedia Flash Player...**, представляющей ссылку на раздел компании Macromedia, посвященный проигрывателю Flash Player.

Свойство `showMenu` принимает булево значение, которое можно задать в любой момент. Если данное свойство принимает значение `true`, то контекстное меню будет содержать полный набор команд. Если свойству `showMenu` присвоить значение `false`, в контекстном меню останутся только две указанные выше команды. Если фильм воспроизводится в автономном проигрывателе (проектор), то данное свойство будет влиять как на контекстное меню, выводимое при нажатии правой кнопки мыши, так и на главное меню самого проигрывателя, содержащее команды **File**, **View**, **Control**, **Help**.

Следует отметить, что свойство `showMenu` не отключает вывод контекстного меню выделяемого и/или редактируемого текстового блока.

Следующий код выполняет попеременное включение и отключение контекстного меню (и главного меню автономного проигрывателя) при каждом нажатии на кнопку `menuBut`.

```
menuBut.onPress = function(){
if(!menuOff){
        Stage.showMenu = false;
        menuOff = true;
}
else{
        Stage.showMenu = true;
        menuOff = false;
        }
};
```

Управление автономным проигрывателем. Функция *fscommand()*

При создании автономных Flash-приложений (с расширением exe), называемых *проекторами* (projectors), довольно часто возникает необходимость управления определенными параметрами проигрывателя. Для реализации этих возможностей в ActionScript существует глобальная функция `fscommand()`. Данная функция может принимать от одного до двух строковых параметров и имеет следующий формат:

```
fscommand("command", "parameters")
```

Здесь *command* — это строка, указывающая название выполняемой команды, а `parameters` — строка, передаваемая проигрывателю Flash Player или содержащему его приложению (Web-браузер, Macromedia Director и др.) в качестве параметра. В табл. 22.2 приведен полный перечень принимаемых данной функцией параметров и описано оказываемое ими действие. Отметим, что все приведенные в таблице параметры дают результат только при использовании проектора или при воспроизведении Flash-фильма в автономном проигрывателе.

Таблица 22.2. *Команды и параметры функции* `fscommand()`

Команда	Параметр	Назначение
`*`	`"true"` или `"false"`	Значение `true` устанавливает для Flash Player полноэкранный режим, значение `false` возвращает обычный режим
`"allowscale"`	`"true"` или `"false"`	Значение `false` исключает возможность масштабирования фильма при изменении размеров окна проектора. В этом случае его размеры строго соответствуют размерам рабочей области, заданным в рабочей среде, однако при изменении размеров окна проигрывателя может стать видна часть вспомогательной области. Значение `true` приводит к тому, что фильм автоматически масштабируется (пропорционально) в окне проигрывателя таким образом, чтобы занимать максимальную область. Для автономного проигрывателя данный параметр дает результат, аналогичный присваиванию свойству `Stage.scaleMode` значения `"noScale"` (`false`) или `"showAll"` (`true`)
`"showmenu"`	`"true"` или `"false"`	Включает или отключает отображение главного меню автономного проигрывателя. Значение `true` также приводит к тому, что контекстное меню, которое появляется при щелчке правой кнопкой мыши в окне Flash Player, будет содержать полный набор команд (управление воспроизведением, масштабирование, качество и т. д.). При установке значения `false` в контекстном меню будут доступны только команды **Settings...** и **About Macromedia Flash Player 7...**. Для автономного проигрывателя данный параметр дает результат, аналогичный применению свойства `Stage.showMenu`

Таблица 22.2 *(окончание)*

Команда	Параметр	Назначение
"trapallkeys"	"true" или "false"	Значение true приводит к тому, что нажатие любых клавиш (в том числе и их сочетаний) перехватывается объектом Key, то есть передается непосредственно фильму. Это исключает возможность использования горячих клавиш автономного проигрывателя для выполнения различных действий (например, выход из проектора <Ctrl>+<Q>, выход из полноэкранного режима <Esc> и т. д.)
"quit"	нет	Закрывает автономный проектор
"exec"	Имя исполняемого файла, расположенного в папке fscommand	Позволяет запустить внешнее приложение (исполняемый файл с расширением exe, bat и т. д.). Вызываемое приложение должно находиться в папке в одном каталоге с файлом проектора

В таблице приведено довольно подробное описание параметров функции fscommand(), поэтому ограничимся только некоторыми дополнительными комментариями.

Применение полноэкранного режима может привести к существенному снижению быстродействия проигрывателя, особенно при интенсивном использовании анимации.

Перехват нажатия клавиш и их сочетаний не распространяется на стандартные сочетания, используемые операционной системой, например, <Ctrl>+<Alt>+<Del> или <Ctrl>+<F4>.

Запуск внешнего приложения может быть осуществлен только из проектора (не просто из SWF-документа, запущенного в автономном проигрывателе) и только в том случае, если соответствующий исполняемый файл находится в папке с именем fscommand, расположенной в том же каталоге, что и файл проектора. Окно внешнего приложения открывается поверх текущего окна. Используя возможность запуска внешнего приложения из проектора, можно обеспечить навигацию между разделами автономной презентации, представляющими собой отдельные проекторы. Так, при необходимости перехода к следующему документу-разделу можно использовать приведенный ниже код.

```
nextBut.onPress = function(){
fscommand("exec", "coda.exe"); // Запускаем следующий документ
fscommand("quit"); // Закрываем текущий
};
```

Следующий код демонстрирует использование основных возможностей функции fscommand():

```
fscommand("fullscreen","false");// Включаем полноэкранный режим
fscommand("allowscale","false");// Запрещаем масштабирование фильма
fscommand("showmenu","false");// Отключаем главное и контекстное меню
```

```
fscommand("trapallkeys","true");// Включаем перехват нажатия клавиш
launchBut.onPress=function(){
fscommand("exec","coda.exe");// Запускаем внешнее приложение
};
quitBut.onPress=function(){
fscommand("quit");// Закрываем проектор. Выход из презентации
};
```

Класс *ContextMenu*. Управление контекстным меню проигрывателя

Независимо от того, в какой среде воспроизводится Flash-фильм, в автономном проекторе или в браузере, при нажатии правой кнопки мыши внутри окна Flash Player выводится стандартное контекстное меню проигрывателя. В этой главе уже рассматривались возможности, позволяющие скрывать стандартные элементы контекстного меню. С выходом Flash MX 2004 в ActionScript появились возможности более детального контроля элементов контекстного меню. Теперь разработчик может включать или отключать отображение произвольного сочетания стандартных элементов меню, добавлять в него свои собственные команды и, что особенно интересно, ассоциировать содержимое контекстного меню с конкретным объектом, расположенным на сцене. Управление контекстным меню осуществляется с помощью классов `ContextMenu` и `ContextMenuItem`, описанию возможностей которых посвящен настоящий раздел. Вначале кратко рассмотрим общий механизм управления контекстным меню.

Механизм управления контекстным меню

Основной особенностью программного управления контекстным меню является то, что каждому экземпляру клипа, кнопки, текстового блока или целому уровню стека документов можно назначить собственное контекстное меню, содержащее произвольный набор команд. Для того чтобы это стало возможным, очевидно, необходимо, во-первых, отдельно хранить информацию о представлении каждого контекстного меню и, во-вторых, связать эту информацию с конкретным объектом, так чтобы при нажатии правой кнопки мыши на объекте выводилось соответствующее ему меню.

Представлением конкретного контекстного меню является объект `ContextMenu`, который традиционно создается как экземпляр соответствующего класса с помощью оператора `new` и конструктора. Объект `ContextMenu` хранит информацию о составе команд, входящих в представляемое им контекстное меню и о выполняемых ими действиях. Несмотря на то, что контекстное меню тесно связано с определенным элементом в фильме, при создании объекта `ContextMenu` он, в отличие, например, от объектов `Color` или `Sound`, не привязывается к какому-то "материальному" объекту. Связывание объекта `ContextMenu` и соответствующего объекта в фильме выполняется уже после его создания. Это обеспечивает большую гибкость, поскольку позволяет использовать для нескольких объектов фильма одинаковые контекстные

меню или, наоборот, при необходимости выводить разные контекстные меню для одного и того же объекта.

При создании объекта ContextMenu используется следующий синтаксис:

```
var cm:ContextMenu = new ContextMenu(функция)
```

Здесь *cm* — это имя переменной, в которой сохраняется ссылка на созданный объект ContextMenu, возвращаемая конструктором.

Параметр *функция* — это ссылка на функцию, которая будет вызываться при нажатии правой кнопки мыши, непосредственно перед появлением контекстного меню. Данный параметр является необязательным. Его применение может оказаться полезным при необходимости настройки элементов меню в зависимости от состояния тех или иных элементов фильма. Так, например, создавая пользовательское меню, содержащее команды для управления анимацией, можно выводить в меню либо команду "Остановка", либо команду "Запуск", в зависимости от того, выполняется в данный момент анимация или нет. Такую проверку стоит поручить функции, ссылку на которую можно передать конструктору ContextMenu(). При вызове данной функции ей всегда передаются два параметра: первый параметр — это ссылка на объект, с которым ассоциировано контекстное меню, а второй — это ссылка на объект ContextMenu, представляющий данное контекстное меню. При необходимости эти параметры могут быть использованы инструкциями в теле функции.

Отметим, что передача ссылки на функцию конструктору — это не единственный способ выполнить настройку меню перед его выводом, поскольку в классе ContextMenu имеется специальный обработчик события onSelect, речь о котором пойдет далее.

Создав экземпляр класса ContextMenu, его необходимо связать с конкретным элементом фильма. Это может быть объект MovieClip, Button, TextField или уровень стека документов _levelN (где N — номер уровня). Для того чтобы связать контекстное меню, представленное объектом ContextMenu, с конкретным объектом, необходимо воспользоваться свойством menu. Данное свойство имеется у всех перечисленных объектов. В качестве значения оно принимает ссылку на объект ContextMenu, представляющий контекстное меню, которое должно выводиться при нажатии правой кнопки мыши на объекте. Так, например, для того чтобы при нажатии правой кнопки мыши на клипе mc появилось контекстное меню, представленное объектом myCM, нужно использовать следующий код:

```
mc.menu = myCM;
```

Если при нажатии правой кнопки мыши курсор находится на нескольких расположенных друг над другом объектах, выводится контекстное меню для верхнего из них.

Приведем простой пример, демонстрирующий создание объекта ContextMenu с передачей конструктору параметра. В листинге 22.4 приведен сценарий, затеняющий содержимое сцены при выводе контекстного меню проигрывателя. Затенение устанавливается путем рисования серого прозрачного прямоугольника, перекрывающего всю область окна проигрывателя, на верхнем свободном уровне стека. Затенение снимается при нажатии левой кнопки мыши вне контекстного меню путем удаления клипа, содержащего нарисованный прямоугольник.

Листинг 22.4. Затенение содержимого сцены при выводе контекстного меню

```
// Создаем объект ContextMenu
var my_cm: = new ContextMenu(dim);
/* Функция, затеняющая содержимое _root и снимающая затемнение при нажатии левой кнопки мыши */
function dim() {
        if(!dimmed){// Если затенение снято, устанавливаем его
        _root.createEmptyMovieClip("screen",
_root.getNextHighestDepth());
        with (_root.screen) {
                moveTo(-2000, -2000);
                beginFill(0x000000, 30);
                lineTo(4000, -2000);
                lineTo(4000, 4000);
                lineTo(-2000, 4000);
                lineTo(-2000, -2000);
        }
        dimmed = true; // Затенение установлено, поэтому переключаем флаг
        }
                // Снимаем затенение при нажатии левой кнопки мыши
                screen.onMouseDown = function() {
                this.removeMovieClip();
                dimmed = false; // Переключаем флаг
        };
}
// Ассоциируем меню с нулевым уровнем стека
_level0.menu = my_cm;
```

Управление стандартным содержимым меню

Содержимое стандартного меню проигрывателя варьируется в зависимости от того, на каком объекте была нажата правая кнопка мыши. Для любого содержимого, кроме выделяемого текста, контекстное меню содержит стандартный набор команд (рис. 22.4, *а*). Контекстное меню для выделяемых и/или редактируемых текстовых полей содержит стандартный набор команд по работе с текстом (рис. 22.4, *б*). Особый вид имеет меню для текстовой гиперссылки (рис. 22.4, *в*). Если указанный ресурс не может быть загружен в проигрыватель или в проигрывателе нет документов, контекстное меню состоит только из двух пунктов **Movie not loaded...** и **About Macromedia Flash Player...**.

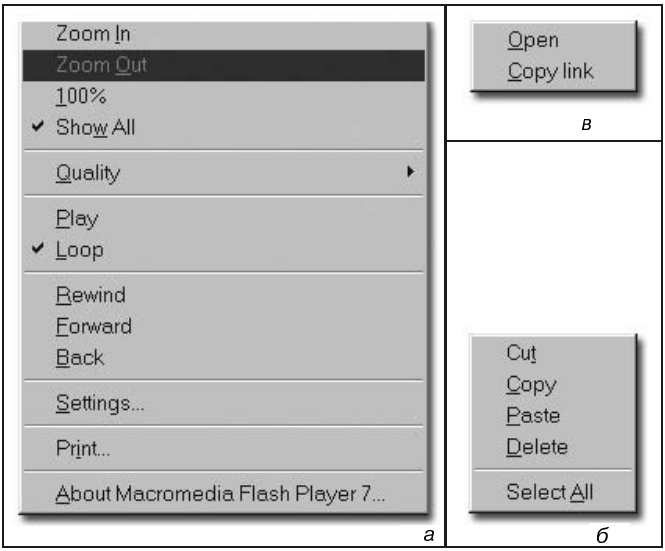


Рис. 22.4. Типы стандартного контекстного меню проигрывателя:
а — основное меню фильма; *б* — меню выделяемого/редактируемого текста;
в — меню текстовой гиперссылки

Класс `ContextMenu` позволяет контролировать только основное меню фильма и меню текстового блока. Причем сразу стоит сказать, что на сегодняшний день ActionScript не располагает возможностями, позволяющими избежать вывода контекстного меню или скрыть два стандартных элемента **Settings...** и **About Macromedia Flash Player...**.

Сокрытие элементов стандартных меню

Ранее в этой главе мы уже познакомились со свойством `Stage.showMenu`, а также функцией `fscommand()`, позволяющими скрыть основное контекстное меню. Особенностью этих возможностей является то, что они действуют глобально на все стандартное содержимое контекстного меню и в пределах целого документа. В некоторых ситуациях может потребоваться отключить только часть стандартных команд (например, только команды изменения масштаба просмотра), оставив остальные. Для избирательного вывода стандартных команд контекстного меню можно воспользоваться свойством `builtInItems` объекта `ContextMenu`.

Свойство `builtInItems` представляет собой объект, содержащий набор свойств, каждое из которых соответствует стандартной команде (или группе команд) основного контекстного меню. Данные свойства принимают булевы значения, определяющее, будет соответствующая команда отображаться в качестве элемента меню или нет. Ниже приведен перечень свойств объекта `builtInItems`, соответствующих элементам стандартного контекстного меню:

- `zoom` — соответствует первой секции из 4-х команд, управляющих масштабом отображения фильма;

- `quality` — соответствует пункту меню **Quality**, позволяющему установить степень сглаживания графики;
- `play` — соответствует команде **Play**, запускающей воспроизведение основной монтажной линейки;
- `loop` — соответствует команде **Loop**, управляющей режимом зацикленного воспроизведения основной монтажной линейки;
- `rewind` — соответствует команде **Rewind**, перемещающей воспроизводящую головку в первый кадр основной монтажной линейки фильма;
- `forward_back` — соответствует командам **Forward** и **Back**, перемещающим воспроизводящую головку на основной монтажной линейке на один кадр вперед и назад, соответственно, с последующей остановкой (аналогично применению функций `nextFrame()` и `prevFrame()`);
- `print` — соответствует команде **Print...**, позволяющей вывести фильм на печать.

Для того чтобы отключить одну из стандартных команд в контекстном меню, представляемом объектом `ContextMenu`, достаточно присвоить соответствующему свойству объекта значение `false`.

Так, например, для того чтобы исключить возможность изменения масштаба просмотра фильма, можно использовать следующий код:

```
mainCM = new ContextMenu();
mainCM.builtInItems.zoom = false;
_level0.menu = mainCM;
```

А следующий код скрывает все стандартные команды контекстного меню:

```
mainCM = new ContextMenu();
for (var i in mainCM.builtInItems){
mainCM.builtInItems[i] = false;
}
_level0.menu = mainCM;
```

Для удобства разработчиков в классе `ContextMenu` существует специальный метод, позволяющий скрыть все стандартные элементы меню, подобно тому, как это было сделано в предыдущем примере. Этот метод называется `hideBuiltInItems()`. Хотя приведенный код, выполняющий сокрытие элементов меню с помощью цикла `for...in`, не занимает много места, его можно было бы заменить следующим, более простым кодом:

```
mainCM = new ContextMenu();
mainCM.hideBuiltInItems();
_level0.menu = mainCM;
```

Примечание

Метод `hideBuiltInItems()` не позволяет скрывать стандартные элементы меню редактируемого текстового блока.

Копирование контекстного меню

В классе ContextMenu имеется метод copy(), позволяющий создать копию объекта ContextMenu. Правда, особенностью данного метода является то, что полученный в результате его применения объект-копия содержит не свои индивидуальные объекты builtInItems и customItems (массив для хранения пользовательских элементов меню, см. далее), а только ссылки на соответствующие свойства объекта-оригинала. Скорее всего, это является следствием ошибки, допущенной разработчиками Macromedia.

Следующий пример иллюстрирует особенности применения данного метода.

```
// Создаем объект ContextMenu
var:ContextMenu = new ContextMenu();
// Создаем копию cm
var cm_copy:ContextMenu = cm.copy();
// Отключаем пункт zoom в меню cm_copy
cm_copy.builtInItems.zoom = false;
// При этом данный пункт отключается и в исходном объекте меню cm
trace(cm.builtInItems.zoom); // Возвращает false
```

Обработка вызова контекстного меню

Как уже говорилось, при создании объекта ContextMenu конструктору в качестве параметра может быть передана ссылка на функцию, которая будет вызываться при нажатии правой кнопки мыши, но перед выводом содержимого контекстного меню. Это дает возможность настроить контекстное меню в зависимости от состояния определенных элементов или процессов в фильме.

В классе ContextMenu имеется обработчик события onSelect, которое генерируется в промежутке между нажатием правой кнопки мыши и перед выводом меню. В принципе, использование обработчика события onSelect аналогично передаче конструктору ContextMenu в качестве параметра ссылки на функцию. Различие состоит, например, в том, что метод обработчика onSelect() может быть легко удален с помощью оператора delete.

Определение метода обработчика onSelect() имеет следующий синтаксис:

```
cm.onSelect = function(объект, меню){
блок кода;
...
};
```

Здесь *cm* — это объект ContextMenu.

Функции, которая присваивается свойству onSelect, при вызове передаются два параметра. Параметр *объект* — это ссылка на объект, с которым ассоциировано контекстное меню. Параметр *меню* — это ссылка на объект *cm*, представляющий данное контекстное меню.

Добавление и настройка пользовательских элементов меню

Наряду с управлением стандартными элементами меню ActionScript позволяет создавать пользовательские элементы, предназначенные для решения специфических практических задач. Так, например, можно поместить в контекстное меню команды, управляющие воспроизведением конкретного клипа, или команды, позволяющие выполнять дублирование (копирование) и удаление объектов, расположенных на сцене. Немаловажным преимуществом этих возможностей является то, что пользовательские элементы меню могут быть названы по-русски, что позволяет лучше адаптировать проект для русскоязычной аудитории.

Пользовательские элементы меню всегда размещаются над стандартными элементами и отделяются от них горизонтальной полосой (separator). При необходимости пользовательские элементы можно разделять такими линиями между собой для того, чтобы объединить тематически близкие элементы в группы. Пользовательские элементы меню можно делать неактивными или отключать их отображение. Каждый элемент может занимать только одну строку.

Максимальное количество пользовательских элементов не может превышать 15.

Для создания пользовательских элементов меню применяется специальный класс, который называется `ContextMenuItem`. Механизм создания меню довольно прост. Каждый пользовательский элемент представляет собой объект `ContextMenuItem`. У этого объекта имеется ряд свойств, управляющих его визуализацией в меню. Обеспечение функциональности пользовательского элемента меню реализуется с помощью обработчика события `onSelect` класса `ContextMenuItem`. В этом разделе мы подробно рассмотрим всю описанную процедуру и приведем практический пример применения пользовательских элементов меню.

Создание пользовательского элемента меню

Создание пользовательского элемента меню начинается с создания экземпляра класса `ContextMenuItem`. При создании объекта `ContextMenuItem` конструктору могут быть переданы параметры, определяющие характеристики визуализации элемента меню и выполняемые им действия. В общем случае для этого используется следующий синтаксис:

```
var cmItem:ContextMenuItem = new ContextMenuItem (подпись,
                                                  функция,
                                                  разделитель,
                                                  активация,
                                                  отображение
                                                 );
```

Здесь `cmItem` — переменная, в которой сохраняется ссылка на объект `ContextMenuItem`.

Формальные параметры, передаваемые конструктору, являются необязательными. Дело в том, что у объекта ContextMenuItem имеется ряд свойств и обработчик события, которые могут быть определены уже после создания объекта. Передача параметров конструктору позволяет сразу же при создании объекта задать ему все необходимые параметры. В дальнейшем параметры, переданные конструктору, могут быть переопределены с помощью соответствующих свойств. Далее приводится описание каждого свойства и соответствующего ему параметра конструктора.

Свойство *caption*. Подпись элемента

Свойство caption (параметр конструктора *подпись*) в качестве значения принимает строку, которая будет выводиться в контекстном меню в качестве названия пункта меню.

Строка, присваиваемая данному свойству, должна содержать, как минимум, один печатный символ (не пробельный). Все управляющие символы (перенос строки, табуляция и др.), используемые в названии элемента, игнорируются. Названия пользовательских элементов меню не должны совпадать с названиями стандартных элементов независимо от регистра символов и знаков препинания. В именах пользовательских элементов запрещается использовать следующие слова: Macromedia, Flash Player, Settings. Нарушение любого из перечисленных правил приведет к тому, что элемент не будет функционировать и отображаться в меню.

Свойство *separatorBefore*. Разделитель элементов

Свойство separatorBefore (параметр конструктора *разделитель*) принимает булево значение, определяющее, будет ли в контекстном меню над данным элементом находиться разделительная линия. Если значение этого параметра true — разделитель будет выводиться над элементом, если значение этого параметра false — разделитель будет отсутствовать. Данный параметр не является обязательным. По умолчанию разделитель не выводится.

Свойство *enabled*. Активация элемента

Свойство enabled (параметр конструктора *активация*) принимает булево значение, определяющее, является ли элемент меню активным. Неактивный элемент отображается светло-серым цветом и, несмотря на то, что он может быть "выбран", не выполняет никаких действий. Если значение этого параметра true — элемент активен, если значение этого параметра false — элемент неактивен. Этот параметр является необязательным. По умолчанию элемент активен.

Свойство *visible*. Отображение элемента

Свойство visible (параметр конструктора *отображение*) принимает булево значение, определяющее, будет ли элемент отображаться в меню. Так, например, одновременно отображать команды "Стоп" и "Пуск", как правило, нет смысла, поскольку объект может находиться только в одном из состояний. Поэтому в меню можно добавить два пользовательских элемента, но в зависимости от ситуации отображать только один из них. Если свойство принимает значение true — объект отображает-

ся в меню, в противном случае — нет. Этот параметр является необязательным. По умолчанию элемент отображается в меню.

Обеспечение функциональности. Обработчик *onSelect()*

Для того чтобы пользовательский элемент меню мог выполнять определенные действия, ему необходимо сопоставить функцию, которая будет вызываться при выборе данного элемента. Это можно сделать двумя способами. Во-первых, ссылку на функцию можно передать конструктору `ContextMenuItem()` в качестве второго параметра *`функция`*. Во-вторых, уже после создания объекта `ContextMenuItem` можно определить метод обработчика `onSelect()`, снабдив его требуемой функциональностью.

Определение метода обработчика `onSelect()` синтаксически очень похоже на определение одноименного метода объекта `ContextMenu`.

```
cmItem.onSelect = function(объект, элемент меню){
блок кода;
...
};
```

Здесь *`cmItem`* — это объект `ContextMenuItem`.

Функции, которая присваивается свойству `onSelect`, при вызове передаются два параметра. Параметр *`объект`* — это ссылка на объект, нажатие правой кнопки мыши на котором инициировало появление контекстного меню. Параметр *`элемент меню`* — это ссылка на объект *`ContextMenuItem`*, представляющий выбранный пользовательский элемент меню *`cmItem`*.

Функция, ссылка на которую передается конструктору *`ContextMenuItem()`* в качестве параметра, также принимает два параметра *`объект`* и *`элемент меню`*. Ссылки, передаваемые этими параметрами, могут быть использованы инструкциями в теле функции, например, для того, чтобы выполнить определенные действия с объектом на сцене при выборе пункта меню.

Копирование пользовательского элемента меню

Для того чтобы скопировать объект `ContextMenuItem`, можно воспользоваться методом `copy()`, имеющимся в классе `ContextMenuItem`. Ранее мы уже рассматривали одноименный метод класса `ContextMenu`, позволяющий скопировать объект, представляющий контекстное меню. В отличие от своего тезки метод `ContextMenuItem.copy()` работает корректно и действительно возвращает новый объект `ContextMenuItem`, который содержит полный набор свойств оригинала. Изменение свойств объекта, полученного в результате копирования, не приводит к изменению значений соответствующих свойств оригинала, поэтому данный метод удобно применять в ситуации, требующей создания элементов меню, обладающих близкой функциональностью.

Добавление пользовательского элемента в меню

После создания, настройки параметров и сопоставления с определенной функцией объект необходимо поместить в контекстное меню. Для этого используется очень простая операция. У объекта `ContextMenu` имеется свойство `customItmes`, которое представляет собой массив, предназначенный для хранения пользовательских элементов меню. Для того чтобы поместить пользовательский элемент в контекстное меню, необходимо просто добавить соответствующий объект *`ContextMenuItem`* в массив `customItems` объекта `ContextMenu`, представляющего контекстное меню. Таким образом, в общих чертах процедура создания пользовательского элемента меню включает следующие этапы.

1. Создание объекта *`ContextMenuItem`*.
2. Настройка параметров этого объекта (подпись, активация и т. д.).
3. Определение обработчика `onSelect` (или передача соответствующего параметра конструктору на шаге 1).
4. Добавление созданного объекта *`ContextMenuItem`* в массив `customItems` объекта *`ContextMenu`*.

Следует иметь в виду, что пользовательские элементы располагаются в меню в соответствии с их порядковыми номерами в массиве `customItems`. Чем ближе к началу массива находится объект *`ContextMenuItem`*, тем выше будет расположен соответствующий пункт в контекстном меню.

Следующий сценарий убирает из контекстного меню все стандартные элементы меню и добавляет туда пользовательский элемент, содержащий информацию об авторских правах. При выборе этого пункта меню будет запущена почтовая программа, и пользователь получит возможность написать письмо авторам.

```
var cm:ContextMenu = new ContextMenu();
cm.hideBuiltInItems();
var cmItem:ContextMenuItem = new ContextMenuItem(
                                        "© 2004 Д.Альберт, Е.Альберт",
                                        doSend
                                        );
cm.customItems.push(cmItem);
function doSend(){
       getURL("mailto:book_flash@mail.ru");
}
_level0.menu=cm;
```

Примеры управления контекстным меню

В завершение темы, посвященной контекстному меню, рассмотрим несколько несложных примеров, демонстрирующих основные аспекты настройки и формирования пользовательских контекстных меню.

В листинге 22.5 приводится сценарий, позволяющий динамически дублировать и удалять со сцены клипы, используя для этого соответствующие команды контекстного меню. Характерной особенностью приведенного сценария является то, что набор пользовательских элементов меню формируется непосредственно при его вызове. Весь код, выполняющий создание и добавление пользовательских элементов меню, помещен в блок обработчика события `onSelect` объекта `ContextMenu` и, соответственно, выполняется после нажатия правой кнопки мыши, но перед фактическим выводом меню.

В исходном файле, помещенном на компакт-диск, также имеется возможность перетаскивания объектов. Соответствующий код помещен непосредственно на монтажную линейку символа с идентификатором связи `clip`.

Листинг 22.5. Дублирование и удаление объектов с помощью команд контекстного меню

```
// Размещаем клипы на сцене
for (i = 1; i < 11; i++) {
       this.attachMovie("clip", "clip" + i, i, {_x:i * 60 - 30, ↳
       _y:200});
}
// Создаем объект ContextMenu
var cm:ContextMenu = new ContextMenu();
// Скрываем стандартные элементы меню
cm.hideBuiltInItems();
// При появлении меню...
cm.onSelect = function(obj:Object, menu:ContextMenu) {
       // Создаем элемент меню для дублирования клипа
       var duplicateItem = new ContextMenuItem();
       duplicateItem.caption = "Создать дубликат";
       duplicateItem.onSelect = function(obj) {
              var depth = obj._parent.getNextHighestDepth();
              obj.duplicateMovieClip("obj" + depth, depth, {_x:obj._x + ↳
              20, _y:obj._y + 20});
              /* Ассоциируем контекстное меню с дубликатом клипа, чтобы
              его тоже можно было копировать */
              obj._parent["obj" + depth].menu = cm;
       };
       // Создаем элемент меню для удаления клипа
       var deleteItem = new ContextMenuItem();
       deleteItem.caption = "Удалить объект";
       deleteItem.onSelect = function(obj) {
```

```
                obj.removeMovieClip();
        };
        // Добавляем пользовательские элементы в меню
        cm.customItems.push(duplicateItem);
        cm.customItems.push(deleteItem);
};
// Ассоциируем контекстное меню с каждым клипом на текущей линейке
for (i in this) {
        if (this[i] instanceof MovieClip) {
                this[i].menu = cm;
        }
}
```

Стандартное контекстное меню по умолчанию содержит набор команд, позволяющих управлять воспроизведением только основной монтажной линейки. Если анимация размещается на монтажной линейке клипа, стандартное меню оказывается бесполезным. Кроме того, некоторых пользователей может смутить то, что в стандартном меню выводятся команды на английском языке. В листинге 22.6 приводится пример создания пользовательского контекстного меню, позволяющего останавливать и запускать анимацию клипа с именем `runner`. Для этих целей создаются два объекта `ContextMenuItem`, которые добавляются в контекстное меню. Поскольку анимация клипа может быть либо запущена, либо остановлена, одновременное наличие обоих элементов в контекстном меню не требуется. В связи с этим, при вызове контекстного меню отображение одного из пользовательских элементов отключается с помощью свойства `ContextMenuItem.visible` (в качестве упражнения читатель может вместо отключения отображения "лишнего" элемента сделать его неактивным с помощью свойства `ContextMenuItem.enabled`). Для контроля состояния анимации используется стандартный алгоритм кнопки-флага.

Листинг 22.6. Управление анимацией клипа с помощью контекстного меню

```
// Создаем объект ContextMenu
var cm = new ContextMenu(switchMenu);
// Скрываем стандартные элементы меню
cm.hideBuiltInItems();
// Элемент меню для остановки анимации
var stopItem=new ContextMenuItem("Стоп", doStop);
// Элемент меню для запуска анимации
var playItem=new ContextMenuItem("Пуск", doPlay);
// Добавляем пользовательские элементы в меню
cm.customItems.push(stopItem);
cm.customItems.push(playItem);
```

```
// Переменная-флаг, определяющая состояние анимации
var bPlay:Boolean=true;
/* Функция, выполняющая переключение элементов меню в зависимости от
состояния анимированного клипа runner. Данная функция вызывается после
нажатия правой кнопки мыши, но перед выводом меню */
function switchMenu(){
if(bPlay){
       stopItem.visible=true;
       playItem.visible=false;
}
else{
       stopItem.visible=false;
       playItem.visible=true;
       }
}
// Функция для остановки анимации
function doStop(){
       runner.stop()
       bPlay=false;
}
// Функция для запуска анимации
function doPlay(){
       runner.play()
       bPlay=true;
}
// Ассоциируем меню с клипом runner
runner.menu=cm;
```

Информация о системе и оборудовании. Объект *System.capabilities*

При разработке профессионального Flash-проекта, предназначенного для широкой аудитории, необходимо снабдить его гибкостью, позволяющей приспособиться к различиям в операционных системах и разнообразию технических характеристик оборудования пользователей. Так, например, разрабатывая автономную презентацию, как правило, ориентируются на среднестатистическое разрешение экрана, используемое целевой аудиторией, однако при этом неплохо было бы учесть и то, что разрешение экрана, используемое конкретным пользователем, может быть другим. Для решения этой задачи необходимо определить текущее разрешение экрана и,

исходя из полученного значения, либо установить соответствующий режим масштабирования фильма, либо загрузить SWF-документ, адаптированный для данного разрешения. Подобных примеров можно привести очень много, например, перед тем, как выполнять загрузку звукового файла, было бы неплохо проверить, располагает ли операционная система пользователя средствами воспроизведения звука, перед выводом документа на печать стоит убедиться в том, что операционная система поддерживает возможность печати и т. д. Особенно остро вопросы, связанные с проверкой различных характеристик операционной системы и оборудования, встают при разработке проекта для мобильных устройств.

Для получения информации об оборудовании и параметрах операционной системы, в которой воспроизводится Flash-фильм, используется объект `System.capabilities`. Данный объект формально является статическим свойством класса `System` и, в свою очередь, имеет целый набор свойств, содержащих разнообразную информацию, получаемую от операционной системы в виде серверной строки. Серверная строка формируется из набора пар "имя=значение", описывающих различные характеристики операционной системы, используя URL-кодировку. Свойства объекта соответствуют элементам серверной строки, но имеют интуитивно понятные имена. Впрочем, объект `System.capabilities` располагает свойством, позволяющим получить полную серверную строку в URL-кодировке. Эта информация при необходимости может быть передана серверному сценарию с помощью методов `GET` или `POST`. Все свойства объекта `System.capabilities`. доступны только для чтения.

Перечень свойств объекта `System.capabilities`, их описания, а также соответствующие им элементы серверной строки представлены в табл. 22.3.

Таблица 22.3. *Свойства объекта* `System.capabilities`

Свойство	Описание	Серверная строка
`avHardwareDisable`	Определяет, разрешено ли фильму использовать микрофон или камеру пользователя. Если доступ к микрофону и камере разрешен, данное свойство возвращает значение `true`, если доступ запрещен или соответствующее оборудование у пользователя не установлено, возвращается значение `false`	AVD
`hasAccessibility`	Определяет, поддерживает ли операционная система взаимодействие между Flash Player и программами для людей с ограниченными возможностями, например, для чтения с экрана (screen reader). Если взаимодействие поддерживается, данное свойство возвращает значение `true`, в противном случае — `false`	ACC
`hasAudio`	Определяет, поддерживает ли операционная система возможность воспроизведения звука. Если такая возможность имеется, возвращает значение `true`, в противном случае — `false`	A

Таблица 22.3 *(продолжение)*

Свойство	Описание	Серверная строка
`hasAudioEncoder`	Определяет, поддерживает ли операционная система возможность кодирования потокового звука, например, поступающего с микрофона. Если такая возможность имеется, данное свойство возвращает значение `true`, в противном случае — `false`	AE
`hasEmbeddedVideo`	Определяет, поддерживает ли операционная система возможность воспроизведения фильма с внедренным видео. Если такая возможность имеется, данное свойство возвращает значение `true`, в противном случае — `false`	EV
`hasMP3`	Определяет, имеется ли в операционной системе пользователя MP3-декодер. Возвращает значение `true`, если декодер имеется, в противном случае возвращает `false`	MP3
`hasPrinting`	Определяет, поддерживает ли операционная система возможность печати. Если такая возможность имеется, возвращает значение `true`, в противном случае — `false`	PR
`hasScreenBroadcast`	Определяет, поддерживает ли проигрыватель возможность создания приложений, передаваемых с помощью Flash Communication Server (например, при создании видеоконференции). Если такая возможность имеется, возвращает значение `true`, в противном случае — `false`	SB
`hasScreenPlayback`	Определяет, поддерживает ли проигрыватель воспроизведение приложений, передаваемых с помощью Flash Communication Server. Если такая возможность имеется, возвращает значение `true`, в противном случае — `false`	SP
`hasStreamingAudio`	Определяет, поддерживает ли проигрыватель воспроизведение потокового звука. Если такая возможность имеется, возвращает значение `true`, в противном случае — `false`	SA
`hasStreamingVideo`	Определяет, поддерживает ли проигрыватель воспроизведение потокового видео (в формате FLV). Если такая возможность имеется, возвращает значение `true`, в противном случае — `false`	SV
`hasVideoEncoder`	Определяет, поддерживает ли операционная система возможность кодирования потокового видео, например, поступающего с Web-камеры. Если такая возможность имеется, данное свойство возвращает значение `true`, в противном случае — `false`	VE

Таблица 22.3 *(продолжение)*

Свойство	Описание	Серверная строка
`isDebugger`	Определяет, является ли проигрыватель, в котором воспроизводится фильм, обычной версией Flash Player или проигрывателем, допускающим возможности отладки (Debugger). Если фильм воспроизводится в проигрывателе с возможностями отладки, данное свойство возвращает `true`, в противном случае — `false`	DEB
`language`	Возвращает код языка, используемого в качестве языка интерфейса операционной системы в соответствии со стандартом ISO 639-1. Перечень кодов языков представлен в табл. 22.4. В Flash Player 7 данное свойство претерпело два изменения по сравнению с его работой в Flash Player 6. Во-первых, код языка для англоязычной операционной системы теперь не включает суффикс с указанием страны, как это было в Flash Player 6 (например, раньше для США использовалось значение en-US). Во-вторых, теперь язык определяется в соответствии с языком, используемым в качестве языка интерфейса операционной системы, в то время как в Flash Player 6 язык определялся на основании региональных настроек	L
`localFileReadDisable`	Определяет, разрешено ли проигрывателю выполнять чтение файлов, расположенных на жестком диске компьютера пользователя. Если данное свойство возвращает `true`, это означает, что доступ для чтения закрыт и Flash не может выполнять чтение и загрузку файлов с диска, используя методы `LoadMovie()`, `loadClip()`, `XML.load()` и `LoadVars.load()`. Также при этом отсутствует возможность использования общих библиотек (shared libraries), расположенных локально на диске. Тем не менее даже при получении значения `true` доступ к SOL-файлам, хранящим локальные объекты, разрешается. Если данное свойство возвращает значение `false`, чтение файлов с диска разрешено	LFD
`manufacturer`	Возвращает строку, определяющую, для какой платформы создан проигрыватель, осуществляющий воспроизведение фильма. Возвращаемая строка имеет следующий формат: `"Macromedia O/C"` (Здесь вместо `O/C` могут использоваться следующие значения: `"Windows"`, `"Macintosh"`, `"Linux"` или `"Other OS Name"` (другая система)	M

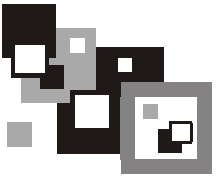
Таблица 22.3 *(окончание)*

Свойство	Описание	Серверная строка
os	Возвращает строку с названием операционной системы, в которой запущен проигрыватель. Допускаются следующие значения: "Windows XP", "Windows 2000", "Windows NT", "Windows 98/ME", "Windows 95", "Windows CE" (доступно только в средстве разработки Flash Player Software Development Kit), "Linux" и "MacOS"	OS
pixelAspectRatio	Возвращает число, определяющее отношение размера точки монитора к ее среднему размеру. Для стандартных мониторов значение этого свойства составляет 1	AR
playerType	Возвращает строку, определяющую, в каком типе проигрывателя воспроизводится фильм. Допускаются следующие значения: "stand-alone" — автономный проигрыватель, "external" — проигрыватель среды тестирования, "plug-in" — подключаемый модуль (в браузерах Netscape), "ActiveX" — элемент "ActiveX" (в браузере Internet Explorer)	PT
screenColor	Возвращает строку, определяющую цветность монитора. Допускаются следующие значения: "color" — цветной, "gray" — оттенки серого и "bw" — черно-белый	COL
screenDPI	Возвращает экранное разрешение (dpi), определяемое количеством точек на дюйм. Возвращаемое значение измеряется в пикселах	DP
screenResolutionX	Горизонтальный размер экрана в пикселах, определяемый используемым режимом монитора, например, 1600	R
screenResolutionY	Вертикальный размер экрана в пикселах, определяемый используемым режимом монитора, например, 1200	R
serverString	Возвращает полную серверную строку с применением URL-кодировки, содержащую все характеристики, приведенные в данной таблице	-
version	Возвращает строку, содержащую название платформы и текущую версию проигрывателя, в котором воспроизводится Flash-фильм, например: "WIN 7,0,19,0"	V

Таблица 22.4. *Коды языков в соответствии с ISO 639-1*

Язык	Код	Язык	Код	Язык	Код
Английский	en	Китайский (упрощенный)	zh-CN	Турецкий	tr
Венгерский	hu	Корейский	ko	Финский	fi
Голландский	nl	Немецкий	de	Французский	fr
Датский	da	Норвежский	no	Чешский	cs
Испанский	es	Польский	pl	Шведский	sv
Итальянский	it	Португальский	pt	Японский	ja
Китайский (традиционный)	zh-TW	Русский	ru	Другой	xu

Глава 23



Класс *PrintJob*. Печать из Flash-фильма

При печатании книги в нее всегда вкрадывается несколько ошибок, которые никто не заметит.

Закон публикаций Джоунса

Открыв сигнальный экземпляр на произвольной странице, автор тут же наткнется на самую грубую ошибку.

Следствие Блоха

Традиционной и широко используемой особенностью HTML является возможность распечатки как всей страницы целиком, так и отдельных ее элементов (текста и растровых изображений). В этом отношении Flash-фильм обладает определенной спецификой, поскольку его вывод на печать сопряжен с некоторыми трудностями. Если автор не запретил возможность печати, убрав из контекстного меню проигрывателя команду **Print...**, пользователь может, воспользовавшись этой командой, распечатать все кадры, расположенные на монтажной линейке основного фильма (или только кадры, разрешенные для печати автором). Тем не менее с помощью данной команды невозможно распечатать только конкретный элемент или область фильма, равно как и содержимое монтажной линейки вложенного символа. Кроме того, печать с помощью контекстного меню проигрывателя не передает прозрачность и цветовые трансформации, что может сделать печатное изображение очень далеким от оригинала. Для того чтобы дать пользователю более широкие возможности печати, необходимо использовать программный инструментарий ActionScript.

Возможности печати из фильма появились еще в Flash Player 4. Для этих целей использовались глобальные функции `print()` и `printAsBitmap()`, а также появившиеся уже в Flash Player 5 глобальные функции `printNum()` и `printAsBitmapNum()`. С выходом Flash Player 7 разработчики получили в свое распоряжение более совершенный, гибкий и универсальный инструмент в виде класса `PrintJob`, о котором и пойдет речь в настоящей главе.

Механизм создания и передачи задания на печать

Перед тем как перейти непосредственно к описанию инструментария класса PrintJob, рассмотрим общую схему его использования, для того чтобы сделать дальнейшее изложение более понятным.

Формирование и передача задания на печать включает несколько этапов.

1. Вначале выполняется создание экземпляра класса PrintJob, представляющего задание на печать. Для этих целей применяется оператор new и конструктор, не принимающий параметров:

   ```
   var prt:PrintJob = new PrintJob();
   ```

   Здесь *prt* — это переменная, в которой сохраняется ссылка на созданный объект PrintJob.

   Необходимо отметить, что в определенный момент времени в проигрывателе может быть задействован только один объект PrintJob. Это означает, что пока один объект PrintJob остается активным, новое задание на печать создать невозможно. Для того чтобы создать новый объект PrintJob, необходимо удалить предыдущий.

2. Передача задания на печать принтеру начинается с вызова метода start(). Вызов данного метода инициирует появление у пользователя на экране стандартного диалогового окна с настройками параметров печати для его текущего принтера. Если пользователь после настройки соответствующих параметров нажимает кнопку **OK**, задание на печать начнет передаваться спулеру операционной системы для постановки в очередь печати. *Спулер печати* (spooler) — это программа, которая принимает документы, посланные пользователем на печать, сохраняет их и отправляет один за другим на доступный принтер. После того как пользователь нажимает кнопку **OK**, появляется стандартное сообщение о состоянии выполнения печати, а объекту PrintJob передается информация о выполненных пользователем настройках печати.

3. Если пользователь запустил процедуру печати, нажав кнопку **OK**, далее необходимо вызвать метод addPage(), который передает спулеру содержимое, предназначенное для вывода на печать. Между вызовами метода start() и метода addPage() можно выполнить подготовку этого содержимого, например, масштабировать или повернуть изображение, так чтобы оно хорошо разместилось в области печати, заданной пользователем. Метод addPage() может вызываться последовательно несколько раз для передачи спулеру нескольких различных элементов, каждый из которых будет печататься отдельной страницей.

4. После передачи содержимого, которое должно быть распечатано, спулеру необходимо вызвать метод send(), который отправит страницы, находящиеся в очереди печати, на принтер.

5. И, наконец, в заключение необходимо удалить объект PrintJob для того, чтобы очистить спулер печати и получить возможность сформировать новое задание на печать (если в этом есть необходимость).

Необходимо иметь в виду, что весь код, осуществляющий создание и передачу задания на печать, должен размещаться в одном и том же месте. Причем это условие требует размещения всех инструкций не просто в одном кадре, но (при использовании обработчика события) в одном и том же блоке кода. Так, например, если вывод на печать из фильма должен осуществляться по нажатию кнопки, весь код, начиная от создания объекта `PrintJob` и заканчивая его удалением, должен быть помещен в тело метода обработчика события `onPress`. Несоблюдение данного правила приведет к невозможности выполнения печати.

Вызов окна с настройками параметров печати

Метод `start()` объекта `PrintJob` применяется для выполнения подготовки к печати. Его вызов приводит к появлению диалогового окна с настройками параметров печати (формат и тип бумаги, ориентация, цветность и т. д.) текущего принтера. До тех пор, пока окно с настройками принтера присутствует на экране, метод `start()` продолжает работать и ожидает окончательного выбора пользователя. Если пользователь, выполнив соответствующие настройки, нажимает кнопку **OK**, метод `start()` возвращает значение `true`. Если пользователь решил отменить печать и нажал на кнопку **Cancel**, метод `start()` возвращает значение `false`. В связи с возможностью отмены печати необходимо всегда проверять возвращаемое данным методом значение. Последующие действия по формированию и передаче задания на печать имеет смысл выполнять только в том случае, если возвращаемым значением метода `start()` является `true`.

Если метод `start()` возвращает `false`, печать выполняться не будет, поэтому необходимо удалить объект `PrintJob`, чтобы очистить спулер печати.

Если метод `start()` возвращает `true`, это означает, что пользователь, выполнив соответствующие настройки, разрешил выполнение печати. В классе `PrintJob` имеется пять свойств, используемых только для чтения и предназначенных для хранения настроек печати, выполненных пользователем. До вызова метода `start()` данные свойства имеют нулевые значения. Если метод `start()` возвращает `false`, значения этих свойств не определены (`undefined`). Если же метод `start()` возвращает `true`, данным свойствам присваиваются значения, отражающие настройки печати, выполненные пользователем. Перечень и описание этих свойств приводятся в табл. 23.1.

Таблица 23.1. *Свойства объекта* `PrintJob`

Свойство	Тип значения	Размерность	Описание
`paperHeight`	Number	точки (points)	Высота листа бумаги
`paperWidth`	Number	точки (points)	Ширина листа бумаги
`pageHeight`	Number	точки (points)	Высота области печати

Таблица 23.1 *(окончание)*

Свойство	Тип значения	Размерность	Описание
`pageWidth`	Number	точки (points)	Ширина области печати
`orientation`	String	безразмерное	Ориентация листа. Допустимые значения: `"landscape"` (альбомная ориентация), `"portrait"` (книжная ориентация)

Свойства объекта `PrintJob` могут оказаться очень полезны для того, чтобы, учитывая их значения, красиво и аккуратно отмасштабировать и закомпоновать выводимое на печать изображение на листе. Так, например, если пользователь установил альбомную ориентацию листа, а выводимое на печать изображение больше соответствует книжной ориентации, можно либо повернуть его, либо уменьшить масштаб и задать размеры печатаемой области так, чтобы оно при печати находилось в центре листа. Следует помнить, что все операции, связанные с подгонкой содержимого, выводимого на печать, под заданные пользователем параметры страницы, необходимо выполнить непосредственно после того, как метод `start()` возвратит значение `true`, но перед вызовом метода `addPage()`, отправляющим страницу с изображением в спулер печати.

При масштабировании графики, выводимой на печать, необходимо иметь представление о взаимосвязи единиц измерения, используемых в работе с печатным изображением и с изображением на экране монитора. Единицами измерения для печатного изображения являются *точки* (points). Экранными единицами измерения являются *пикселы* (pixels). Точка — это единица измерения, имеющая фиксированную физическую величину, составляющую 1/72 дюйма. Пиксел — это более абстрактная единица измерения, поскольку физическая величина пиксела зависит от экранного разрешения. Так, например, если монитор имеет разрешение 72 пиксела на дюйм, то физические размеры пиксела и точки совпадают. В своем внутреннем представлении Flash для хранения координат использует специальные единицы измерения, которые называются *твипы* (twips). Один твип составляет 1/20 логического пиксела. Размер логического пиксела совпадает с размером экранного пиксела при масштабе просмотра 100%. Применение твипов позволяет работать во Flash с дробными значениями пикселов, которые аппроксимируются с помощью сглаживания (вот почему у объектов с дробными координатами и размерами наблюдаются несколько размытые края). Ниже приводятся соотношения между точками, дюймами, сантиметрами и твипами:

1 точка = 1/72 дюйма = 20 твипов;

1 дюйм = 72 точки = 1440 твипов;

1 см = 567 твипов.

При экранном разрешении 72 dpi:

1 точка = 1пиксел.

Если метод `start()` возвращает значение `true`, необходимо сразу же вызвать метод `addPage()` и затем `send()`. Если между успешным вызовом метода `start()` и вызо-

вом метода send() проходит 10 секунд, Flash Player автоматически вызывает метод send(), который посылает на принтер все страницы, отправленные в спулер печати предыдущими вызовами метода addPage(), и прекращает постановку заданий на печать в очередь.

Отметим, что после вызова метода start() при появлении окна настройки параметров печати принтера все процессы, связанные с взаимодействием с пользователем, и в том числе анимация, прерываются до тех пор, пока окно настройки не будет закрыто. После его закрытия функциональность всех элементов восстанавливается. Правда, следует иметь в виду, что если принтер пользователя отключен, возобновления выполнения анимации может не произойти даже после закрытия окна настроек печати. Для того чтобы застраховаться от такой возможности, можно принудительно выполнить ее запуск перед удалением объекта PrintJob.

Отправка задания в спулер печати

После успешного запуска процедуры печати с помощью метода start() необходимо сформировать страницу для вывода на печать и отправить ее в спулер печати. Эту функцию выполняет метод addPage(). Метод addPage() помещает на страницу изображение отдельного клипа или содержимого уровня стека и отправляет эту страницу в спулер печати. Данный метод также позволяет задать размеры области изображения, которую следует выводить на печать, режим печати и номер кадра, содержимое которого требуется распечатать.

Метод addPage() имеет следующий синтаксис:

prt.addPage(*целевой объект*, *область печати*, *режим печати*, *кадр*)

Здесь *prt* — это объект PrintJob. Рассмотрим назначение каждого параметра:

- *целевой объект* — данный параметр представляет собой ссылку на целевой объект, изображение которого нужно распечатать. Данный параметр может принимать числовое или строковое значение. Для того чтобы вывести на печать содержимое уровня стека документов, необходимо указать номер этого уровня. Для того чтобы вывести на печать изображение клипа, необходимо указать имя соответствующего клипа в виде строки или просто передать ссылку на него, используя имя экземпляра;
- *область печати* — данный параметр является необязательным и представляет собой объект, задающий размеры области, которая должна быть выведена на печать. Объект *область печати* должен иметь четыре обязательных свойства:
  - xMin — горизонтальная координата левого верхнего угла;
  - xMax — горизонтальная координата правого нижнего угла;
  - yMin — вертикальная координата левого верхнего угла;
  - yMax — вертикальная координата правого нижнего угла.

  Значения координат этих свойств задаются в пикселах относительно точки регистрации объекта *целевой объект*. Координаты должны иметь такие значения, чтобы ширина и высота печатаемой области были больше нуля. При выводе на

печать левый верхний угол печатаемой области будет совмещен с левым верхним углом области печати, заданной в настройках принтера (размеры области печати хранятся в свойствах page.Width и page.Height). Если часть изображения выходит за правую и нижнюю границы области печати, эта часть при печати будет срезана.

Следует иметь в виду, что размер печатаемой области никак не связан с текущим масштабом печатаемого объекта. Так, пусть печатаемая область имеет размер 100×200 пикселов, соответственно, результирующее изображение будет состоять из 20 000 пикселов. Если с помощью свойств _xscale, _yscale перед вызовом метода addPage() размер объекта *целевой объект* будет уменьшен вдвое, то результирующее изображение по-прежнему будет состоять из 20 000 пикселов (100× ×200), хотя размер объекта будет вдвое меньшим (а фон будет занимать большую площадь).

Если параметр *область печати* отсутствует (или задан некорректно), размер печатаемой области будет соответствовать размерам ограничивающей рамки, заключающей все графическое содержимое, которое выводится на печать. Так, если требуется распечатать первый кадр клипа с именем animClip, то по умолчанию на печать будет выведена только область этого кадра, занятая графикой, причем на листе изображение будет позиционировано так, чтобы его левый верхний угол и левый верхний угол печатной области совпадали. Чтобы не указывать размеры области, но иметь возможность задания параметров *режим печати* и *кадр*, в качестве данного параметра можно передать значение null.

Следует иметь в виду, что цвет фона фильма также выводится на печать. Так, если при печати клипа с белым фоном размер печатаемой области задается большим, чем размер клипа, площадь между ограничивающей рамкой клипа и печатной областью листа будет окрашена в цвет фона фильма;

- *режим печати* — данный параметр является необязательным и представляет собой объект со свойством printAsBitmap. В зависимости от значения этого свойства изображение при печати будет растрироваться или выводиться на печать как векторное. Свойство printAsBitmap принимает булево значение. Если данному свойству присваивается значение true, изображение при печати будет растрироваться. Если данному свойству присваивается значение false, при наличии PostScript-принтера изображение будет печататься как векторное. При использовании принтера, не поддерживающего PostScript, изображение растрируется вне зависимости от значения свойства printAsBitmap. Печатать изображение в растровом формате имеет смысл в том случае, если оно содержит растровые элементы, прозрачность и цветовые эффекты. В противном случае векторная печать дает более высокое качество.

  Если параметр *режим печати* опущен или указан некорректно, по умолчанию выполняется печать в векторном режиме. Чтобы не указывать режим печати, но воспользоваться параметром *кадр*, можно в качестве значения данного параметра передать null;

- *кадр* — этот параметр является необязательным и позволяет указать номер кадра монтажной линейки объекта *целевой объект*, содержимое которого должно быть

выведено на печать. Если данный параметр опущен, на печать будет выведен текущий кадр объекта *целевой объект*, то есть тот кадр, в котором воспроизводящая головка находилась в момент вызова метода `start()`.

Метод `addPage()` возвращает значение `true`, если страница успешно передана спулеру, в противном случае возвращается значение `false`. Значение может быть получено, если вызову метода `addPage()` не предшествовал вызов метода `start()` или если пользователь прервал печать, нажав кнопку **Cancel** в окне состояния печати.

Метод `addPage()` может последовательно вызываться несколько раз для отправки спулеру печати новых страниц. При этом необходимо иметь в виду, что если очередной вызов метода `addPage()` возвращает значение `false`, дальнейшая передача новых страниц в спулер печати окажется невозможной (то есть дальнейшие попытки вызова метода `addPage()` не дадут результата). Тем не менее при последующем вызове метода `send()` все успешно переданные в спулер печати предыдущими вызовами `addPage()` страницы будут выведены на принтер.

Если после вызова метода `start()` объекты, предназначенные для вывода на печать, подвергались тем или иным модификациям для компоновки в пределах печатной области листа (масштабирование, смещение, поворот и т. д.), после вызова метода `addPage()` их можно вернуть в исходное состояние, чтобы фильм не подвергся изменениям. Так, например, если размер клипа превышает размер области печати, после вызова метода `start()` масштаб клипа можно уменьшить, а после вызова метода `addPage()`, отправляющего страницу с уменьшенным изображением клипа спулеру, масштаб клипа нужно восстановить.

Вывод содержимого на печать

Для того чтобы вывести на печать страницы, помещенные в спулер с помощью метода `addPage()`, нужно воспользоваться методом `send()`. Данный метод не принимает параметров и не возвращает значения:

```
prt.send()
```

Здесь *prt* — это объект `PrintJob`.

Следует иметь в виду, что вывод на печать осуществляется только в том случае, если в результате предшествующего вызова метода `start()` и хотя бы одного вызова метода `addPage()` были получены значения `true`. В противном случае вызов метода `send()` не будет иметь никакого результата.

Применение класса *PrintJob*

Теперь, после того как мы рассмотрели теоретические вопросы использования класса `PrintJob`, приведем несколько примеров, иллюстрирующих его практическое применение.

В листинге 23.1 демонстрируется простейший пример создания кнопки, отпускание которой инициирует запуск процедуры печати клипа с именем `animClip`. В данном примере методу `addPage()` передается только первый обязательный параметр, по-

этому при формировании страницы для отправки принтеру используются параметры, установленные по умолчанию. В частности, размер печатаемой области будет установлен равным размеру области, занимаемой графикой в текущем кадре клипа. Изображение будет по умолчанию печататься в векторном режиме и на печать будет выведен только текущий кадр клипа `animClip`. Обратите внимание на последовательно вложенные друг в друга условные предложения. Поскольку и метод `start()`, и метод `addPage()` возвращают булевы значения, их вызов можно использовать в составе проверочного выражения, так чтобы сразу оценить результат их выполнения и в зависимости от этого произвести (или нет) дальнейшие действия.

Листинг 23.1. Вывод текущего кадра клипа на печать по нажатию кнопки

```
// При отпускании кнопки printBut...
printBut.onRelease = function() {
       // создаем объект PrintJob
       var prt:PrintJob = new PrintJob();
       // Если пользователь нажал OK в окне настроек печати...
       if (prt.start()) {
              /* Отправляем страницу с текущим кадром клипа animClip в
              спулер печати с настройками по умолчанию... */
              if (prt.addPage(animClip)) {
                     /* Если страница передана спулеру, отправляем ее на
                     печать */
                     prt.send();
              }
       }
       // Вновь запускаем анимацию клипа
       animClip.play();
       /* Удаляем объект prt, чтобы очистить спулер печати, вне
       зависимости от того, была она выполнена или нет */
       delete prt;
};
```

Обратите внимание на последнюю строчку приведенного сценария. Здесь выполняется удаление объекта `PrintJob`, поскольку он уже исполнил свою роль и дальнейшему применению не подлежит. Удаление объекта `PrintJob` всегда следует выполнять после вызова метода `send()` вне зависимости от того, какое значение было получено в результате вызова методов `start()` и `addPage()`. Следует отметить, что в данном случае явное удаление с помощью оператора `delete` является излишним, поскольку переменная `prt`, хранящая ссылку на объект `PrintJob`, была объявлена как локальная и, соответственно, будет автоматически удалена после выполнения инструкций обработчика `onRelease`. Тем не менее если удаление объекта `PrintJob` не выполняется автоматически (например, если бы `prt` была объявлена как нело-

кальная), эту процедуру нужно выполнить явно, чтобы очистить спулер печати и получить возможность создания нового объекта PrintJob, например, для следующей печати.

В приведенном примере никак не учитывалось то, что размеры печатной области листа (которые задаются пользователем в настройках печати) и размеры печатаемой области могут соотноситься совершенно по-разному. В качестве одного из вариантов решения этой проблемы можно было бы, получив сведения о размерах печатной области листа, выполнить в фильме пропорциональное масштабирование содержимого, предназначенного для вывода на печать так, чтобы оно занимало на листе максимально возможную площадь. При выполнении этой операции, особенно при увеличении масштаба, необходимо учитывать характер выводимого на печать изображения. Вряд ли стоит увеличивать до размера печатной области формата A4 небольшое растровое изображение, поскольку, скорее всего, его качество будет неудовлетворительным. Однако при распечатке детализированного векторного персонажа вполне можно его увеличить. Информация о размерах печатной области, заданных пользователем (или используемых принтером по умолчанию), становится доступной после успешного вызова метода start() и содержится в свойствах pageWidth и pageHeight объекта PrintJob. Для получения требуемого результата необходимо после вызова метода start() (при условии, что метод возвращает значение true) выполнить пропорциональное масштабирование содержимого, выводимого на печать в соответствии с размерами печатной области, после чего вызвать метод addPage(), который передаст спулеру печати страницу с отмасштабированным изображением. После отправки изображения на печать нужно восстановить его исходный масштаб.

В приведенном ниже листинге представлена функция matchContent(), выполняющая пропорциональное масштабирование объекта так, чтобы он при выводе на печать занимал максимально возможную площадь печатной области. Данная функция принимает три параметра:

- pageW — ширина печатной области;
- pageH — высота печатной области;
- contentObj — ссылка на объект (клип или уровень стека документов), выводимый на печать.

На сопровождающем компакт-диске данная функция находится в папке Глава 23, в файле с именем matchContent.as.

Функция matchContent(). Подгонка размеров клипа под размеры области печати

```
/* Функция, выполняющая пропорциональное масштабирование объекта
contentObj так, чтобы он занимал максимально возможную площадь области
печати, ширина и высота которой задается параметрами pageW и pageH
соответственно */
function matchContent(pageW:Number, pageH:Number, contentObj:Object){
        /* Находим процентное выражение размеров области печати от
        размеров клипа */
        var ratioW = pageW / contentObj._width * 100;
        var ratioH = pageH / contentObj._height * 100;
```

```
        /* Если и ширина, и высота клипа больше или меньше ширины и высоты
        области печати */
        if ((ratioW<100 && ratioH<100) || (ratioW>100 && ratioH>100)) {
                /* Если отношение ширин меньше отношения высот, подгоняем
                размер клипа по ширине */
                if (Math.min(ratioW, ratioH) == ratioW) {
                        contentObj._xscale = contentObj._yscale = ratioW;
                }
                /* Если отношение ширин больше отношения высот, подгоняем
                размер клипа по высоте */
                else {
                contentObj._xscale = contentObj._yscale = ratioH;
                }
        }
        // Если клип уже и вьше области печати, подгоняем по высоте
        else if (ratioW >= 100 && ratioH <= 100) {
                contentObj._xscale = contentObj._yscale = ratioH;
        }
        // Если клип шире и ниже области печати, подгоняем по ширине
        else if (ratioW < 100 && ratioH > 100) {
        contentObj._xscale = contentObj._yscale = ratioW;
        }
}
```

В листинге 23.2 приводится сценарий, выводящий на печать содержимое второго кадра клипа `animClip`, пропорционально масштабируя изображение на листе так, чтобы оно занимало максимальную часть печатной области. В данном примере предполагается, что до выполнения печати печатаемый клип не масштабировался, то есть его масштаб составляет 100%.

Листинг 23.2. Печать заданного кадра клипа с применением пропорционального масштабирования до размера печатной области листа

```
#include "matchContent.as"
// При отпускании кнопки printBut...
printBut.onRelease = function() {
        // Создаем объект PrintJob
        var prt:PrintJob = new PrintJob();
        // Если пользователь нажал OK в окне настроек печати...
        if (prt.start()) {
```

```
            /* Пропорционально масштабируем клип так, чтобы он занимал
            максимальную часть области печати */
            matchContent(prt.pageWidth, prt.pageHeight, animClip);
            // Если страница отправлена спулеру печати...
            if (prt.addPage(animClip, null, null, 2)) {
                  // Отправляем страницу на печать
                  prt.send();
            }
      }
      // Возвращаем клипу исходные размеры
      animClip._xscale = animClip._yscale = 100;
      // Запускаем анимацию клипа
      animClip.play();
      // Удаляем объект PrintJob
      delete prt;
};
```

Иногда пользователь хочет распечатать не один кадр анимации, а сразу несколько или даже все кадры анимации. Например, при создании слайд-шоу можно предоставить пользователю возможность распечатки всех кадров монтажной линейки, содержащих разные изображения.

В листинге 23.3 приведен пример сценария, выводящего на печать все кадры монтажной линейки клипа `animClip`. В основе его действия лежит та же модель, что и в предыдущих примерах. Особенность состоит в том, что вызов метода `addPage()` осуществляется в цикле, количество итераций которого соответствует числу кадров на монтажной линейке клипа `animClip`. Если при очередном вызове метода `addPage()` возвращаемым значением является `true`, выполняется инкрементирование переменной `pageCount`, исполняющей роль счетчика страниц, успешно отправленных спулеру печати. Как уже обсуждалось, вызов метода `send()` следует применять только в том случае, если хотя бы один из предшествующих вызовов этого метода возвратил значение `true`. В нашем случае индикатором успешности передачи страниц спулеру печати является переменная `pageCount`. Если после всех вызовов метода `addPage()` ее значение оказывается отлично от нуля, значит, хотя бы одна страница была поставлена в очередь на печать и, следовательно, имеет смысл вызывать метод `send()`.

Листинг 23.3. Печать всех кадров монтажной линейки клипа

```
// При отпускании кнопки printBut...
printBut.onRelease = function() {
      // Создаем объект PrintJob
      var prt:PrintJob = new PrintJob();
```

```
        // Если пользователь нажал OK в окне настроек печати...
        if (prt.start()) {
                /* Переменная, хранящая количество страниц, успешно
                отправленных спулеру методом addPage() */
                var pageCount:Number = 0;
                /* Отправляем спулеру печати изображение каждого кадра
                клипа */
                for (var i = 1; i <= animClip._totalframes; i++) {
                        if (prt.addPage(animClip,{xMin:0,
                        xMax:animClip._width, yMin:0,
                        yMax:animClip._height}, null, i)) {
                        /* Если очередная страница успешно отправлена,
                        увеличиваем значение счетчика страниц */
                        pageCount++;
                        }
                }
                /* Если хотя бы одна страница успешно передана спулеру,
                отправляем все его содержимое на печать */
                if (pageCount) {
                        prt.send();
                }
        }
        // Запускаем анимацию клипа
        animClip.play();
        /* Явно удалять объект PrintJob в данном случае не требуется,
        поскольку ссылка на него хранится в локальной переменной, и он
        автоматически удаляется сборщиком мусора после выполнения кода
        обработчика onRelease() */
};
```

Пример, представленный в листинге 23.4, совмещает в себе рассмотренные ранее возможности и позволяет пользователю распечатать один кадр или все кадры монтажной линейки любого клипа, расположенного на текущей линейке. Возможность печати реализуется с помощью контекстного меню, в которое добавлены два пользовательских пункта **"Напечатать текущий кадр"** и **"Напечатать все кадры"**. Печать содержимого выполняется с учетом размеров печатной области листа, как это было показано в листинге 23.2.

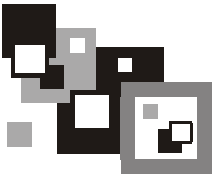
Листинг 23.4. Печать с помощью пользовательских элементов контекстного меню

```
#include "matchContent.as"
var cm:ContextMenu = new ContextMenu();
```

```
cm.hideBuiltInItems();
// Создаем элемент меню для вывода на печать всех кадров
var printAllItem:ContextMenuItem = new ContextMenuItem();
printAllItem.caption = "Напечатать все кадры";
// Выполнение печати всех кадров
printAllItem.onSelect = function(obj) {
       var prt:PrintJob = new PrintJob();
       if (prt.start()) {
       matchContent(prt.pageWidth, prt.pageHeight, obj);
              var pageCount:Number = 0;
       for (var i = 1; i <= obj._totalframes; i++) {
              if (prt.addPage(obj, {xMin:0, xMax:obj._width, yMin:0,↳
              yMax:obj._height}, null, i)) {
              pageCount++;
              }
       }
       if (pageCount) {
              prt.send();
       }
              obj._xscale = obj._yscale = 100;
       }
       obj.play();
};
// Создаем элемент меню для вывода на печать текущего кадра
var printCurrentItem:ContextMenuItem = new ContextMenuItem();
printCurrentItem.caption = "Напечатать текущий кадр";
// Выполнение печати текущего кадра
printCurrentItem.onSelect = function(obj) {
       var prt:PrintJob = new PrintJob();
       if (prt.start()) {
              matchContent(prt.pageWidth, prt.pageHeight, obj);
              if (prt.addPage(obj, {xMin:0, xMax:obj._width, yMin:0,↳
              yMax:obj._height})) {
                     prt.send();
              }
              obj._xscale = obj._yscale = 100;
       }
```

```
        obj.play();
};
// Добавляем пользовательские элементы в контекстное меню
cm.customItems.push(printCurrentItem);
cm.customItems.push(printAllItem);
// Ассоциируем меню со всеми клипами текущей линейки
for (i in this) {
        if (this[i] instanceof MovieClip) {
                this[i].menu = cm;
        }
}
```

Глава 24



Динамическая загрузка и воспроизведение видео

Неважно, что что-то идет неправильно.
Возможно, это хорошо выглядит...

Первый закон Скотта

Традиционно видео в Интернете представлялось скорее экзотикой, чем обычным явлением. Это было в первую очередь обусловлено ограничениями на скорость передачи информации, накладываемыми существующими линиями связи. В настоящее время благодаря развитию интернет-технологий значительно увеличилось количество пользователей, обладающих скоростными линиями доступа к глобальной сети. В связи с этим видео стало достаточно часто использоваться в Сети как мощное средство рекламы и взаимодействия с аудиторией.

Применение видео в сочетании с Flash-технологией имеет огромное преимущество, которое заключается в том, что видео воспроизводится в том же окне проигрывателя, что и Flash-фильм, и является неотделимым от него. Это позволяет очень тесно интегрировать видео в Web-среду, сделав его составляющим элементом SWF-документа. Flash MX 2004 предоставляет три различных варианта интеграции видео в SWF-документ:

- внедрение видеофайла;
- динамическая загрузка видеофайла в формате FLV (Flash Video);
- потоковая загрузка и трансляция видео в формате FLV.

Возможность внедрения видеофайлов появилась еще в Flash 6. В результате внедрения видеофайла он становится частью SWF-документа. Характерной особенностью внедренного видео является жесткая привязка к монтажной линейке. Внедренный видеофайл занимает диапазон кадров, длительность которого соответствует его продолжительности. Принципиально это очень похоже на использование анимированного символа типа Graphic. Основным преимуществом внедренного видео является возможность его тесной интеграции в среду Flash-документа. С внедренным видео можно работать, используя инструментарий векторного редактора и монтажной линейки. Можно выполнять операции масштабирования, поворота, скоса, использовать эффекты масок и даже анимацию. Поскольку внедренное видео привязано к монтажной линейке, можно очень легко создавать элементы управления его воспро-

изведением. Видео можно поместить на монтажную линейку клипа и управлять им, используя все возможности контроля клипов. Тем не менее у внедренного видео имеется ряд очень существенных недостатков.

- Внедрение видео существенно сказывается на объеме конечного SWF-документа.
- При загрузке SWF-документа по сети внедренный видеофайл начинает воспроизводиться только после окончания своей загрузки.
- При воспроизведении SWF-документа внедренное видео должно быть полностью загружено в память компьютера пользователя.
- При воспроизведении внедренного видео в течение времени, превышающего 120 секунд, возможно нарушение синхронизации звука и видео.
- Скорость воспроизведения внедренного видео всегда совпадает со скоростью воспроизведения, используемой в документе.
- Имеет место ограничение на длительность видео, которая не может превышать 16 000 кадров.
- При необходимости обновления видео приходится редактировать содержащий его SWF-документ.

С выходом Flash MX 2004 у разработчиков появилась возможность осуществлять динамическую загрузку (progressive download) видео в формате FLV (Flash Video) в SWF-документ с помощью ActionScript. Для этого в SWF-документе создается контейнер, в который уже непосредственно в процессе воспроизведения документа загружается внешний FLV-файл. Программный инструментарий ActionScript позволяет осуществлять контроль состояния загрузки видеофайла, управление его воспроизведением и параметрами звука. К преимуществам динамической загрузки видео можно отнести следующие.

- Объем SWF-документа определяется только его содержимым и никак не зависит от объема видеофайла, который будет в него загружаться.
- Воспроизведение динамически загружаемого видеофайла начинается, как только на диск будет записана порция информации, необходимая для начала воспроизведения.
- Динамически загруженный FLV-файл загружается, кэшируется и затем воспроизводится с локального диска пользователя. Поэтому для запуска воспроизведения не требуется, чтобы весь видеофайл целиком был загружен в память компьютера.
- Качество динамически загруженного видео выше, чем внедренного. Это объясняется тем, что кодек, осуществляющий импорт видео в Flash, обеспечивает более низкое качество по сравнению с кодеком, используемым при работе с динамически загруженным видео.
- Отсутствуют проблемы синхронизации звука и видео.
- Не существует ограничений на длительность видеофайла.
- В видеофайле может использоваться скорость воспроизведения, отличная от скорости, используемой в Flash-документе.

В этой главе мы подробно рассмотрим возможности, позволяющие реализовать динамическую загрузку видео и контроль его воспроизведения. В конце главы будет приведен пример создания проигрывателя файлов в формате Flash Video.

Наряду с внедрением и динамической загрузкой видео Flash поддерживает возможности работы с потоковым видео в формате FLV. Для этих целей в сочетании со средой разработки Flash используется самостоятельное приложение, которое называется Flash Communication Server. При установке Flash Communication Server MX в рабочей среде Flash MX 2004 появляется набор дополнительных инструментов, включающих специальные инструкции ActionScript, служебные панели и компоненты, позволяющие осуществлять обмен потоковым видео и аудио с сервером, поддерживающим Flash Communication Server. При использовании этой технологии видео- и/или аудиопоток отправляется клиенту посредством соединения, установленного между его компьютером и видеосервером. При этом имеется возможность детального контроля характеристик соединения и потока данных, позволяющего достаточно гибко управлять параметрами видео, чтобы оптимальным образом приспособить его под текущую скорость соединения. Одним из преимуществ использования потокового видео и аудио является то, что поток данных, передаваемых с помощью Flash Communication Server, воспроизводится на компьютере пользователя и сразу же удаляется из памяти, при этом кэширования данных не происходит. Flash Communication Server позволяет обеспечить передачу потокового видео, записанного с помощью Web-камеры или DV-камеры, с одного компьютера одновременно нескольким другим пользователям. Это находит применение при создании видеоконференций в реальном времени или развлекательных порталов. Описание Flash Communication Server и возможностей передачи потокового видео выходит за рамки этой книги. Для изучения этих вопросов читатель может обратиться к официальной документации Macromedia и ресурсам Интернета. В табл. 24.1 приведены сравнительные характеристики внедрения и динамической загрузки видео.

Таблица 24.1. *Сравнительные характеристики внедрения и динамической загрузки видео*

Характеристика	Внедренное видео	Динамическая загрузка видео
Поддерживаемые форматы (Flash MX 2004)	Audio Video Interleaved (AVI) Digital video (DV) Motion Picture Experts Group (MPEG) QuickTime movie (MOV) Windows Media file (WMV) Flash Video (FLV)	Flash Video (FLV)
Кодирование	Кодируется при импорте в Flash с применением кодека Sorenson Spark. При импорте FLV перекодировка не требуется	Кодирование выполняется при экспорте из внешних приложений с применением плагина Flash Video Exporter. Кодирование также может быть выполнено при экспорте видео из Flash

Таблица 24.1 *(окончание)*

Характеристика	Внедренное видео	Динамическая загрузка видео
Размер файла	Видеофайл со звуковым сопровождением входит в состав SWF-документа, увеличивая его объем	Видео располагается отдельно от SWF-документа
Привязка к монтажной линейке	Видеофайл жестко привязан к монтажной линейке и воспроизводится только на протяжении диапазона кадров, в которых он находится	Нет привязки к монтажной линейке
Скорость воспроизведения	Совпадает со скоростью воспроизведения, используемой в Flash-документе	Может отличаться от скорости воспроизведения, используемой в Flash-документе
Управление воспроизведением	Управление воспроизведением реализуется с помощью управления монтажной линейкой	Управление воспроизведением осуществляется с помощью класса `NetStream`
Загрузка из Интернета	При воспроизведении SWF-документа внедренное видео должно быть полностью загружено в память компьютера пользователя для начала воспроизведения	Для начала воспроизведения не требуется, чтобы весь видеофайл целиком был загружен в память компьютера
Использование	Короткие видеофайлы небольшого размера с низкой скоростью воспроизведения	Более продолжительные видеофайлы большего размера с произвольной скоростью воспроизведения
Совместимость	Flash Player 6 и выше	Flash Player 7

Формат *Flash Video*. Конвертирование видео в формат FLV

Формат Flash Video (FLV) был впервые предложен компанией Macromedia в 2002 году как средство обмена потоковым видео с использованием Macromedia Flash Communication Server. С выходом Flash MX 2004 появилась возможность, позволяющая непосредственно выполнять динамическую загрузку видео в SWF-документ и управлять его воспроизведением с помощью сценариев ActionScript, размещенных в документе.

Flash-видео кодируется как последовательность ключевых кадров (keyframes), хранящих полноразмерное несжатое изображение кадра видео. Ключевые кадры создаются на основе исходных кадров видео с определенным интервалом. Промежуточные кадры не сохраняются целиком, вместо этого кодер записывает только информацию, описывающую, как нужно изменить пикселы изображения, хранящегося в предыдущем ключевом кадре, для того чтобы получить изображение следующего ключевого кадра. При воспроизведении FLV-файла декодер восстанавлива-

ет промежуточные кадры, ориентируясь на содержимое ключевых. За счет этого удается добиться снижения объема файла. Очевидно, что длительность интервалов между ключевыми кадрами существенно отражается на качестве видео и на его объеме. Короткие интервалы позволяют получить видео более высокого качества, но при этом приводят к увеличению его объема, поскольку требуют сохранять в файле больше информации. Это может оказаться необходимым при работе с очень динамичным видео, содержащим большое количество часто изменяющихся изображений (например, запись гонок Формулы 1). Напротив, при работе с видео, состоящим из похожих изображений с небольшим количеством изменений, можно увеличить интервалы между ключевыми кадрами, уменьшая тем самым объем файла без существенного ухудшения качества. Частота расстановки ключевых кадров играет важную роль при организации возможностей поиска и пошаговой перемотки видео, поскольку в FLV-файле переход выполняется только от одного ключевого кадра к другому. FLV-файл может включать по одному аудио- и видеопотоку, синхронизированных друг с другом. Сжатие, используемое форматом FLV, основывается на стандарте H.263, принятом в мае 1996 года и получившем название H.263v1. При его реализации в Flash были сделаны некоторые отступления от данного стандарта за счет удаления определенных характеристик и добавления новых. Более подробную техническую информацию о формате FLV читатель может получить по адресу **www.macromedia.com/software/flash/open/licensing/fileformat**.

Для того чтобы конвертировать имеющийся видеофайл в формат FLV, можно воспользоваться следующими средствами.

- Экспортировать импортированный в рабочую среду Flash MX 2004 видеофайл с помощью команды библиотеки документа.
- Экспортировать видеофайл из приложения видеообработки, поддерживающего плагин Flash Video Exporter.
- Воспользоваться пакетом Flash Video Kit, в состав которого входит Sorenson Squeeze, позволяющий конвертировать большинство наиболее распространенных форматов видео в формат FLV.

Для того чтобы конвертировать имеющийся видеофайл в формат FLV, используя только Flash MX 2004, необходимо вначале импортировать его в библиотеку документа. Это можно сделать с помощью команды главного меню **File>Import>Import to Library...**. В процессе импорта можно задать все необходимые параметры, на основании которых будет выполнено кодирование импортируемого файла. После того как файл оказался в библиотеке документа, необходимо выделить его и в контекстном меню библиотеки выполнить команду **Properties...**. В появившемся при этом диалоговом окне **Embedded Video Properties** нужно нажать кнопку **Export...**.

Для того чтобы экспортировать видеофайл в формат FLV из специализированного приложения видеообработки, необходимо предварительно установить плагин, который называется Flash Video Exporter. Этот плагин входит в комплект поставки Flash MX 2004 Professional, однако он требует отдельной установки[1]. После установки

[1] При использовании пробной (trial) или нелицензионной версии Flash MX 2004 плагин Flash Video Exporter будет работать в демонстрационном режиме. Он будет обладать всеми возможностями полнофункциональной версии, однако в созданном с его помощью FLV-файле изображение будет перечеркнуто двумя диагональными красными линиями. — *Прим. автора*.

Flash Video Exporter поддерживающие его приложения для видеомонтажа позволяют выполнить экспорт в формат FLV, используя стандартную команду **File>Export...**. Использование данного подхода на практике является значительно более удобным, чем экспорт видео из Flash, поскольку позволяет задать параметры кодировки аудио- и видеопотоков, не требует затрат времени на импорт видео в Flash и позволяет получить файл более высокого качества, и лучше приспособленный к своему целевому назначению. Ниже представлен перечень Windows-приложений, поддерживающих Flash Video Exporter:

- Adobe After Effects;
- Adobe Premier 6.5;
- Anystream Agility;
- Avid Media Composer;
- Avid Xpress DV, Avid Xpress Pro;
- Discreet Cleaner;
- Discreet Cleaner XL.

Для того чтобы экспортировать текущий видеофайл в формат FLV, необходимо выполнить команду главного меню **File>Export...>Quick Time**, в появившемся диалоговом окне выбрать формат **Macromedia Flash Video (FLV)** и нажать кнопку **Options...**. В результате будет выведено диалоговое окно с настройками параметров кодирования видео и звука, представленное на рис. 24.1.

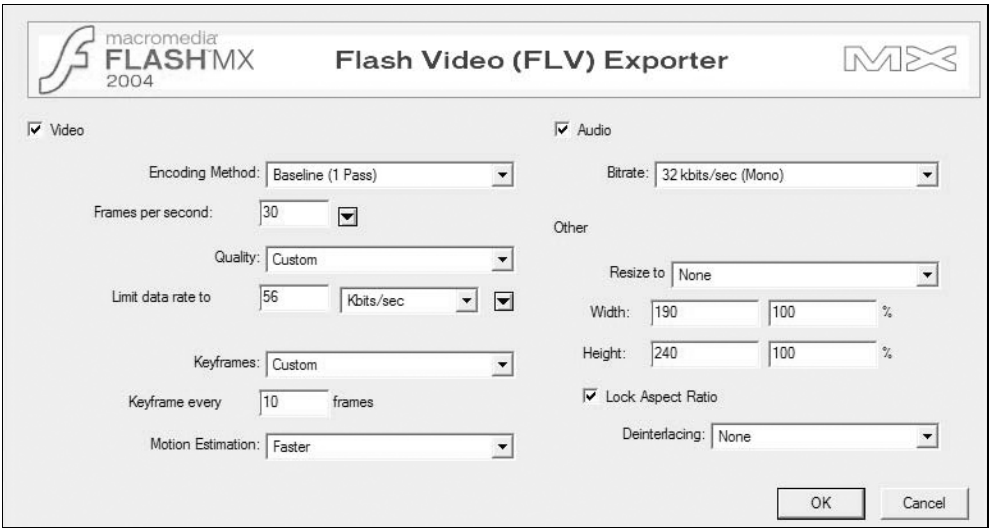


Рис. 24.1. Окно **Flash Video Exporter** с настройками параметров кодирования видео и звука при экспорте в формат FLV из внешних приложений обработки видео

FLV-файл может включать как видео и звук, так и только видео или только звуковое сопровождение. С помощью флажков **Video** и **Audio** можно указать, какая составляющая должна войти в состав результирующего FLV-файла. Для того чтобы в результирующий FLV-файл вошли и исходное видео, и оригинальная звуковая дорожка видеофайла, необходимо установить оба флажка **Video** и **Audio**. Краткое описание параметров экспорта приводится в табл. 24.2.

Таблица 24.2. *Параметры экспорта видеофайла в формат FLV с помощью плагина Flash Video Exporter*

Параметр	Описание
Encoding Method	Задает используемый метод кодирования. Этот параметр влияет на качество видео и конечный объем файла
Baseline (1 Pass)	Используется сжатие, соответствующее стандарту H263. Скорость передачи данных (bit rate) остается постоянной и не меняется от кадра к кадру вне зависимости от характера видеоматериала. Объем результирующего файла во многом определяется значением, введенным в поле **Limit data rate to**
Better (1 Pass VBR)	Используется сжатие, соответствующее стандарту H263. Кодирование выполняется с переменной скоростью передачи данных, учитывая характер видеоматериала. На участки, описание которых требует большего объема информации, выделяется больше битов. Этот метод кодирования позволяет получить изображение более высокого качества, но требует больших временных затрат и производит файл большего размера, значение которого достаточно трудно предугадать
Screen Recording Codec	При использовании данного режима все остальные параметры настройки для видео становятся недоступными. Используется сжатие без потерь (lossless). Кодек фиксирует все изменения, происходящие от кадра к кадру, и производит файлы небольшого размера в случае, если видео состоит из кадров, с минимальным количеством существенных отличий. Эта опция отлично подходит для кодирования видео, полученного в результате записи действий, происходящих на экране. При использовании этой опции для кодирования обычного видео размер конечного файла получается очень большим
Frames per second	Данный параметр позволяет задать скорость воспроизведения, используемую в результирующем FLV-файле. Для сохранения исходного качества следует задать скорость, используемую в оригинале (например, 29,97 fps для NTSC и 25 fps для PAL). Уменьшение скорости воспроизведения позволяет снизить количество кодируемой информации и таким образом уменьшить конечный объем файла. При задании более низкой скорости Flash Video Exporter через равные промежутки пропускает "лишние" кадры. Уменьшение скорости следует выполнять таким образом, чтобы новая скорость была в точности кратна исходной, в противном случае может произойти существенное ухудшение качества изображения. При существенном снижении скорости по сравнению со скоростью оригинала может стать заметным эффект ступенчатого, рваного воспроизведения

Таблица 24.2 *(продолжение)*

Параметр	Описание
Quality	Данный параметр позволяет указать приблизительную скорость передачи данных, используемую при кодировании видео с заданной градацией качества: **low**, **medium** и **high**. При выборе в выпадающем списке одного из этих значений в поле **Limit data rate to** будет выведена примерная скорость передачи данных, рассчитанная на основании заданных значений скорости воспроизведения (**Frames Per Second**) и размеров видео (поля **Width** и **Height**). Все три параметра: качество, скорость воспроизведения и размеры видео прямо пропорциональны скорости передачи данных, необходимой для передачи потокового видео. При выборе опции **Custom** можно задать требуемое значение скорости передачи данных в поле **Limit data rate to** для получения нужного качества
Limit data rate to	Данный параметр позволяет задать значение скорости передачи данных, влияющее на объем и качество конечного файла. В качестве единиц измерения могут быть выбраны Кбайт/с или Кбит/с. На основании значения, введенного в это поле, определяется верхний предел объема FLV-файла. Например, если длительность видео составляет 60 секунд, и в поле **Limit data rate to** задано значение 100 Кбайт/с, конечный объем файла не будет превышать 60 Мбайт. Ограничение объема реализуется в ущерб качеству. Для сохранения качества при заданном значении данного параметра можно либо уменьшить размеры изображения, либо снизить скорость воспроизведения
Keyframes	Данный выпадающий список позволяет установить режим автоматического или пользовательского определения длины интервала между ключевыми кадрами в результирующем FLV-видео. При выборе значения **Auto** Flash Video Exporter устанавливает частоту расстановки ключевых кадров кратной текущей скорости воспроизведения. Установка значения **Custom** позволяет задать произвольную частоту расстановки ключевых кадров. При этом требуемое значение может быть задано в поле **Keyframe every...frames**. Длина интервала между ключевыми кадрами не может превышать числа 100
Motion Estimation	Данный выпадающий список позволяет выбрать схему кодирования, отдающую приоритет качеству или скорости. При выборе значения **Faster** данные будут распределены так, чтобы обеспечить более высокую скорость кодирования за счет снижения качества. Значение **Better** позволяет получить более высокое качество, за счет снижения скорости кодирования
Audio Bitrate	Скорость передачи данных для аудиопотока, синхронизированного с видео
Resize to	Данный выпадающий список позволяет масштабировать видео, выбрав один из предложенных стандартных размеров. При необходимости можно задать произвольные размеры вручную, указав их в пикселах или в процентах в полях **Width** и **Height**. Для выполнения пропорционального масштабирования необходимо установить флажок **Lock Aspect Ratio**. Несмотря на то, что стандарт H263 допускает использование произвольных размеров кадра, некоторое преимущество дает использование размеров, кратных размеру 8×8 пикселов

Таблица 24.2 *(окончание)*

Параметр	Описание
Deinterlacing	Данный выпадающий список позволяет установить режим выполнения античересстрочного[1] преобразования, позволяющего удалить из кадра "лишнее" изображение, соответствующее полю, сформированному нечетными или четными строками. Если выполнение античересстрочного преобразования не требуется, в списке нужно выбрать значение **None**. При выборе значения **Upper** будет удалено нечетное поле кадра, эта опция применяется для видео, записанного в системе PAL. При выборе значения **Lower** будет удалено четное поле кадра, эта опция применяется для видео, записанного в системе NTSC

[1] Кадр телевизионного изображения состоит из горизонтальных строк, количество которых различается для разных стандартов (525 в NTSC, 625 в PAL). По строкам экрана с достаточно высокой частотой (60 раз в секунду в NTSC, 50 раз в секунду в PAL) проходит электронный луч. Каждую секунду электронный луч выполняет два прохода, сначала по нечетным строкам, потом по четным. После каждого прохода на экране создается поле. Этот процесс создания кадра называется черестрочной разверткой. В мониторе компьютера кадр формируется за один проход. При переводе видео в цифровую форму оба поля, составляющие изображение совмещаются в одном кадре. При воспроизведении видео, записанного в системах NTSC или PAL на компьютере, черестрочная развертка (совмещение изображений) может стать причиной появления артефактов и ухудшения качества, особенно при работе с динамичным видео. Причина этого заключается в том, что в одном кадре содержатся изображения, соответствующие двум моментам времени. — *Прим. автора*.

Механизм выполнения загрузки FLV-файлов

Для выполнения загрузки и воспроизведения внешних FLV-файлов необходимо задействовать возможности трех встроенных классов: `NetConnection`, `NetStream` и `Video`. В общих чертах эта процедура включает следующие этапы.

1. В рабочей среде на сцене необходимо разместить экземпляр видеообъекта, придав ему требуемые размеры. Видеообъект выполняет роль контейнера, в котором в процессе воспроизведения Flash-документа будет отображаться видео, загружаемое из внешнего FLV-файла.
2. Выполняется создание объекта `NetConnection` и осуществляется вызов метода `connect()`, открывающего локальное соединение, с помощью которого будет выполняться загрузка внешнего FLV-файла.
3. Выполняется создание объекта `NetStream`, который используется для контроля процесса загрузки и управления воспроизведением FLV-файла.
4. С помощью метода `attachVideo()` класса `Video` видеопоток, представляемый объектом `NetStream`, ассоциируется с видеообъектом, созданным на шаге 1. Эта операция необходима для того, чтобы загружаемое из внешнего файла видео отображалась в области видеообъекта, представляющего собой своеобразный экран.

5. Для того чтобы начать воспроизведение внешнего FLV-файла, необходимо вызвать метод `play()` объекта `NetStream`, передав ему в качестве параметра URL, по которому расположен загружаемый файл.
6. Дальнейшее управление воспроизведением и контроль состояния загрузки внешнего FLV-файла осуществляются с помощью возможностей класса `NetStream`.

Создание объекта *Video*. Методы и свойства видеообъекта

Класс `Video` используется для обеспечения возможности воспроизведения потокового видео, полученного с Web-камеры или загруженного из внешнего FLV-файла в SWF-документе. Экземпляр класса `Video` или видеообъект играет роль контейнера, в котором отображается загруженное в документ видео. Видеообъект представляет собой своеобразный экран, определяющий физические размеры и местоположение видео. При выводе видеофайла в видеообъект изображение принимает его размеры и наследует все примененные к нему трансформации (поворот, скос, отражение). Если пропорции видеообъекта отличны от пропорций видео, изображение будет масштабироваться с изменением исходных пропорций. Подобно объектам `MovieClip` и `Button`, объект `Video` создается не с помощью конструктора, а непосредственно в рабочей среде. Так, например, внедренное в документ видео представляет собой видеообъект, являющийся экземпляром класса `Video`. В принципе, его можно использовать для вывода видео, загруженного из внешнего видеофайла, в формате FLV. При этом исходное (внедренное) видео будет заменено загружаемым видеопотоком. Однако на практике такой подход не используется.

Чтобы создать на сцене видеообъект, предназначенный для вывода динамически загруженного видео, нужно создать пустой видеообъект и снабдить его идентификатором (имя экземпляра). Для этого необходимо выполнить следующую последовательность действий.

1. Открыть окно библиотеки и в контекстном меню выполнить команду **New Video...**. В результате в библиотеке появится новый пустой видеообъект.
2. Вытащить экземпляр видеообъекта на сцену и задать ему требуемые размеры и местоположение.
3. В панели **Properties** задать имя экземпляра в поле **Instance Name**.

В процессе воспроизведения фильма видеообъектом можно управлять, используя все базовые свойства класса `MovieClip`: `_x`, `_y`, `_width`, `_height`, `_xscale`, `_yscale`, `_rotation`, `_alpha`, `_visible`. Следует иметь в виду, что точка регистрации видеообъекта расположена в его левом верхнем углу. Для того чтобы программно выполнить перемещение или трансформацию видеообъекта с использованием другого местоположения точки регистрации, его можно поместить на монтажную линейку клипа и трансформировать этот клип.

Для того чтобы связать видеообъект с конкретным видео, которое должно выводиться в занимаемой им области, необходимо воспользоваться методом `attachVideo()` класса `Video`. Данный метод указывает Flash, в каком видеообъекте следует отображать видеопоток, полученный из внешнего файла или с Web-камеры. При загрузке

внешнего FLV-файла метод attachVideo() следует вызывать с применением следующего синтаксиса:

видеообъект.attachVideo(*объект NetStream*)

Здесь *видеообъект* — это объект Video, в области которого будет отображаться загруженный видеофайл.

Параметр *объект NetStream* — это ссылка на объект NetStream, управляющий процессом загрузки и воспроизведения FLV-файла.

Объект Video обладает свойствами width и height, которые доступны только для чтения и позволяют получить значения высоты и ширины видео, которое выводится в области видеообъекта. Следует четко понимать различия между наследуемыми свойствами видеообъекта _width и _height и свойствами width, height объекта Video. Свойства _width и _height определяют размеры видеообъекта, то есть контейнера, в котором отображается видео. Изменение значений этих свойств приводит к изменению масштаба видеообъекта и, соответственно, выводимого в нем изображения. Свойства возвращают исходные размеры видео, которые могут не совпадать с размерами видеообъекта. Эти свойства можно использовать, например, для того, чтобы выполнить масштабирование видеообъекта в соответствии с пропорциями выводимого в нем видеофайла. Отметим, что при загрузке внешнего FLV-файла свойства width и height принимают значения только после вывода изображения в видеообъект. До этого момента значения свойств остаются равными нулю. Для того чтобы установить момент появления изображения в области видеообъекта, можно воспользоваться обработчиком события onStatus класса NetStream.

Для того чтобы очистить область видеообъекта от выводимого в текущий момент изображения, не скрывая при этом сам видеообъект, можно воспользоваться методом clear(). Данный метод очищает область видеообъекта, однако его действие распространяется только на изображение, выводимое в момент его вызова (например, если видео поставлено на паузу). Если после вызова этого метода текущая позиция воспроизведения изменяется, изображение вновь будет выводиться в области видеообъекта.

При масштабировании видео можно использовать функцию "сглаживание". Сглаживание позволяет уменьшить резкость границ пикселов изображения, но требует значительных ресурсов процессора. Для управления сглаживанием используется свойство smoothing, принимающее булево значение. Если данное свойство принимает значение true, сглаживание включено, в противном случае сглаживание отключается. Сглаживание видео выполняется только при использовании режима **High Quality** проигрывателя. По умолчанию данное свойство принимает значение false.

Открытие локального соединения

Для получения возможности воспроизведения внешнего FLV-файла необходимо открыть локальное соединение, через которое будет выполняться передача из Интернета или из локальной файловой системы. Соединение осуществляется с помощью объекта NetConnection, который создается с помощью оператора new и конструктора.

```
var nc:NetConnection = new NetConnection()
```

Здесь nc — это переменная, в которой сохраняется ссылка на объект NetConnection.

Для того чтобы открыть соединение, необходимо вызвать метод connect() объекта NetConnection, передав ему в качестве параметра значение null.

```
nc.connect(null);
```

Параметр метода является обязательным и его значение должно быть именно таким. Дело в том, что объект NetConnection также может быть использован для установления удаленного соединения и непосредственного вызова удаленных процедур с сервера, на котором установлена служба Flash Remoting. В этом случае при установке удаленного соединения методу connect() в качестве параметра передается строка, определяющая шлюз службы Flash Remoting. Поскольку для выполнения загрузки внешнего FLV-файла используется локальное соединение, методу connect() в качестве параметра передается null.

Контроль загрузки и воспроизведения FLV-видео. Класс *NetStream*

Класс NetStream используется для осуществления загрузки и управления воспроизведением FLV-видео. Объект NetStream создается традиционным образом с помощью оператора new и конструктора:

```
var nStream:NetStream = new NetStream(объект NetConnection)
```

Здесь nStream — это переменная, в которой сохраняется ссылка на объект NetStream.

Параметр *объект NetConnection* представляет собой ссылку на объект, открывший локальное соединение, с помощью которого осуществляется загрузка внешнего FLV-файла. Данный параметр является обязательным.

Инициирование выполнения загрузки

Для запуска загрузки FLV-файла используется метод play(), который имеет следующий синтаксис:

```
nStream.play(url)
```

Здесь *nStream* — это объект NetStream, управляющий загрузкой и воспроизведением FLV-файла.

Параметр *url* — это строка, указывающая местоположение FLV-файла. Путь может быть задан с использованием абсолютной или относительной адресации. Относительный путь указывается относительно фактического местоположения SWF-документа.

В результате вызова метода play() начинает выполняться загрузка FLV-файла. Его воспроизведение начинается автоматически сразу же, как только будет загружено количество информации, соответствующее заданному размеру буфера. По умолча-

нию размер буфера составляет 0,1 секунды. Это означает, что воспроизведение будет выполняться только после того, как в память компьютера будет загружена 0,1 секунды видео (и/или звука). Размер буфера может быть задан произвольно с помощью метода `setBufferTime()`.

Для того чтобы загружаемое видео отображалось на экране, необходимо выполнить вызов рассмотренного выше метода `attachVideo()`, позволяющего ассоциировать поток загружаемых данных с объектом `Video`, исполняющим роль контейнера. В противном случае зритель будет слышать звуковое сопровождение, но видео отображаться не будет. Если FLV-файл включает только звук, вызывать метод `attachVideo()` не требуется.

Ниже приводится пример кода, выполняющего загрузку и запуск воспроизведения видеофайла с именем movie.flv, расположенного в том же каталоге, что и SWF-файл.

```
var nc:NetConnection = new NetConnection();
nc.connect(null);
var nStream:NetStream = new NetStream(nc);
videoObj.attachVideo(nStream);
nStream.play("movie.flv");
```

Контроль процесса загрузки

В классе `NetStream` имеются свойства `bytesLoaded` и `bytesTotal`, позволяющие контролировать состояние процесса загрузки FLV-файла в проигрыватель. Свойство `bytesLoaded` возвращает количество байт видеофайла, загруженных на момент обращения к нему. Свойство `bytesTotal` возвращает общий объем видеофайла. Оба свойства используются только для чтения.

Ниже представлен сценарий, отслеживающий состояние загрузки FLV-файла и отражающий это с помощью стандартной полосы состояния и вывода процента загруженной информации текстового поля. Полоса состояния представлена клипом `loadStatusBar`, а текстовое поле имеет имя `statusTxt`.

```
var nc:NetConnection = new NetConnection();
nc.connect(null);
var nStream:NetStream = new NetStream(nc);
videoObj.attachVideo(nStream);
nStream.play("video/movie.flv");
loadStatusBar._xscale = 0;
function checkLoadStatus() {
        var loaded = nStream.bytesLoaded;
        var total = nStream.bytesTotal;
        var streamStatus = Math.round(loaded / total * 100);
        statusTxt.text = streamStatus + "%";
        loadStatusBar._xscale = streamStatus;
```

```
        if (streamStatus >= 100) {
                clearInterval(loadInt);
        }
}
var loadInt:Number = setInterval(checkLoadStatus, 300);
```

Необходимо отметить, что отследить ход загрузки можно только при реальной загрузке из сети. Если FLV-файл размещен локально в файловой системе пользователя, значение свойства `bytesLoaded` совпадает со значением свойства `bytesTotal`. Это также имеет место и в среде тестирования даже при использовании режима имитации загрузки **Simulate Download**.

Установка размера буфера

При загрузке видеофайла данные буферизуются для того, чтобы уменьшить количество вынужденных остановок воспроизведения, вызванных недостаточной скоростью передачи данных. Буферизация приводит к тому, что файл начинает воспроизводиться только после заполнения буфера. Так, например, если размер буфера равен 20 секундам, воспроизведение начнется, как только в память будут загружены первые 20 секунд видео. Это гарантирует непрерывное воспроизведение в течение 20 секунд. Если буфер оказывается пуст (из-за недостаточной скорости передачи данных), воспроизведение прекращается до тех пор, пока он снова не заполнится.

По умолчанию размер буфера составляет 0,1 секунды. С помощью метода `setBufferTime()` можно задать произвольный размер буфера. Данный метод имеет следующий синтаксис:

nStream`.setBufferTime(`*секунды*`)`

Параметр *секунды* — это число, задающее размер буфера в секундах. Следует иметь в виду, что размер буфера не должен превышать продолжительности видео, в противном случае изображение будет отсутствовать.

Для того чтобы определить размер буфера, заданный с помощью метода `setBufferTime()`, можно воспользоваться доступным только для чтения свойством `bufferTime`.

Для того чтобы определить продолжительность видео (звука), находящегося в буфере в данный момент, можно воспользоваться свойством `bufferlength`. Это свойство доступно только для чтения и возвращает значение, измеряемое в секундах. Следует отметить, что по мере приближения видео к концу количество данных, хранящихся в буфере, уменьшается, начиная с того момента, когда оставшийся фрагмент видео имеет меньшую продолжительность, чем длительность, заданная для буфера.

Используя свойства `bufferTime` и `bufferlength`, можно выводить пользователю информацию о состоянии буфера. Подобный пример входит в состав кода проигрывателя, представленного в конце данной главы.

Управление воспроизведением

Вызов метода `play()` приводит к началу процесса загрузки и автоматически запускает воспроизведение FLV-файла. Для последующих остановки и продолжения воспроизведения видео и звука используется метод `pause()`. Данный метод имеет два варианта использования. При передаче методу параметра он в зависимости от значения этого параметра останавливает или запускает остановленное ранее воспроизведение. Если метод `pause()` вызывается без параметра, он действует как переключатель. Если воспроизведение запущено, он останавливает воспроизведение. Если воспроизведение остановлено, он запускает его. В общем случае метод `pause()` имеет следующий синтаксис:

nStream.pause(*флаг*)

Параметр *флаг* принимает булево значение, определяющее направление действия метода. Если в качестве параметра передается значение `true`, метод останавливает воспроизведение на ближайшем ключевом кадре. Если в качестве параметра передается значение `false`, метод запускает воспроизведение.

Данный метод очень удобно использовать для создания элементов управления, выполняющих остановку (пауза) и запуск (воспроизведение) видео. Пример такого использования входит в состав кода проигрывателя, представленного в конце данной главы.

Для того чтобы определить текущее положение позиции воспроизведения, нужно воспользоваться свойством `time`. Данное свойство доступно только для чтения и хранит значение текущей позиции воспроизведения относительно начала видео в секундах. Код, приведенный в следующем примере, выводит в текстовое поле `seekStatusTxt` время, прошедшее с начала видеоролика в минутах и секундах.

```
var nc:NetConnection = new NetConnection();
nc.connect(null);
var nStream:NetStream = new NetStream(nc);
videoObj.attachVideo(nStream);
nStream.play("video/movie.flv");
function checkTimeStatus() {
        var curPosition:Number = nStream.time;
        var minutes = Math.floor(curPosition / 60);
        var seconds = Math.floor(curPosition % 60);
        if (seconds < 10) {
                seconds = "0" + seconds;
        }
        if (minutes < 10) {
                minutes = "0" + minutes;
        }
        seekStatusTxt.text = minutes + ":" + seconds;
}
setInterval(checkTimeStatus, 300);
```

При тестировании SWF-документа может оказаться полезной информация о количестве кадров видео, отображаемых на протяжении секунды. Эти данные можно использовать при установке параметров компрессии перед экспортом видеофайла в формат FLV. Для определения текущего количества кадров, отображаемых на протяжении секунды, используется свойство `currentFps`. Это свойство доступно только для чтения.

Ручной поиск, перемотка

Одной из базовых, привычных для всех пользователей функций проигрывателей видео является возможность перемещения к указанной временной позиции. Обычно в проигрывателях предусматривается возможность пошагового перемещения с фиксированным интервалом и перемещения, выполняемого с помощью ползунка вручную. Для реализации подобных возможностей при работе с динамически загружаемым видео используется метод `seek()`.

Метод `seek()` позволяет переместить текущую позицию воспроизведения к заданному в секундах от начала воспроизведения моменту времени. С помощью данного метода можно перемещаться как вперед, так и назад. Тем не менее его использование связано с некоторыми важными ограничениями.

- Метод `seek()` позволяет перемещаться только от одного ключевого кадра к другому. Соответственно, возможности поиска непосредственно зависят от количества ключевых кадров в видеофайле, заданных в параметрах настройки кодирования перед выполнением экспорта в формат FLV. Чем больше количество ключевых кадров, тем точнее можно осуществлять поиск нужного фрагмента в видеоролике.
- Файл FLV не предоставляет информации о своей продолжительности. Определить ее можно только после полного окончания загрузки файла. Однако, не зная длительности видео, невозможно создать качественный механизм поиска. Например, при реализации функциональности полосы ручного поиска ее ширина должна быть связана с продолжительностью видео, для того чтобы можно было верно рассчитать текущую позицию воспроизведения на основании положения бегунка, установленного пользователем. Единственное решение этой проблемы состоит в том, чтобы заранее задать продолжительность FLV-файла. Таким образом, для обеспечения возможности выполнения ручного поиска в видеоролике необходимо заранее знать его продолжительность.

Метод `seek()` имеет следующий синтаксис:

nStream.seek(*временная позиция*)

Параметр *временная позиция* — это неотрицательное число, указывающее, в какую временную позицию по отношению к началу следует переместить позицию воспроизведения. Временная позиция задается в секундах. Так, например, для того чтобы вернуть воспроизведение в начало, нужно использовать следующий код:

```
nStream.seek(0);
```

Следующий вызов перемещает позицию воспроизведения на 15-ю секунду от начала.

```
nStream.seek(15);
```

Следует иметь в виду, что значение, передаваемое параметром *временная позиция*, соблюдается приблизительно. Поскольку переход выполняется только по ключевым кадрам, метод `seek()` устанавливает позицию воспроизведения не точно на заданное значение времени, а на ближайший к нему ключевой кадр. Так, например, если имеется два одинаковых видеофайла, но с разными значениями частоты размещения ключевых кадров, то вызов метода `seek()` с одним и тем же параметром, скорее всего, приведет к тому, что после перехода зритель будет видеть разные изображения.

Если значение параметра *временная позиция* превышает продолжительность видео, переход не выполняется и воспроизведение останавливается на текущей позиции. Это обстоятельство необходимо иметь в виду при выполнении перемотки вперед.

Для того чтобы выполнить переход относительно текущей позиции воспроизведения, необходимо передать методу в качестве параметра выражение с использованием свойства `time`. Следующий код перемещает позицию воспроизведения на 10 секунд относительно ее текущего положения:

```
nStream.seek(nStream.time + 10);
```

Ниже представлен сценарий, выполняющий пошаговую перемотку вперед и назад с интервалом в 5 секунд при нажатии кнопок `fwBut` и `rwBut` соответственно.

```
var totalTime:Number = 120 // Продолжительность видео
fwBut.onRelease = function() {
              nStream.seek(nStream.time + 5);
};
rwBut.onRelease = function() {
       if (nStream.time >= 5) {
              nStream.seek(nStream.time - 5);
       }
       // Если до начала осталось менее 5 секунд, переходим в начало
       else {
              nStream.seek(0);
       }
};
```

Контроль звукового сопровождения

При работе с динамически загружаемым FLV-файлом имеется возможность управления его звуковым сопровождением. Это позволяет реализовать такие стандартные элементы, как регулятор громкости, регулятор баланса каналов и возможность отключения звукового сопровождения.

Для того чтобы обеспечить возможность управления звуковым сопровождением FLV-файла, его нужно связать с клипом с помощью метода `attachAudio()` класса `MovieClip`. После этого звуком в FLV-файле можно управлять, используя возмож-

ности класса Sound так же, как если бы этот звук был просто изначально прикреплен к клипу.

Для того чтобы связать звук из FLV-файла с клипом с помощью метода attachAudio(), при его вызове необходимо использовать следующий синтаксис:

клип.attachAudio(*объект NetStream*)

Здесь *клип* — это экземпляр клипа, к которому прикрепляется звук из FLV-файла. Этот клип может быть создан как в рабочей среде, так и динамически, например, с помощью метода CreateEmptyMovieClip().

Параметр *объект NetStream* — это ссылка на объект NetStream, осуществляющий загрузку и управление воспроизведением FLV-файла.

Следует отметить, что при управлении звуком из FLV-файла, прикрепленным к клипу, могут быть реализованы далеко не все возможности класса Sound. Так, например, нельзя определить продолжительность звука с помощью свойства duration, равно как и текущую позицию воспроизведения с помощью свойства position. Поскольку в FLV-файле звук точно синхронизирован с видеорядом, невозможно выполнить остановку и запуск звукового сопровождения с помощью методов Sound.stop() и Sound.play(). Обработчики событий класса Sound также не работают. Возможности управления звуковым сопровождением FLV-файла ограничиваются только манипуляциями с уровнем громкости (метод setVolume()) и балансом каналов (методы setPan() и setTransform()). Для того чтобы отключить звуковое сопровождение, достаточно просто понизить уровень звука до нуля.

В следующем сценарии демонстрируется пример создания стандартной кнопки-переключателя с именем toggleSoundBut, выполняющей отключение/включение звука в FLV-файле.

```
var nc:NetConnection = new NetConnection();
nc.connect(null);
var nStream:NetStream = new NetStream(nc);
videoObj.attachVideo(nStream);
// Сохраняем url, по которому расположен FLV-файл
var url:String = "movie.flv"
nStream.play(url);
// Создаем клип, с которым будем связывать звук
this.createEmptyMovieClip("soundmc", this.getNextHighestDepth());
// Создаем контролирующий звук объект Sound
var snd:Sound = new Sound("soundmc");
// Связываем звук с клипом soundmc
soundmc.attachAudio(nStream);
toggleSoundBut.onPress = function() {
       if (!soundOff) {
              snd.setVolume(0);
```

```
            soundOff = true;
        }
        else {
            snd.setVolume(100);
            soundOff = false;
        }
};
```

Обработка событий объекта *NetStream*

В классе NetStream имеется уже знакомый нам по классам SharedObject и LocalConnection обработчик события onStatus. Событие onStatus генерируется при обнаружении изменения в состоянии загрузки и воспроизведения или при возникновении ошибки, связанной с невозможностью выполнить загрузку. Для контроля наступления данного события необходимо определить соответствующий метод обработчика для объекта NetStream. При этом используется следующий синтаксис:

```
nStream.onStatus = function(информационный объект){
блок кода;
...
};
```

При вызове метода обработчика ему передается параметр *информационный объект*, содержащий два свойства code и level, принимающих строковые значения. Значения данных свойств указывают на причину возникновения события onStatus. Как правило, в теле метода обработчика события onStatus осуществляется проверка значений этих свойств, на основании которой выполняются те или иные действия. В табл. 24.3 представлены возможные значения свойств code и level для объекта NetStream и их описание.

Таблица 24.3. *Значения свойств* code *и* level *для объекта* NetStream

Значение свойства code	**Значение свойства** level	**Описание значения**
"NetStream.Buffer.Empty"	"status"	Буфер пуст. Это может произойти либо из-за того, что данные не успевают поступать с достаточной для его заполнения скоростью, либо если воспроизведение файла закончилось. При возникновении этого события воспроизведение прерывается и возобновляется только после того, как буфер вновь будет заполнен, после чего снова будет сгенерировано событие onStatus, при котором свойство code будет иметь значение "NetStream.Buffer.Empty"

Таблица 24.3 (окончание)

Значение свойства `code`	Значение свойства `level`	Описание значения
`"NetStream.Buffer.Full"`	`"status"`	Буфер заполнен. Всякий раз, когда буфер заполняется до значения, заданного методом `setBufferTime()`, генерируется событие `onStatus` и свойство `code` получает данное значение. В процессе воспроизведения файла данное событие может возникать неоднократно
`"NetStream.Play.Start"`	`"status"`	Запуск воспроизведения. Это событие генерируется при запуске загрузки и воспроизведения файла, инициированных успешным вызовом метода `play()`
`"NetStream.Play.Stop"`	`"status"`	Остановка воспроизведения. Это событие возникает при завершении или прерывании воспроизведения. Отметим, что вызов метода `pause()` не приводит к появлению данного события
`"NetStream.Play.StreamNotFound"`	`"error"`	Ошибка при загрузке. Данное событие генерируется, если указанный параметром метода `play()` FLV-файл не может быть найден

Применение обработчика события `onStatus` может оказаться необходимым, например, в ситуации, когда загружаемое видео нужно масштабировать относительно области, занимаемой видеообъектом, размеры которой заданы в рабочей среде.

Как известно, размеры загруженного видеоизображения в точности соответствуют размерам принимающего его видеообъекта. Если размеры видеофайла известны, можно заранее придать соответствующие размеры видеообъекту и избежать возможного искажения пропорций из-за масштабирования. Тем не менее если один и тот же видеообъект предназначен для вывода разных видеофайлов с разными размерами, необходимо выполнить его масштабирование, чтобы пропорции видео не были нарушены. Ранее в этой главе мы рассматривали свойства видеообъекта `width` и `height`, возвращающие размеры изображения исходного FLV-файла. Используя эти свойства и зная заданный размер видеообъекта, можно довольно просто выполнить его масштабирование таким образом, чтобы видео выводилось в предназначенной для него области без искажения пропорций. Единственная проблема заключается в том, что свойства `width` и `height` получают значения, отличные от нуля только после того, как видеофайл будет выведен в область видеообъекта или другими словами, после того, как произойдет полное заполнение буфера. В связи с этим масштабирование видеообъекта на основании размеров видеоролика можно выполнять только после обнаружения события `onStatus`, при котором свойство `code` будет иметь значение `"NetStream.Buffer.Full"`. Поскольку такое значение свойства может возвращаться регулярно (при каждом очередном заполнении буфера), важно позаботиться о том, чтобы масштабирование было выполнено только один раз, по-

сле первого заполнения буфера. Для этого можно ввести флаг, значение которого должно меняться сразу же после первого обнаружения события `onStatus` с соответствующим значением свойства `code`.

Ниже приводится фрагмент кода, иллюстрирующий описанный подход. При вызове метода `play()` воспроизведение файла запускается автоматически. Тем не менее в некоторых случаях желательно, чтобы воспроизведение загруженного файла запускалось только по требованию пользователя. Для этого нужно принудительно остановить воспроизведение в начале, как это показано далее. После заполнения буфера выполняются масштабирование видеообъекта и остановка воспроизведения в нулевой позиции (пауза в начале). Масштабирование выполняется с помощью функции `matchScreen()`, хранящейся во внешнем AS-файле, код которой приводится далее.

```
#include "matchScreen.as"
var nc:NetConnection = new NetConnection();
nc.connect(null);
var nStream:NetStream = new NetStream(nc);
videoObj.attachVideo(nStream);
/* Отключаем отображение видеообъекта. Его отображение будет включено
после выполнения масштабирования и центрирования */
videoObj._visible = false;
nStream.play("movie.flv");
// После первого заполнения буфера...
nStream.onStatus = function(infoObject:Object) {
switch (infoObject.code) {
case 'NetStream.Buffer.Full' :
       if (!played) {
       nStream.pause(true); // Останавливаем воспроизведение
       nStream.seek(0); // Переходим в начало
       videoObj._visible = true; // Включаем отображение видеообъекта
       // Пропорционально масштабируем и центрируем видеообъект
       matchScreen(videoObj._width, videoObj._height, videoObj);
       played = true; /* Переключаем флаг, чтобы все предыдущие действия
       выполнились только после первого заполнения буфера */
       }
       break;
       }
};
```

В следующем листинге приводится код функции, которая выполняет пропорциональное масштабирование видеообъекта `videoObj`, так чтобы он помещался в пределах экрана, ширина и высота которого задается параметрами `screenW` и `screenH` соответственно. Функция также центрирует видеообъект относительно границ экрана. Размеры экрана — это размеры видеообъекта, заданные в рабочей среде, их

можно считать или изменить, используя свойства _width и _height. Размеры видео — это исходные размеры FLV-файла, возвращаемые свойствами width и height. Для использования данной функции необходимо позиционировать на сцене видеообъект videoObj и придать ему размеры, соответствующие размерам воображаемого экрана, с учетом того, что видеоизображение не будет выходить за его границы. В основе работы данной функции лежит принцип, использованный при создании функции matchContent() в *главе 23*. На сопровождающем компакт-диске данная функция находится в папке Глава 24, в файле с именем matchScreen.as.

Функция matchScreen(). Пропорциональное масштабирование видеообъекта до заданных размеров с центрированием относительно границ экрана

```
function matchScreen(screenW:Number, screenH:Number, videoObj){
// Находим процентное выражение размеров области экрана от размеров видео
var ratioW = screenW / videoObj.width * 100;
var ratioH = screenH / videoObj.height * 100;
// Сохраняем значение коэффициента форматного соотношения
var vAspectRatio = videoObj.width / videoObj.height;
// Если и ширина, и высота видео больше или меньше ширины и высоты экрана
if ((ratioW < 100 && ratioH < 100) || (ratioW > 100 && ratioH > 100)) {
        /* Если отношение ширин меньше отношения высот, подгоняем размер
        видео по ширине */
        if (Math.min(ratioW, ratioH) == ratioW) {
        videoObj._width = screenW;
        videoObj._height = videoObj._width / vAspectRatio;
        }
        /* Если отношение ширин больше отношения высот, подгоняем размер
        видео по высоте */
        else {
        videoObj._height = screenH;
        videoObj._width = videoObj._height * vAspectRatio;
        }
}
        // Если видео уже и выше экрана, подгоняем по высоте
        else if (ratioW >= 100 && ratioH <= 100) {
        videoObj._height = screenH;
        videoObj._width = videoObj._height * vAspectRatio;
        }
        // Если видео шире и ниже экрана, подгоняем по ширине
        else if (ratioW < 100 && ratioH > 100) {
        videoObj._width = screenW;
```

```
        videoObj._height = videoObj._width / vAspectRatio;
        }
        // Центрируем видео относительно экрана
        videoObj._x += (screenW - videoObj._width)/2;
        videoObj._y += (screenH - videoObj._height)/2;
}
```

Прекращение загрузки

Для прекращения загрузки и воспроизведения FLV-файла, выполняемого с помощью объекта `NetStream`, нужно использовать метод `close()`. Вызов этого метода приводит к прекращению загрузки FLV-файла, обнуляет значения свойств `bufferLength` и `time` и удаляет с диска пользователя локально сохраненную копию FLV-файла, загруженного из сети. После вызова данного метода объект `NetStream` можно использовать для загрузки нового файла.

Следующий код при отпускании кнопки `closeBut` прекращает загрузку FLV-файла, удаляет его локальную копию с диска и очищает область видеообъекта.

```
closeBut.onRelease = function() {
        nStream.close();
        videoObj.clear();
};
```

Создание видеопроигрывателя с динамической загрузкой FLV-файла

В заключительной части этой главы в качестве иллюстрации возможностей работы с динамически загружаемым видео рассмотрим пример создания проигрывателя, снабженного типовыми элементами интерфейса. Его внешний вид представлен на рис. 24.2.

Вначале опишем функциональность, которой должен будет обладать наш проигрыватель.

- Загрузка внешнего FLV-файла должна выполняться только после нажатия соответствующей кнопки. В нашем примере эта кнопка снабжена именем `playStreamBut`. При загрузке файла его воспроизведение не должно начинаться автоматически. Запуск выполняется только при нажатии пользователем кнопки воспроизведения с именем `playBut`.
- Панель управления воспроизведением включает кнопки, позволяющие останавливать/запускать воспроизведение и с определенным шагом перемещать текущую позицию воспроизведения вперед и назад.
  - Кнопка воспроизведения с именем `playBut` и кнопка паузы с именем `pauseBut` располагаются строго друг над другом (вышележащая кнопка перекрывает нижележащую). При выполнении воспроизведения отображение кнопки

playBut отключается, и зритель видит только кнопку паузы. Если воспроизведение остановлено, отключается отображение кнопки паузы и зритель может воспользоваться кнопкой воспроизведения для его повторного запуска с текущей позиции.

- Пошаговая перемотка вперед и назад выполняется при нажатии кнопок fwBut и rwbut соответственно. Перемотка выполняется вне зависимости от того, остановлено воспроизведение или нет.

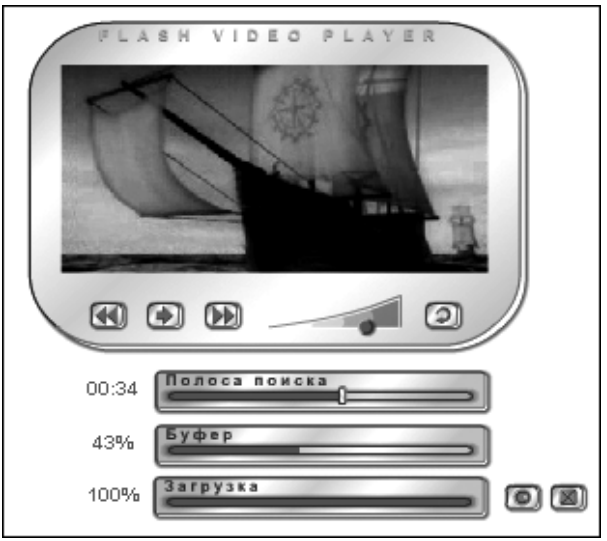


Рис. 24.2. Проигрыватель FLV-файла

- Имеется возможность циклического воспроизведения. Для этого используется кнопка-флаг loopBut. Если эта кнопка "включена", воспроизведение будет выполняться в зацикленном режиме.
- Проигрыватель снабжен ползунковым регулятором громкости звука. Принцип работы этого элемента совершенно аналогичен регулятору громкости, используемому в листинге 19.7. В связи с этим в целях экономии места мы не стали помещать его программный код на страницы книги. Читатель сможет найти его в исходном файле проигрывателя, помещенном на сопровождающий диск.
- Проигрыватель включает стандартную полосу поиска (seek bar) с бегунком, позволяющую переместить позицию воспроизведения в любую точку. В процессе воспроизведения бегунок перемещается по шкале, указывая текущую позицию воспроизводящей головки. Шкала поиска состоит из трех программно контролируемых элементов.
  - Сама шкала фиксированного размера представлена клипом seekSlideBar. Этот клип находится на заднем плане и имеет 20%-ю прозрачность. Его точка регистрации находится на левой границе.

- Синяя полоса индикации позиции воспроизведения, ширина которой изменяется в соответствии с текущим положением бегунка, представлена клипом `seekHighlightBar`, по размерам, в точности совпадающим с клипом `seekSlideBar` и расположенным строго над ним. Точка регистрации клипа `seekHighlightBar` также находится на его левой границе.
- Бегунок, положение которого соответствует текущей позиции воспроизведения, представлен клипом `seekSlider`. Он имеет двойное назначение. С одной стороны, при воспроизведении он автоматически перемещается, указывая текущую позицию. С другой стороны, перемещая его, пользователь может задать текущую позицию вручную.

- Проигрыватель снабжен стандартными полосами состояния, указывающими состояние процесса загрузки и степень заполненности буфера. Для отслеживания хода загрузки используются клип, представляющий полосу состояния с именем `loadStatusBar` и текстовое поле с именем `statusTxt`. Для индикации степени заполненности буфера аналогичным образом используются клип `bufferStatusBar` и текстовое поле `bufferStatusTxt`.
- Все элементы интерфейса снабжены всплывающими подсказками, реализованными с помощью функции `addTextHint()`, код которой включается с помощью директивы `#include`. Описание этой функции и комментарии по ее созданию были приведены в листинге 14.1.

Исходный программный код снабжен подробными комментариями, поэтому здесь мы остановимся только на его основных функциональных элементах.

Контроль состояния загрузки осуществляется с помощью функции `checkLoadStatus()`, которая вызывается периодически посредством глобальной функции `setInterval()`. В основе работы функции `checkLoadStatus()` лежит стандартный алгоритм предварительного загрузчика.

Контроль состояния буфера осуществляется с помощью функции `checkBufferStatus()`, также периодически вызываемой посредством глобальной функции `setInterval()`. Степень заполненности буфера вычисляется как процент его текущего заполненного объема от объема, заданного методом `setBufferTime()`.

Смещение бегунка, указывающего текущую позицию воспроизведения, осуществляется посредством периодического вызова функции `checkTimeStatus()`. Для этого также применяется функция `setInterval()`. Функция `checkTimeStatus()` также сравнивает текущую позицию воспроизведения с позицией, соответствующей концу видеоролика, и при активизации режима зацикливания автоматически возвращает воспроизводящую головку в начало с помощью метода `seek()`.

Отметим, что периодический вызов функций `checkLoadStatus()` и `checkBufferStatus()` начинается только после первого заполнения буфера, то есть когда изображение появилось в области видеообъекта. Периодический вызов функции `checkTimeStatus()` начинается только при фактическом запуске воспроизведения, то есть при нажатии кнопки `playBut`. Это сделано для того, чтобы не тратить ресурсы на постоянное вычисление положения бегунка, в то время как он остается в одной и той же позиции, поскольку воспроизведение остановлено. Идентификационный номер каждого интервала, установленного функцией `setInterval()`, сохра-

няется в переменной. Это дает возможность при необходимости прекращать периодическое выполнение той или иной функции.

Для выполнения ручной прокрутки (поиска) с помощью бегунка `seekSlider` применяется стандартный алгоритм, неоднократно используемый ранее. Следует обратить внимание на то, что на время перемещения бегунка пользователем необходимо прекратить периодический вызов функции `checkTimeStatus()`, очистив соответствующий интервал при нажатии на `seekSlider`. Если этого не сделать, код функции, выполняющей автоматическое смещение бегунка, и код ползункового регулятора будут "конкурировать" друг с другом.

Программный код проигрывателя приведен в листинге 24.1. Здесь не показан фрагмент, реализующий регулятор громкости, и фрагмент, создающий всплывающие подсказки. Полный вариант находится на сопровождающем книгу компакт-диске.

Листинг 24.1. Проигрыватель FLV-файла

```
// Включаем код функции, выполняющей масштабирование и центрирование
// видео
#include "matchScreen.as"
var nc:NetConnection = new NetConnection();
nc.connect(null);
var nStream:NetStream = new NetStream(nc);
videoObj.attachVideo(nStream);
// Сохраняем url, по которому расположен FLV-файл
var url:String = "video/movie.flv"
/* Сохраняем значение продолжительности видео в секундах (она должна быть
известна заранее) */
var vTotalTime:Number = 59.416;
/* Отключаем отображение кнопки pauseBut, так как воспроизведение по
умолчанию остановлено */
pauseBut._visible = false;
/* Отключаем отображение видеообъекта. Его отображение будет включено
после выполнения масштабирования и центрирования */
videoObj._visible = false;
// Устанавливаем нулевую ширину для всех полос состояния
seekHighlightBar._width = 0;
bufferStatusBar._width = 0;
loadStatusBar._width = 0;
/* Отключаем возможность перемещения ползунка поиска до тех пор, пока не
началась загрузка видео */
seekSlider.enabled = false;
// Задаем величину буфера - 5 секунд
nStream.setBufferTime(5);
```

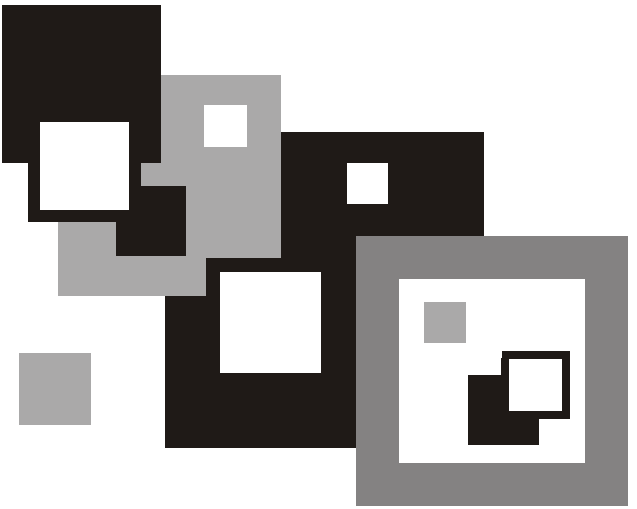
```
// Функция, выполняющая обновление полосы состояния загрузки
function checkLoadStatus() {
        var loaded = nStream.bytesLoaded;
        var total = nStream.bytesTotal;
        var streamStatus = Math.round(loaded / total * 100);
        statusTxt.text = streamStatus + "%";
        loadStatusBar._xscale = streamStatus;
        if (streamStatus >= 100) {
                clearInterval(loadInt);
        }
}
// Функция, выполняющая обновление полосы состояния буфера
function checkBufferStatus() {
        var bufferSta-
tus:Number=Math.min(Math.round(nStream.bufferLength/↳
nStream.bufferTime * 100), 100);
        bufferStatusBar._xscale = bufferStatus;
        bufferStatusTxt.text = bufferStatus + "%";
}
/* Функция, позиционирующая бегунок в соответствии с текущей позицией
воспроизведения и выводящая ее значение в минутах и секундах */
function checkTimeStatus() {
        var curPosition:Number = nStream.time;
        seekSlider._x = nStream.time * seekSlideBar._width / vTotalTime +↳
        seekSlideBar._x;
        seekHighlightBar._width = seekSlider._x - seekSlideBar._x;
        var minutes = Math.floor(curPosition / 60);
        var seconds = Math.floor(curPosition % 60);
        if (seconds < 10) {
                seconds = "0" + seconds;
        }
        if (minutes < 10) {
                minutes = "0" + minutes;
        }
        seekStatusTxt.text = minutes + ":" + seconds;
        /* Если режим зацикливания включен, при достижении конца видео
        переходим в начало. Переменная loop - это флаг, значение которого
        устанавливается при нажатии на кнопку loopBut */
        if (nStream.time >= vTotalTime && loop) {
```

```
                nStream.seek(0);
        }
}
// После первого заполнения буфера...
nStream.onStatus = function(infoObject:Object) {
        switch (infoObject.code) {
        case 'NetStream.Buffer.Full' :
                if (!played) {
                nStream.pause(true); // Останавливаем воспроизведение
                nStream.seek(0); // Переходим в начало
                seekSlider.enabled = true; // Включаем возможность поиска
                videoObj._visible = true; // Включаем отображение
                                          // видеообъекта
                // Пропорционально масштабируем и центрируем видеообъект
                matchScreen(videoObj._width, videoObj._height, videoObj);
                played = true; /* Переключаем флаг, чтобы все предыдущие
                действия выполнились только после первого заполнения
                буфера. Переменная played принимает значение true только
                после того, как буфер был заполнен в первый раз, и сохраняет
                это значение до удаления FLV с помощью метода close()*/
                }
                break;
        }
};
// При нажатии кнопки начинаем загрузку и воспроизведение FLV-файла
playStreamBut.onRelease = function() {
if(!played){ //Только в том случае, если он уже не загружается
        nStream.play(url);
        /* Инициируем периодическое обновление состояния полосы загрузки
        и полосы буфера. Поскольку в начале воспроизведение остановлено,
        бегунок текущей позиции остается на месте */
        loadInt = setInterval(checkLoadStatus, 300);
        bufferInt = setInterval(checkBufferStatus, 300);
        }
};
// Выполнение ручной прокрутки видео
seekSlider.onPress = function() {
        /* При нажатии на бегунок прекращаем периодический вызов функции
        checkTimeStatus, позиционирующей его в соответствии с текущей
        позицией воспроизведения */
        clearInterval(timeInt);
```

```
        this.startDrag(false, seekSlideBar._x, seekSlideBar._y,↳
        seekSlideBar._x + seekSlideBar._width, seekSlideBar._y);
        this.onMouseMove = function() {
                /* Включаем отображение видеообъекта. Это нужно сделать на
                тот случай, если пользователь начнет выполнять ручную
                прокрутку до первого заполнения буфера */
                videoObj._visible = true;
                /* Установка позиции воспроизведения в соответствии
                с положением бегунка */
                nStream.seek(Math.floor(vTotalTime / seekSlideBar._width *↳
                (seekSlider._x - seekSlideBar._x)));
                seekStatusTxt.text = Math.floor(nStream.time);
                updateAfterEvent();
        };
};
// При отпускании мыши прекращаем перетаскивание бегунка
seekSlider.onRelease = function() {
        this.stopDrag();
        /*Вновь инициируем периодический вызов функции checkTimeStatus,
        для того чтобы положение бегунка соответствовало текущей позиции
        воспроизведения */
        if(pauseBut._visible){
        timeInt = setInterval(checkTimeStatus, 100);
        }
        else{
        checkTimeStatus();
        }
        delete this.onMouseMove;
};
seekSlider.onReleaseOutside = seekSlider.onRelease;
/* Кнопки запуска и остановки воспроизведения располагаются друг над
другом. Видимой является только одна из них. При нажатии на одну из
кнопок выполняем соответствующее действие, отключаем видимость данной
кнопки и включаем видимость другой */
playBut.onRelease = function() {
        /* Запуск может быть выполнен только после первого заполнения
        буффера */
        if(played){
        timeInt = setInterval(checkTimeStatus, 100);
        nStream.pause();
        this._visible = false;
```

```
        pauseBut._visible = true;
        }
};
pauseBut.onRelease = function() {
        nStream.pause();
        this._visible = false;
        playBut._visible = true;
        /* Прекращаем вызов функции checkTimeStatus(), когда
        воспроизведение остановлено */
        clearInterval(timeInt);
};
// Перемотка вперед с шагом 2 секунды
fwBut.onRelease = function() {
        nStream.seek(nStream.time + 2);
};
// Перемотка назад с шагом 5 секунд
rwBut.onRelease = function() {
        if (nStream.time >= 5) {
                        nStream.seek(nStream.time - 5);
        }
        else {
                nStream.seek(0);
        }
};
/* Кнопка-флаг, определяющая, выполняется воспроизведение в зацикленном
режиме или нет. Клип loopBut состоит из двух кадров с метками "loopOn" и
"loopOff" */
loopBut.onPress = function() {
        if (!loop) {
                this.gotoAndStop("loopOn");
                loop = true; // Устанавливаем флаг
        }
        else {
                this.gotoAndStop("loopOff");
                loop = false; // Снимаем флаг
        }
};
/* При отпускании кнопки прекращаем загрузку, очищаем видеообъект
и возвращаем все элементы интерфейса в исходное состояние */
closeBut.onRelease = function() {
```

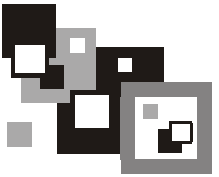
```
        // Прекращаем загрузку, удаляем локальную копию файла
        nStream.close();
        // Очищаем область видеообъекта
        videoObj.clear();
        // Устанавливаем полосу загрузки в исходное состояние
        loadStatusBar._width = 0;
        statusTxt.text = "0%";
        // Отключаем возможность перемещения ползунка поиска
        seekSlider.enabled = false;
        /* Прячем кнопку паузы и делаем видимой кнопку воспроизведения,
        поскольку вначале воспроизведение остановлено */
        playBut._visible = true;
        pauseBut._visible = false;
        /* Прекращаем периодический вызов функций checkTimeStatus() и
        checkBufferStatus() */
        clearInterval(timeInt);
        clearInterval (bufferInt);
        /* Вызываем checkTimeStatus() и checkBufferStatus() в последний
        раз, чтобы установить бегунок и полосу состояния буфера в нулевое
        положение */
        checkTimeStatus();
        checkBufferStatus();
        /* Переключаем флаг, чтобы при следующей загрузке снова выполнить
        масштабирование и остановить воспроизведение в начале */
        played = false;
};
```



ЧАСТЬ III

Объектно-ориентированное программирование. Создание компонентов

Глава 25



Создание пользовательских классов

Создайте систему, которой сможет пользоваться даже дурак,
и только дурак захочет ею пользоваться.

Принцип Шоу

В предыдущих главах при изучении возможностей встроенных классов ActionScript мы постоянно использовали терминологию и категории, присущие объектно-ориентированному программированию. Такие понятия, как класс, метод, свойство, конструктор, должны быть хорошо знакомы читателю. Причина этого состоит в том, что применение большинства встроенных классов формально ничем не отличается от применения классов, созданных пользователем. Поэтому программирование в Flash всегда в большей или меньшей степени предполагает использование элементов объектно-ориентированного подхода.

В Flash существует два принципиальных подхода к разработке программы: процедурный и объектно-ориентированный. Процедурный подход предполагает использование функций в качестве элементов, реализующих выполнение типовых действий. Так, например, для перемещения объекта можно создать функцию `move()`, которая в качестве параметра будет принимать ссылку на объект и расстояние, на которое его нужно сместить. Контейнерами для хранения данных в этом случае являются переменные. Так, например, переменная `moveIncrement` может хранить значение, соответствующее шагу смещения. Подобный подход мы использовали в примерах, рассматриваемых в предыдущих главах этой книги. Процедурный подход очень удобен при создании небольших программ, предназначенных для решения конкретных задач. Однако при создании сложных проектов его применение приводит к возникновению значительных трудностей, связанных с обеспечением взаимодействия между разными частями программы, внесением изменений и многократным использованием типовой функциональности.

Объектно-ориентированный подход предполагает создание самостоятельных программных модулей — классов, каждый из которых выполняет возложенную на него задачу. Такие модули, будучи совершенно независимы друг от друга, могут успешно взаимодействовать в приложении. Объектно-ориентированный подход позволяет реализовать более качественную организацию структуры программы и, что особенно важно, обеспечить широкие возможности повторного использования кода. Это оказывается необходимо при создании комплексных программ, требующих многократ-

ного применения типовой функциональности, которую можно легко и эффективно модифицировать.

Как правило, ни один из подходов не используется в чистом виде. Они всегда применяются в сочетании, дополняя друг друга. Так, например, класс может описывать функциональность абстрактного объекта, а его "физическое" воплощение (например, визуализация) может быть реализовано с помощью кода с использованием процедурного подхода.

В этой главе мы рассмотрим процесс создания пользовательских классов и приведем практический пример реализации проекта с применением объектно-ориентированного подхода.

Файлы классов. Понятие *classpath*

В ActionScript 2.0 все пользовательские классы сохраняются в отдельных файлах с расширением as. Для создания файлов пользовательских классов можно использовать Flash MX 2004 Professional или любой внешний текстовый редактор.

Каждый пользовательский класс должен сохраняться в виде самостоятельного файла. Один AS-файл не может содержать объявления нескольких классов. Имя каждого файла должно в точности совпадать с названием содержащегося в нем класса. Так, например, файл, содержащий класс `Lift`, должен быть сохранен с именем Lift.as, файл, содержащий класс `Building`, должен быть сохранен с именем Building.as и т. д.

Несмотря на то, что пользовательские классы создаются во внешних файлах, они существуют отдельно от использующего их исходного FLA-файла только до момента публикации. При компиляции исходного документа программный код всех используемых в нем пользовательских классов автоматически включается в конечный документ и компилируется точно так же, как если бы он был создан в рабочей среде. Таким образом, результирующий SWF-документ включает в себя весь программный код, обеспечивающий функциональность, и уже не требует наличия внешних AS-файлов. Для того чтобы в процессе публикации все используемые в документе пользовательские классы могли быть включены в конечный документ и скомпилированы, необходимо "указать" Flash их местоположение. Путь к внешним пользовательским классам называется **classpath**. При выполнении публикации документа Flash ищет файлы используемых в нем пользовательских классов в тех каталогах, на которые указывает **classpath**. Настройки, указывающие Flash местоположение внешних файлов классов, могут быть заданы глобально и могут использоваться сразу для всех документов или только для конкретного документа. Для задания глобальных настроек можно выполнить одно из следующих действий.

- Выполнить команду главного меню **Edit>Preferences...** и в появившемся диалоговом окне **Preferences** перейти на вкладку **ActionScript**. Здесь нужно нажать на кнопку **ActionScript 2.0 Settings...**.
- В контекстном меню панели Actions выполнить команду **Edit>Preferences...** и нажать на кнопку **ActionScript 2.0 Settings...**.

В результате этих действий появляется диалоговое окно **ActionScript Settings**, представленное на рис. 25.1.

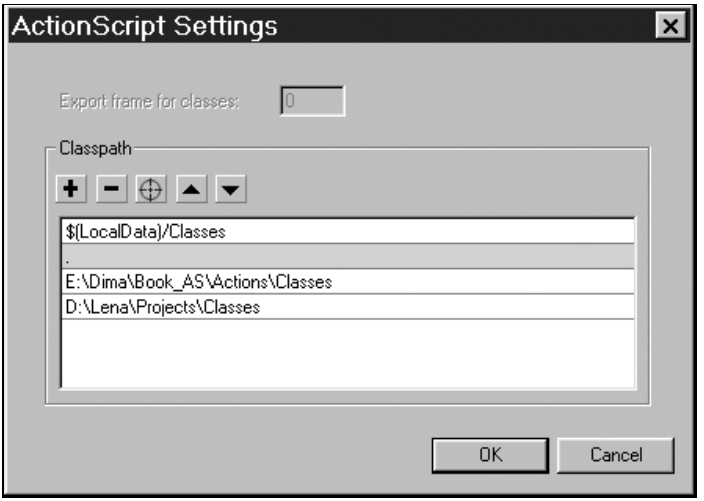


Рис. 25.1. Диалоговое окно **ActionScript Settings**

В разделе **Classpath** выводится перечень каталогов, в которых Flash выполняет поиск пользовательских классов при публикации. Причем последовательность расположения ссылок в списке определяет последовательность, в которой будет выполняться поиск. В первую очередь Flash ищет файлы классов в папке, путь к которой указан в верхней строке, затем, не найдя их там, переходит в следующий каталог и т. д. Этот список по умолчанию содержит два стандартных обозначения: **$(LocalData)/Classes** и знак точки (.). Первое из этих значений — это ярлык для папки, в которой хранятся все AS-файлы, соответствующие встроенным классам Flash. Эта папка автоматически создается при установке Flash MX 2004 и имеет примерно следующее местоположение:

- в Windows 98 — **C:\Windows\Application Data\Macromedia\Flash MX 2004\<language>\Configuration\Classes**;
- в Windows 2000 и Windows XP — **C:\Documents and Settings\<user>\Local Settings\ Application Data\Macromedia\Flash MX2004\<language>\Configuration\ Classes**.

Точка — это относительный адрес, указывающий на текущий каталог, в котором расположен публикуемый FLA-документ. Таким образом, Flash вначале выполнит поиск в стандартной папке с классами, затем в папке, в которой расположен публикуемый FLA-документ. Для того чтобы добавить новый каталог в глобальный список путей к классам, можно воспользоваться кнопкой с пиктограммой (+) и затем, щелкнув по кнопке **Browse To Path**, указать путь к требуемому каталогу. Для того чтобы удалить путь к классу, его нужно выделить и нажать кнопку с пиктограммой (−). Для изменения порядка следования путей в списке (и соответственно, порядка,

в котором Flash выполняет поиск файлов классов), нужно воспользоваться пиктограммами со стрелками. Отметим, что путь может быть задан с использованием относительной адресации, например **../classes**. В этом случае его нужно просто вручную ввести в соответствующую строку.

Наряду с глобальными настройками **Classpath** путь к файлам классов может быть задан локально на уровне конкретного документа. Для этого необходимо выполнить команду главного меню **File>Publish Settings...** и в появившемся диалоговом окне переместиться на вкладку **Flash**. Здесь нужно нажать на кнопку **Settings** (она активизируется только при установке значения **ActionScript 2.0** в списке **ActionScript Version**). По умолчанию список локальных путей к файлам классов пуст. Его можно сформировать точно так же, как и глобальный список путей. Локальный список путей действует только при публикации текущего документа и не распространяется на другие документы. При разрешении ссылок на классы, используемые в FLA-документе, поиск сначала выполняется в каталогах, заданных локальными настройками. Если локальный список адресов пуст или если требуемый файл класса там не найден, поиск выполняется в каталогах, заданных глобальными настройками. В случае отсутствия файла класса в глобальных каталогах компилятор выведет сообщение об ошибке, связанной с невозможностью загрузки класса (The class *имя класса* could not be loaded).

Во время публикации код класса по умолчанию включается в первый кадр монтажной линейки использующего данный класс документа. Следствием этого является то, что загрузка включенного в документ программного кода, соответствующего пользовательским классам, происходит перед загрузкой остального содержимого первого кадра фильма. Если код используемых в документе классов достаточно велик, это обстоятельство может привести к некорректной работе предзагрузчика, расположенного в первом кадре, поскольку его инициализация и работа начнутся только после загрузки пользовательских классов. Поэтому такой предзагрузчик не сможет полностью отразить ход загрузки документа, и до его появления пользователю придется ожидать окончания загрузки перед пустым экраном.

Одним из вариантов решения описанной проблемы является включение кода классов не в первый кадр (как это происходит по умолчанию), а в один из последующих кадров фильма. При задании локальных настроек в диалоговом окне **ActionScript Settings** кроме пути к файлам классов можно указать номер кадра основной монтажной линейки, в который будет включен их программный код. Для этих целей используется поле **Export frame for classes**. Если значение, заданное в этом поле, отлично от единицы, при публикации программный код классов будет включен не в первый кадр, а в кадр, номер которого соответствует введенному значению. Важно помнить, что код классов должен быть загружен до того, как будут загружены элементы, использующие эти классы. Поэтому номер кадра, заданного в поле **Export frame for classes**, не должен быть меньше номера кадра, в котором класс будет впервые использован. Так, если код классов включается в кадр с номером 5, то обращение к этим классам может выполняться, только начиная с пятого кадра. Если в поле **Export frame for classes** будет введен номер несуществующего кадра, при публикации Flash не будет включать классы в конечный документ, а в панель **Output** будет выведено предупреждающее сообщение.

В процессе компиляции Flash автоматически создает для каждого пользовательского класса, применяемого в проекте, особый файл в формате ASO (ActionScript Object),

имеющий расширение aso. В этом файле сохраняется скомпилированная версия класса в виде байткода (bytecode) — последовательности символов, в которую в результате компиляции преобразуются любые сценарии ActionScript. Все ASO-файлы сохраняются в каталоге, который указан в глобальных настройках **Classpath** в качестве каталога, используемого по умолчанию. Обычно это каталог, на который указывает ярлык **$(LocalData)/Classes**.

Назначение ASO-файлов заключается в том, что они позволяют ускорить процедуру компиляции. Дело в том, что если исходный код класса, хранящийся в AS-файле, не претерпевает изменений, то при компиляции (выполняющейся во время публикации FLA-файла) код класса не подвергается повторной компиляции. Вместо этого Flash использует уже скомпилированную версию класса, хранящуюся в FLA-файле, поэтому публикация документа выполняется значительно быстрее.

ASO — это внутренний формат файлов Flash, которые используются исключительно на этапе компиляции и не оказывают никакого влияния на функционирование конечного SWF-файла.

Примечание

Иногда после внесения изменений в код класса при публикации Flash вместо повторной компиляции использует более старую версию этого класса, хранящуюся в соответствующем ASO-файле. Если изменения, выполненные в AS-файле класса, после публикации не отразились на конечном SWF-файле, можно посоветовать удалить соответствующий этому классу ASO-файл, расположенный в стандартном каталоге Classes, и затем повторно опубликовать документ.

Следует иметь в виду, что в Flash существует ограничение на максимальный объем AS-файла с определением класса. Максимальный объем скомпилированного класса не должен превышать 32 767 байт (примерно 1500 строк). В противном случае Flash выведет сообщение об ошибке. В связи с этим слишком обширные классы необходимо разбивать на части и оформлять каждую из них как самостоятельный класс.

Объявление класса

Класс — это программный модуль, представляющий собой своеобразный шаблон, на основе которого выполняется создание объектов определенного типа, моделирующих объекты реального мира или абстрактные процессы и категории. Класс состоит из членов, определяющих набор характеристик и функциональных возможностей представляемого им объекта. Характеристики объекта описываются *свойствами* (properties) класса, а выполняемые с ним или с его помощью действия — *методами* (methods). В качестве примера возьмем хорошо известный и часто используемый в повседневной жизни объект Лифт (Lift). Для того чтобы смоделировать Лифт, мы можем создать класс `Lift`, содержащий набор свойств, описывающих характеристики лифта, и методов, позволяющих осуществлять его эксплуатацию. В свойствах объекта Лифт могут храниться данные о грузоподъемности лифта, его скорости, текущем состоянии (свободен/занят) и т. д. Методы объекта Лифт могут реализовывать перемещение лифта с этажа на этаж, осуществлять вызов лифта на указанный этаж и т. д.

Определение класса состоит из следующих основных элементов.

- Объявление класса.
- Объявление свойств.
- Объявление конструктора.
- Объявление методов.

Каждый экземпляр класса, то есть конкретный объект, получает в свое распоряжение или наследует свойства и методы, объявленные в породившем его классе.

Объявление класса выполняется во внешнем AS-файле с помощью предложения `class` и в наиболее простом случае имеет следующий синтаксис:

```
class имя класса {
тело класса
}
```

Здесь *имя класса* — это имя, служащее идентификатором класса. Имя класса должно соответствовать всем правилам, предъявляемым к идентификаторам `ActionScript`. Принято имена классов начинать с прописной буквы. Это правило не является обязательным, однако его соблюдение является повсеместно используемой практикой и поэтому настоятельно рекомендуется. Необходимо помнить, что имя класса должно в точности совпадать с именем файла класса (несоблюдение этого правила вызовет ошибку!).

Примечание

Использование предложения `class` в сценарии FLA-документа вызовет ошибку. Объявление классов с помощью этого предложения выполняется только во внешних AS-файлах.

Все объявления свойств и методов класса должны обязательно размещаться в его теле, то есть внутри фигурных скобок, в противном случае при проверке синтаксиса или публикации конечного файла возникнет ошибка компиляции. Далее приводится пример объявления класса `Lift`:

```
class Lift {
...
}
```

Каждый класс определяет специфичный для него тип данных, название которого совпадает с именем класса. Эта особенность присуща как встроенным классам ActionScript, так и классам, создаваемым пользователем. Так, например, класс `Date` определяет тип данных Date, класс `Color` определяет тип данных `Color`, пользовательский класс `Lift` определяет тип данных `Lift` и т. д. Это позволяет реализовать строгий контроль типов и, таким образом, избежать ошибок несоответствия типов данных. Так, например, если мы создали переменную типа `Lift`, используя строгую типизацию, и затем сохранили в ней ссылку на объект `Lift`, это гарантирует нас от возможности вызова метода или свойства, которые не определены в классе `Lift`.

Создание экземпляра класса

Для реализации возможностей, заложенных в классе, как правило, требуется создать его экземпляр, то есть конкретный объект, тип которого соответствует типу, определяемому классом. Так, например, класс `Lift` описывает набор характеристик и функциональность, присущие категории Лифт, то есть всем лифтам вообще. Однако при решении практической задачи, например, создании модели дома, мы должны создать конкретные лифты и снабдить их функциональностью, заложенной в классе `Lift`. Класс `Lift` может включать объявление свойства `currentFloor`, определяющего, на каком этаже находится лифт, и содержать объявление метода `elevate()`, выполняющего перемещение лифта на заданный этаж. Создавая экземпляр класса `Lift`, мы получаем конкретный объект, имеющий тип `Lift` и обладающий свойством `currentFloor` и методом `onElevate()`. Разные лифты могут находиться на разных этажах, аналогично, экземпляры одного и того же класса, обладая одинаковым набором собственных свойств, объявленных в классе, могут иметь разные значения этих свойств. Каждый экземпляр класса `Lift` имеет свойство `currentFloor`, однако у разных экземпляров класса `Lift` значение свойства `currentFloor` может быть различным. Точно так же каждый экземпляр класса `Lift` обладает методом `elevate()`, позволяющим ему перемещаться на заданный для данного лифта этаж.

Важнейшей особенностью объектно-ориентированного программирования является сокрытие подробностей внутренней реализации функциональности объекта. Так, создав объект `Lift`, мы можем вызвать метод `elevate()`, чтобы отправить его на нужный этаж, однако нас не интересует, каким образом реализуется это перемещение. Алгоритм, лежащий в основе метода `elevate()`, скрыт, или *инкапсулирован*, внутри определения класса. Инкапсуляция, то есть сокрытие подробностей внутренней реализации объектов — это один из ключевых аспектов объектно-ориентированного подхода. Создав экземпляр класса, пользователь может осуществлять управление им, используя программный интерфейс, представляющий собой совокупность свойств и методов, определенных в классе. При этом ему не нужно знать устройство самого класса для успешного применения его возможностей.

Следует четко понимать, что создание класса выполняется во внешнем AS-файле, в то время как создание экземпляра этого класса, скорее всего, должно быть выполнено в сценарии исходного FLA-документа (естественно, экземпляр одного класса также может быть создан внутри определения другого). При этом необходимо убедиться в правильности указания глобального и/или локального пути к классам, чтобы Flash смог отыскать используемый класс.

Для создания экземпляра класса используется хорошо знакомый нам синтаксис:

`var` *`переменная`*`:`*`тип`* `= new` *`конструктор`* `(`*`п1`*`, `*`п2`*`,..., `*`пN`*`)`

Здесь *`переменная`* — это имя переменной, в которой сохраняется ссылка на экземпляр класса, созданный конструктором.

`тип` — это тип данных, определяемый классом, название которого совпадает с именем класса.

`конструктор` — это конструктор класса.

Параметры *п1, п2, ..., пN* — это параметры, которые передаются конструктору для инициализации собственных свойств создаваемого объекта. Например, при создании объекта `Lift` можно сразу поместить его на заданный этаж, передав соответствующий параметр конструктору.

Приведем пример создания объекта `Lift` и поместим его на пятый этаж:

```
var myLift:Lift = new Lift(5);
```

Пакеты

Как уже говорилось, файлы классов хранятся в определенных каталогах, путь к которым задается локальными или глобальными настройками **classpath**. Размещение всех файлов классов, относящихся к различным проектам, в одном и том же каталоге запутывает их организацию и может привести к конфликту имен. Поскольку каждый файл в пределах одного каталога должен иметь уникальное имя, приходится следить за тем, чтобы все классы имели различные имена. Это оказывается возможным далеко не всегда, особенно при использовании классов, созданных другими разработчиками.

Решение данной проблемы заключается в применении пакетов, предоставляющих удобные возможности четкой организации структуры классов, исключающей возникновение конфликтов имен. *Пакетом* (package) называется папка, содержащая файлы классов, расположенная в каталоге, путь к которому задается локальными или глобальными настройками **classpath**. Так, например, если в настройках пути к классам указан каталог Classes, а в папке Classes находится вложенная папка Elevators, содержащая файл Lift.as, это означает, что класс `Lift` находится в пакете Elevators. Использование пакетов позволяет упорядочить структуру классов, сгруппировав близкие по назначению классы в индивидуальные пакеты.

Если класс входит в состав пакета, его имя обязательно должно отражать местоположение пакета относительно каталога, указанного в настройках **classpath**. Если файл класса Lift.as сохраняется в папке Elevators, расположенной в каталоге, указанном в настройках **classpath**, имя класса должно включать название пакета:

```
class Elevators.Lift {
...
}
```

Обратите внимание, что имя класса включает перечень каталогов, отражающий местоположение пакета относительно папки, на которую указывает **classpath**. Причем в имени класса эти каталоги отделяются друг от друга точкой. Например, класс, сохраненный в каталоге **Projects/Commercials/Interaction** в файле DrawAPI.as, принадлежит пакету Projects.Commercials.Interaction и имеет имя Projects.Commercials.Interaction.DrawAPI. Соответственно, его объявление будет выглядеть следующим образом:

```
class Projects.Commercials.Interaction.DrawAPI{
...
}
```

Следует иметь в виду, что имя пакета должно соответствовать правилам, предъявляемым к идентификаторам ActionScript (за исключением того, что в нем могут использоваться точки). Это необходимо учитывать при создании соответствующих папок в файловой системе.

При создании экземпляра класса, сохраненного в пакете, необходимо использовать его полное имя, включающее указание пакета, к которому принадлежит класс, например:

```
var myLift:Elevators.Lift = new Elevators.Lift (5);
```

Если полное имя класса оказывается очень длинным, можно воспользоваться специальным ключевым словом `import`. Данное ключевое слово позволяет один раз указать транслятору, в каком пакете находится используемый класс, и в остальном программном коде использовать его сокращенное имя. Ключевое слово `import` имеет следующий синтаксис:

```
import полное имя класса
```

Так, например, создание экземпляра класса `Projects.Commercials.Interaction.DrawAPI` будет выглядеть следующим образом:

```
import Projects.Commercials.Interaction.DrawAPI;
var api:DrawAPI = new DrawAPI();
```

Если требуется использовать несколько классов, хранящихся в одном и том же пакете, то вместо того, чтобы импортировать их по отдельности, можно сразу импортировать все классы пакета. Для этих целей после указания имени пакета нужно поместить символ `*`. Например, следующий код импортирует сразу все классы, хранящиеся в пакете Elevators, что позволит при обращении к ним использовать сокращенные имена.

```
import Elevators.*
```

Отметим, что если в коде отсутствует ссылка на класс (то есть он не используется), импорт этого класса не выполняется даже при наличии ключевого слова `import`. Поэтому импорт всех классов пакета не приводит к неоправданному увеличению объема файла из-за включения "лишних" классов. В конечный документ входит программный код только тех классов, которые реально в нем используются.

Действие ключевого слова `import` распространяется только на тот код, в котором оно используется. Это означает, что при необходимости применения класса в сценариях разных кадров или объектов каждый раз необходимо выполнять его импорт.

Объявление свойств

Свойства — это контейнеры данных, описывающие различные характеристики объекта. Класс содержит определение свойств, присущих всем объектам, созданным на основе этого класса. Все экземпляры класса обладают одинаковым набором свойств, однако их значения у разных экземпляров могут быть разными.

Обычно объявление свойств класса выполняется в его начале, то есть после предложения с объявлением самого класса. Объявление свойств выполняется с помощью

ключевого слова var точно так же, как явное объявление переменных. При объявлении свойств можно использовать строгую типизацию. Это не является обязательным требованием, однако настоятельно рекомендуется, поскольку позволяет избежать ошибок несоответствия типа, определяемых на этапе компиляции. Свойство, как и обычная переменная, может содержать данные любого типа.

```
class Lift {
var speed:Number;
var isBusy:Boolean;
var message:String;
var lifts:Array;
...
}
```

При объявлении свойств можно сразу же присвоить им значения, используемые по умолчанию. В этом случае при создании экземпляра класса из него сразу же можно будет получить доступ к значению данного свойства. Значения, присваиваемые свойствам при их объявлении, обязательно должны являться константами, то есть на момент компиляции их значение должно быть известно и фиксировано. Попытка при объявлении присвоить свойству значение, которое определяется только в процессе воспроизведения фильма, вызовет ошибку компиляции. По этой же причине нельзя присвоить свойству выражение, включающее обращение к встроенному классу.

```
class Test {
        var a:Number = 300; // Константа
        var b:Number = a / c; /* Константа. Не вызовет ошибки даже
        несмотря на то, что свойство с объявлено позднее */
        var c:Number = 3;
        var date:String = Date(); /* Вызовет ошибку, поскольку содержит
        обращение к встроенному классу */
        var move:Function = m(); /* Вызовет ошибку, поскольку m()
        не является константой */
        function m() {
        }
}
```

Важно помнить, что любое свойство, к которому обращаются инструкции внутри определения класса, должно быть объявлено с помощью ключевого слова var. Обращение к несуществующему свойству не приведет к его автоматическому созданию (как это происходит в случае с переменными), а вызовет ошибку компиляции (в панель **Output** будет выведено: There is no property with the name '*имя свойства*').

Свойства, которые объявляются в классе с помощью ключевого слова var, являются *наследуемыми*. Эти свойства единожды создаются в классе и могут быть доступны его экземплярам. Однако важно понимать, что наследуемые свойства не определяются для каждого нового объекта, порожденного классом, а просто заимствуются им из класса. Это позволяет существенно оптимизировать использование памяти, поскольку одни и те же элементы данных не копируются в каждый экземпляр индиви-

дуально. Вместо этого все экземпляры могут обратиться к данным, хранящимся в одном месте, по ссылке, которую предоставляет соответствующее свойство, объявленное в классе. Проиллюстрируем это на примере:

```
class Lift {
        var currentFloor = 7;
        var speed = 500;
}
```

Теперь в FLA-файле создадим два экземпляра класса `Lift` и определим значения свойств `currentFloor` и `speed` для каждого из них:

```
var myLift1:Lift = new Lift();
var myLift2:Lift = new Lift();
trace (myLift1.currentFloor); // Выводит 7
trace (myLift1.speed); // Выводит 500
trace (myLift2.currentFloor); // Выводит 7
trace (myLift2.speed); // Выводит 500
```

Как мы видим, оба экземпляра ссылаются на одни и те же значения свойств, объявленных в классе. Теперь убедимся в том, что свойства `currentFloor` и `speed` действительно не принадлежат самим экземплярам, а просто доступны им (то есть входят в их область видимости). Для этого попробуем перечислить свойства объектов `myLift1` и `myLift2` в цикле `for...in`.

```
for (i in Lift1) {
        trace(i); // Ничего не выводится
}
for (i in Lift2) {
        trace(i); // Ничего не выводится
}
```

Таким образом, экземпляры класса `Lift` просто имеют возможность ссылаться на свойства, объявленные в классе или, другими словами, наследуют их.

Если значение унаследованного свойства для конкретного объекта должно отличаться от значения, заданного при объявлении этого свойства в классе, можно вызвать это свойство из порожденного классом экземпляра и присвоить ему новое значение. Такая операция называется переопределением. В результате переопределения у экземпляра будет создано собственное свойство с указанным значением, которое заменит одноименное свойство, унаследованное от класса. Так, например, если в предыдущем примере лифты должны находиться на разных этажах, то есть объекты `Lift1` и `Lift2` должны иметь разные значения свойства `currentFloor`, необходимо в каждом из них переопределить наследуемое свойство `currentFloor`, создав одноименное собственное свойство, как это сделано ниже:

```
myLift1.currentFloor = 2;
myLift1.currentFloor = 10;
trace (myLift1.currentFloor); // Выводит 2
```

```
trace (myLift2.currentFloor); // Выводит 10
for (i in) {
       trace(i); // Выводит currentFloor
}
for (i in Lift2) {
       trace(i); // Выводит currentFloor
}
```

Как мы видим, теперь объекты `Lift1` и `Lift2` обладают своим собственным свойством `currentFloor`. Для того чтобы у объекта класса можно было создать собственное свойство, необходимо, чтобы в породившем его классе имелось объявление свойства с таким же именем. В противном случае при компиляции Flash выдаст сообщение об ошибке (There is no property with the name '*имя свойства*'). Добавлять новые свойства экземплярам можно только в том случае, если породивший их класс является динамическим *(см. разд. "Динамические классы" данной главы)*.

Для того чтобы определить, является ли свойство объекта наследуемым или собственным, можно воспользоваться встроенным методом класса `Object`, который называется `hasOwnProperty()`. Данный метод имеет следующий синтаксис:

```
объект.hasOwnProperty("свойство")
```

Здесь *объект* — это идентификатор экземпляра, для свойства которого выполняется проверка.

Параметр *свойство* — это строка, содержащая имя свойства.

Если проверяемое свойство объекта является собственным, метод `hasOwnProperty()` возвращает значение `true`. Если проверяемое свойство является наследуемым или если у объекта вообще отсутствует свойство с указанным именем, метод `hasOwnProperty()` возвращает значение `false`.

Проиллюстрируем применение данного метода. Пусть в классе `PropTest` имеется наследуемое свойство `prop`.

```
class PropTest {
       var prop:Number = 10;
}
```

В FLA-файле создадим экземпляр класса `PropTest` и проверим его на наличие свойства `prop`.

```
var pt:PropTest = new PropTest();
trace(pt.hasOwnProperty("prop")); /* Выводит false, так как свойство prop
- не собственное */
```

Теперь создадим собственное свойство объекта `pt`:

```
var pt:PropTest = new PropTest();
pt.prop = 20;
trace(pt.hasOwnProperty("prop")); /* Выводит true, так как свойство prop
-собственное */
```

Установка слежения за значением свойства

В некоторых ситуациях возникает необходимость в отслеживании момента, когда значение определенного свойства объекта изменяется. Обычно при этом требуется оценить новое значение свойства и в зависимости от него либо выполнить определенные действия, либо искусственно скорректировать данное значение. Для этих целей можно воспользоваться методом `watch()` класса `Object`. Данный метод позволяет установить наблюдение за изменением значения свойства объекта и при обнаружении такого изменения выполняет вызов пользовательской функции, ссылка на которую передается ему в качестве параметра. Метод `watch()` имеет следующий синтаксис:

объект.watch(*свойство*, *функция*, *данные*)

Здесь *объект* — это объект, за свойством которого должно быть установлено наблюдение.

Параметр *свойство* — это строка, содержащая имя свойства, за которым должно быть установлено наблюдение.

Параметр *функция* — это ссылка на пользовательскую функцию, которая будет вызываться всякий раз, когда значение свойства *свойство* будет изменяться.

Параметр *данные* — это необязательный параметр, который может представлять собой любые данные (число, строку, объект и т. д.). При изменении значения свойства *свойство* параметр *данные* будет автоматически передаваться функции *функция*. Если этот параметр не указан, то вместо него функции *функция* при вызове будет передаваться значение `undefined`.

Пользовательская функция *функция*, вызываемая методом `watch()`, при обнаружении факта изменения значения свойства *свойство*, за которым установлено наблюдение, при вызове получает четыре параметра и имеет следующий формат:

```
function функция (свойство, старое значение, новое значение, данные){
...
}
```

Параметр *свойство* — это строка, содержащая имя свойства, значение которого было изменено.

Параметр *старое значение* — это предыдущее значение свойства.

Параметр *новое значение* — это новое значение свойства.

Параметр *данные* — это данные, передаваемые в качестве четвертого (необязательного) параметра методу `watch()`. Если при вызове метода `watch()` ему было передано только три параметра, то пользовательской функции *функция* будет в качестве четвертого параметра передаваться значение `undefined`.

Метод `watch()` возвращает булево значение `true`, если слежение за значением свойства было успешно установлено, и значение `false` — в противном случае.

Таким образом, процедура установки наблюдения за значением свойства некоторого объекта A включает два этапа.

- Создание пользовательской функции с четырьмя формальными параметрами, которая должна быть вызвана при изменении значения данного свойства объекта A.
- Вызов метода watch() объекта A с передачей ему трех или четырех описанных выше параметров.

Важной особенностью работы этого механизма является то, что возвращаемое значение пользовательской функции автоматически присваивается свойству, за которым установлено наблюдение. Это дает возможность необходимым образом скорректировать его измененное значение. Постановка конкретной задачи может требовать просто зафиксировать факт изменения значения свойства, предотвратить это изменение или же скорректировать присвоенное свойству значение.

- Для выполнения слежения за изменением значения свойства, не требующего его корректировки, необходимо, чтобы пользовательская функция, вызываемая методом watch(), возвращала значение параметра *новое значение*.
- Для предотвращения изменения значения свойства необходимо, чтобы пользовательская функция, вызываемая методом watch(), возвращала значение параметра *старое значение*.
- Для того чтобы скорректировать новое значение свойства (например, привести его к допустимому значению), необходимо, чтобы пользовательская функция, вызываемая методом watch(), возвращала требуемое скорректированное значение.

Если пользовательская функция не имеет возвращаемого значения, то при ее вызове свойству, за которым установлено наблюдение, будет присвоено значение undefined.

Приведем пример использования метода watch() для наблюдения за изменением значения свойства energy некоторого объекта obj. Если уровень энергии оказывается ниже критического, будет выводиться соответствующее сообщение.

```
// Создаем экземпляр объекта
var obj = new Object();
// Создаем собственное свойство energy
obj.energy = 50;
// Функция, вызываемая при изменении значения свойства energy
function checkEnergy(prop, oldVal, newVal, energyLimit) {
        trace("Значение свойства " + prop + " изменено");
        trace("Старое значение: " + oldVal);
        trace("Новое значение: " + newVal);
        // Если новое значение свойства меньше критического значения,
        // передаваемого параметром energyLimit
        if (newVal < energyLimit) {
                trace("Критический уровень энергии !!!");
        }
```

```
        return (newVal);// Просто отслеживаем изменение значения
}
// Устанавливаем слежение за свойством energy
obj.watch("energy", checkEnergy, 5);
// Изменяем значение свойства
obj.energy = 4;
```

В результате выполнения этого кода в панель **Output** будет выведено следующее:

```
Значение свойства energy изменено
Старое значение: 50
Новое значение: 4
Критический уровень энергии !!!
```

Следует иметь в виду, что для каждого свойства может быть выполнена только одна установка наблюдения. Повторный вызов метода `watch()` заменит его предыдущий вызов.

Метод `watch()` нельзя использовать для отслеживания getter/setter-свойств *(о getter/setter-свойствах будет сказано далее в этой главе)*. Это связано с тем, что определение значений таких свойств выполняется только при обращении к ним, а не при фактическом изменении связанных с ними характеристик. Такой подход называется "ленивыми вычислениями" (lazy evaluation) и применяется с целью экономии машинных ресурсов и увеличения быстродействия. Ленивые вычисления позволяют вычислять значения свойств только при необходимости, а не загружать ресурсы компьютера постоянным обновлением их значений. Большинство встроенных свойств ActionScript, как, например `_x`, `_y`, `_xscale`, `_yscale` и т. д., являются getter/setter-свойствами, в связи с чем наблюдение за ними не может быть установлено с помощью метода `watch()`.

При удалении свойства, за которым было установлено наблюдение, наблюдение не прекращается. Если данное свойство будет создано вновь, наблюдение будет продолжено. Для того чтобы прекратить наблюдение за свойством, можно воспользоваться методом `unwatch()`, имеющим следующий синтаксис:

объект`.unwatch(`*свойство*`)`

Здесь *объект* — это объект, наблюдение за свойством которого должно быть прекращено.

Параметр *свойство* — это строка, содержащая имя свойства, наблюдение за которым должно быть прекращено.

Метод `unwatch()` возвращает булево значение `true`, если слежение за значением свойства было прекращено, и значение `false` — в противном случае.

Объявление конструктора

Конструктор — это функция, объявленная внутри определения класса, которая используется для инициализации собственных свойств создаваемых объектов (то есть

свойств, принадлежащих самим объектам). Конструктор в сочетании с оператором `new` применяется для создания новых экземпляров класса. Код функции конструктора выполняется при создании каждого нового экземпляра класса. Поэтому с помощью конструктора можно выполнить определенные действия с экземпляром непосредственно в момент его создания. Конструктор может принимать параметры, на основании которых будет выполняться инициализация свойств объекта.

Объявление конструктора очень похоже на объявление обычной функции, однако имеет две характерные особенности:

- имя конструктора должно совпадать с именем класса;
- конструктор не должен возвращать никакого значения.

Объявление конструктора обычно выполняется после объявления свойств (хотя это не является жестким правилом). В принципе, объявление конструктора не является обязательным, однако конструктор рекомендуется объявлять явно, даже если он не выполняет никаких действий. В случае отсутствия объявления конструктора Flash автоматически создаст пустой конструктор во время компиляции.

Объявление конструктора имеет следующий синтаксис:

```
function Имя (п1:тип, п2:тип, ..., пN:тип){
код конструктора;
}
```

Здесь *Имя* — это сокращенное имя класса (то есть имя без указания пакета, к которому относится класс).

Параметры *п1*, *п2*, ..., *пN* — в общем случае необязательны, хотя их использование позволяет сразу при создании объекта настроить определенные параметры его функционирования.

С помощью конструктора можно осуществлять определение собственных свойств объектов, порождаемых классом. Для этого необходимо в теле конструктора выполнить присваивание значения свойствам, имена которых совпадают с именами свойств, объявленных в определении класса. Зачастую значения, инициализирующие собственные свойства объекта, передаются конструктору в качестве параметров. В результате присваивания в конструкторе значений свойствам при создании объекта он автоматически наделяется собственными копиями данных свойств. Инициализировать собственные свойства объекта с помощью конструктора можно только в том случае, если они объявлены в классе. В противном случае возникнет ошибка компиляции. Продемонстрируем инициализацию свойств объекта на простом примере:

```
class Lift{
        // Объявление свойства
        var currentFloor:Number = 3;
        ...
        // Объявление конструктора
        function Lift(floor:Number){
        // Инициализируем собственное свойство currentFloor объекта
```

```
        currentFloor = floor;
        }
        ...
}
```

Здесь конструктор в качестве параметра принимает значение, на основании которого устанавливается значение собственного свойства объекта, определяющего, на каком этаже находится лифт, представленный новым объектом `Lift`. Следует отметить, что объект, создаваемый с помощью конструктора, входит в его область видимости. Это дает возможность обращаться из конструктора к членам класса без применения какой-либо адресации или использования идентификатора `this`. Тем не менее если из кода конструктора нужно явно сослаться на создаваемый им объект, для этого можно использовать идентификатор `this`. Проверим созданный нами конструктор в действии. Для этого в FLA-файле создадим следующий код:

```
var myLift:Lift = new Lift(10);
trace(myLift.currentFloor); /* Выводит 10 - значение собственного
свойства currentFloor объекта myLift */
```

Если же мы создадим объект `Lift`, не передав конструктору число в качестве параметра, результат будет иной:

```
var myLift2:Lift = new Lift();
trace(myLift2.currentFloor); /* Выводит undefined. Для данного объекта
определено собственное свойство currentFloor, однако оно имеет значение
undefined. */
```

Определить значение свойства `currentFloor` для объекта `myLift2` в этом случае можно уже после его создания:

```
myLift2.currentFloor = 5; /* Присваиваем значение собственному свойству
объекта */
trace(myLift2.currentFloor); // Выводит 5
```

Блок кода конструктора может включать не только выражения присваивания, инициализирующие свойства объекта, но и любые другие инструкции. Так, например, при каждом создании нового объекта `Lift` в панель **Output** будет выводиться соответствующее сообщение:

```
class Lift {
        ...
        function Lift (){
        trace (Создан новый экземпляр Lift);
        }
        ...
}
```

В процессе компиляции Flash помещает функцию-конструктор в объект `_global` в качестве обычного свойства. В этом можно легко убедиться путем анализа списка глобальных переменных, выполнив в среде тестирования команду **Debug>List Vari-**

ables. Это, в частности, означает, что конструктор обладает глобальной областью видимости. Наследуемые свойства и методы, объявленные внутри определения класса, при компиляции сохраняются в специальном свойстве функции-конструктора, которое называется `prototype` и создается автоматически. Это свойство конструктора является скрытым (его нельзя непосредственно обнаружить с помощью цикла `for...in`), однако его можно "открыть" с помощью функции `ASSetPropFlags()`. Свойство `prototype` представляет собой объект, используемый во внутренней реализации механизма наследования. В ActionScript 1.0 реализация наследования осуществлялось именно с непосредственным применением свойства `prototype`. В ActionScript 2.0 появился более комфортный и лучше формализованный способ организации наследования, хотя его внутренний механизм не претерпел никаких изменений. При использовании стиля программирования ActionScript 2.0 разработчику не требуется выполнять обращение к объекту `prototype` конструктора для того, чтобы обратиться к наследуемому свойству или вызвать наследуемый метод. Тем не менее приведем фрагмент кода, позволяющий выполнить это именно с использованием `prototype`.

```
// В AS-файле
class Lift {
        var currentFloor = 2;
        var speed = 3500;
...
}
// В FLA-документе
var myLift:Lift = new Lift();
// Снимаем защиту со свойства prototype конструктора Lift
ASSetPropFlags(Lift.prototype, null, null, true);
// Перечисляем свойства объекта prototype
for (i in Lift.prototype) {
       trace(i + " = " + Lift.prototype[i]);
}
```

В панель **Output** будет выведено:

```
speed = 3500
currentFloor = 2
...
```

Объявление методов

Методы — это функции, определенные в пределах класса, позволяющие управлять состоянием объекта и получать доступ к хранящимся в нем данным. Как и свойства, методы определяются в классе и могут вызываться для его экземпляров. Свойства предоставляют возможность для хранения данных, а методы обеспечивают функциональность объекта, изменяя значения свойств в соответствии с заложенным ал-

горитмом и/или возвращая данные, полученные в результате обработки их исходных значений.

Объявление методов формально выполняется так же, как и объявление обычных функций. Поскольку объявление метода выполняется в пределах определения класса, он автоматически связывается с объектом, создаваемым на основе этого класса. Как и в случае с конструктором, из тела метода можно непосредственно обращаться к членам класса (свойствам и другим методам). Идентификатор `this`, используемый в теле метода, возвращает ссылку на экземпляр класса, из которого выполняется вызов данного метода. Метод может принимать параметры и иметь возвращаемое значение. При объявлении метода используется следующий синтаксис:

```
function имя метода (п1:тип, п2:тип, ..., пN:тип):тип{
блок кода
...
}
```

Строгая типизация параметров и возвращаемого методом значения не является обязательной, однако ее применение позволяет идентифицировать ошибки несоответствия типа на этапе компиляции.

Рассмотрим простой пример создания класса `Lift` с двумя методами, выполняющими перемещение лифта на этаж вверх и вниз.

```
class Lift {
        // Текущий этаж
        var currentFloor:Number;
        // Максимальное число этажей
        var maxFloor:Number = 10;
        // Объявление конструктора
        function Lift(floor:Number){
        currentFloor = floor;
        }
        // Метод, поднимающий лифт на этаж вверх
        function floorUp(){
        if(currentFloor < maxFloor){
                currentFloor++;
        }
        return currentFloor;
        }
        // Метод, опускающий лифт на этаж вниз
        function floorDown(){
        if(currentFloor > 1){
                currentFloor--;
        }
```

```
        return currentFloor;
        }
}
```

Проверим работу методов. Для этого в FLA-файле создадим следующий код:

```
var myLift:Lift = new Lift (5);
// Определяем текущий этаж
trace(myLift.currentFloor); // Выводит 5
// Поднимаем лифт на один этаж
trace(myLift.floorUp()); // Выводит 6
// Опускаем лифт на один этаж
trace(myLift.floorDown()); // Выводит 5
```

В теле метода так же, как и в теле обычной функции, можно создавать локальные переменные с использованием ключевого слова `var`. Такие переменные удаляются из памяти после окончания работы метода. Следует помнить, что при необходимости создания переменных в теле метода это нужно обязательно выполнять явно, то есть с помощью `var`. Если переменная будет создана неявно, Flash проверит, имеется ли в классе свойство с таким именем, и в случае его отсутствия выдаст ошибку (There is no property with the name '*имя свойства*'). Так, например, при использовании в теле метода цикла `for` переменную-итератор в его заголовке необходимо инициализировать с использованием ключевого слова `var` или же выполнять в классе объявление специального свойства.

Регулирование доступа к членам класса

По умолчанию все члены класса (свойства и методы) являются доступными из экземпляров этого класса. Это означает, что при создании объекта на основе определенного класса данный объект может получить возможность обращаться к свойствам класса и вызвать любые его методы. Такое поведение нарушает одну из основных концепций объектно-ориентированного программирования — инкапсуляцию. Действительно, зачастую бывает весьма нежелательно, чтобы объект имел доступ ко всем членам класса, поскольку некоторые из них могут применяться для реализации внутренней функциональности класса и их "насильственное" изменение или использование извне может привести к некорректной работе класса или даже к утрате его работоспособности. Для того чтобы скрыть определенные члены класса от кода, расположенного вне определения класса, можно при их объявлении ввести ограничения внешнего доступа.

Регулирование доступа к членам класса осуществляется с помощью ключевых слов `public` и `private`, которые указываются перед объявлением члена.

- Ключевое слово `public` указывает на то, что доступ к члену может осуществляться извне определения класса. Такой член называется открытым (public). Как уже было сказано, все члены класса по умолчанию являются открытыми, поэтому применение ключевого слова `public` не является необходимым. Тем не менее его использование позволяет сделать код более читабельным и понятным.

- Ключевое слово `private` указывает на то, что доступ к члену класса может быть получен только со стороны других членов класса, то есть из его определения (или со стороны членов подклассов при использовании наследования). Такой член называется защищенным (private). Имена защищенных свойств обычно принято начинать с символа нижнего подчеркивания. Это позволяет легче ориентироваться в программном коде.

Далее приводится пример использования атрибутов доступа `public` и `private`.

```
class Lift{
       public var currentFloor:Number; // Открытое свойство
       private var _currentFloor:Number = 1; // Защищенное свойство
       ...
       public function elevate(floor:Number):Void{ // Открытый метод
       ...
       }
       private function moveLift():Void{// Защищенный метод
       ...
       }
       ...
}
```

Использование защищенных членов позволяет скрыть часть внутренней функциональности класса и исключить возможность внешнего вмешательства в механизм его работы. Попытка обратиться к защищенному свойству класса или вызвать защищенный метод из экземпляра класса приведет к возникновению ошибки компиляции и выводу сообщения 'The member is private and cannot be accessed'. Тем не менее защита доступа к членам класса не является совершенной и имеет два существенных недостатка.

- Регулирование доступа к членам класса не выполняется, если при создании экземпляра не использовалась строгая типизация.
- Доступ к защищенному члену класса может быть получен при обращении к свойству или вызове метода с помощью оператора доступа к элементу массива `[]`.

Проиллюстрируем это на примере. В FLA-файле создадим экземпляр приведенного ранее класса `Lift`:

```
var myLift:Lift = new Lift();
```

Выполним обращение к защищенному свойству:

```
trace (myLift._currentFloor); // Ошибка!
```

Теперь выполним обращение к этому же свойству, но с использованием оператора `[]`:

```
trace (myLift["_currentFloor"]); // Возвращает 1. Ошибки не возникает
```

Если ссылка на созданный экземпляр класса будет сохранена в нестрого типизированной переменной, доступ к защищенному члену класса может быть получен обычным образом:

```
var myLift = new Lift();
trace (myLift._currentFloor); // Возвращает 1. Ошибки не возникает
```

Описанные особенности еще раз доказывают преимущество использования строгой типизации.

Приведем пример создания класса, реализующего упрощенную модель лифта.

```
class Lift {

        // ********** ОБЪЯВЛНИЕ СВОЙСТВ *************
        // Текущий этаж (по умолчанию - первый)
        private var _currentFloor:Number = 1;
        // Этаж, к которому нужно переместить лифт
        private var _floor:Number;
        // ID интервала, с помощью которого выполняется перемещение лифта
        private var _moveInterval:Number;
        // Свойство, определяющее, за сколько миллисекунд лифт будет
        // перемещаться между двумя соседними этажами
        private var _speed:Number = 1000;

        // ********** ОБЪЯВЛНИЕ КОНСТРУКТОРА *************
        // Конструктор, ему в качестве параметра передается этаж, на
        // котором лифт находится в момент создания объекта Lift.
        function Lift(currentFloor:Number) {
                _currentFloor = currentFloor;
                trace("Класс Lift загружен");
        }
```

В данном классе объявлено два метода, один из которых является открытым, другой — защищенным. Защищенный метод `moveLift()` содержит код, перемещающий лифт с его текущего этажа по направлению к этажу, номер которого хранится в защищенном свойстве `_floor`. Перемещение реализуется путем изменения значения свойства `_currentFloor`. Для простоты здесь смещение лифта выполняется с шагом, равным одному этажу. Вызов открытого метода `elevate()` запустит процесс перемещения лифта к заданному этажу путем установки интервала, осуществляющего периодический вызов защищенного метода `moveLift()`. Метод `moveLift()` содержит проверку текущего положения лифта и при достижении заданного этажа очищает интервал, установленный методом `elevate()`, в результате чего лифт останавливается.

```
        // ********** ОБЪЯВЛНИЕ МЕТОДОВ *************
        // Метод, инициирующий перемещение лифта на указанный этаж
```

```
        public function elevate (floor:Number){
                // Присваиваем свойству _floor номер целевого этажа
                _floor = floor;
                _moveInterval = setInterval(this, "moveLift", _speed);
        }
        // Метод, используемый для внутренней реализации перемещения лифта
        private function moveLift():Void {
                // Вычисляем, сколько этажей должен пройти лифт, и
                // сохраняем полученное значение в локальной переменной
                var distance:Number = _floor - _currentFloor;
                        // Если лифт уже находится на этаже floor
                if(distance == 0){
                        trace("Лифт прибыл на " + _currentFloor + " этаж");
                        // Прекращаем его перемещение
                        clearInterval(_moveInterval);
                        return
                }
                trace("Лифт на " + _currentFloor + " этаже");
                // Если лифт находится ниже этажа _floor, поднимаем его на
                // один этаж
                if (distance > 0) {
                        _currentFloor++;
                }
                // Если лифт находится выше этажа _floor, опускаем его на
                // один этаж
                else if (distance < 0) {
                        _currentFloor--;
                }
        }
}
```

Теперь в FLA-файле поместим в кадр следующий код:

```
// Создаем первый лифт на 1 этаже
var myLift1:Lift = new Lift (1);
// Создаем второй лифт на 10 этаже
var myLift2:Lift = new Lift (10);
// Заставляем первый лифт подниматься с 1-го этажа на десятый
myLift1.elevate(10); // Выводит ...1..., ...2..., ...
// Заставляем второй лифт опускаться с 10-го этажа на первый
myLift2.elevate(1); // Выводит ...10..., ...9..., ...
```

Статические члены класса

До сих пор мы имели дело с членами класса, которые были доступны из каждого создаваемого на основе данного класса объекта. Такие члены называются *членами экземпляра* (instance members). Так, например, объявленное в классе `Lift` свойство `_currentFloor` при создании каждого очередного экземпляра наследовалось этим экземпляром или заменялось собственным свойством, что давало возможность разным лифтам одновременно находиться на разных этажах. Обращение к членам экземпляров выполняется хорошо знакомым нам способом с помощью ссылки на экземпляр и оператора точки.

Иногда возникает необходимость в создании члена, который не должен копироваться в каждый экземпляр, а, напротив, принадлежать самому классу. Такие члены называются *статическими* (static). Формально статические члены представляют собой свойства конструктора класса (поскольку конструктор является функцией, статические члены — это свойства объекта `Function`). Статический член класса существует в единственном числе, и доступ к нему осуществляется не из экземпляра, а непосредственно из класса, то есть с помощью конструктора. Мы уже неоднократно сталкивались со статическими членами встроенных классов ActionScript. К их числу относятся методы и свойства классов `Math`, `Mouse`, `Stage` и др.

Потребность в создании статических членов возникает при необходимости сохранения или получения данных, относящихся сразу ко всем объектам класса. Так, например, в классе `Lift` можно определить статическое свойство `speed`, устанавливающее единую скорость для всех лифтов, порожденных данным классом. Сделать это свойство статичным имеет смысл в том случае, если данный класс реализует конкретную модификацию лифтов (например, лифты в жилом многоэтажном доме). Как правило, однотипные лифты, установленные в одном здании, обладают одинаковой номинальной скоростью.

Для создания статичного члена класса используется ключевое слово `static`, которое указывается перед объявлением свойства или метода. Статические свойства и методы могут быть как открытыми (доступными извне), так и закрытыми (доступными только членам класса). При объявлении статичных свойств и методов (необязательный) атрибут, определяющий тип доступа к члену, предшествует атрибуту `static`. В именах статических свойств принято использовать прописные буквы.

```
private static var MIN_SPEED:Number;
public static var SPEED:Number;
public static function getMinSpeed(){
...;
}
private static function getOverloadLimits(){
...;
}
```

В объявлении статических свойств и методов не допускается использование ссылок на нестатические члены класса, то есть на члены экземпляров.

```
public var a:Number = 10;
static var B:Number = a; // Ошибка!
```

При попытке присвоить статическому свойству выражение, в состав которого входит ссылка на нестатический член класса, или при обращении к нестатическому члену класса из тела статического метода Flash выдаст сообщение об ошибке (Instance variables and functions may not be used to initialize static variables).

Приведем пример создания статического свойства `_pLifts`, в котором хранится массив, содержащий ссылки на все созданные экземпляры класса.

```
class Lift {
       // Массив для хранения ссылок на все созданные объекты Lift
       private static var _pLifts:Array = new Array();
       ...
       // Объявление конструктора
       function Lift() {
              // Добавляем ссылку на текущий объект в массив _pLifts при
              // создании каждого нового объекта Lift
              _pLifts.push(this);
       ...
       }
       ...
}
```

С помощью массива `_pLifts` можно из класса контролировать все порожденные им объекты. Это оказывается необходимо при создании методов, реализующих взаимодействие между экземплярами класса. Так, например, при вызове лифта на этаж необходимо отправить туда ближайший свободный лифт. Для этого метод, осуществляющий вызов лифта, должен оценить удаленность каждого лифта от заданного этажа. Код, помещенный в конструктор, выполняется при каждом создании экземпляра класса, поэтому, сохраняя ссылку на текущий объект в массиве `_pLifts`, представляющем собой статическое свойство класса, мы получаем возможность обратиться к любому экземпляру этого класса из кода его метода.

Важным отличием статических членов класса от членов экземпляра является то, что статические члены перечисляются в цикле `for...in`, в то время как наследуемые члены экземпляра этим циклом не перечисляются. Покажем это на примере:

```
class Lift{
       static var SPEED = 10;
       static var MIN_SPEED = 300;
       static var MAX_SPEED = 5000;
...
}
```

Перечислим статические члены класса `Lift`:

```
for (i in Lift) {
        trace(i); // Выводит MAX_SPEED MIN_SPEED SPEED
}
```

Теперь удалим атрибуты `static` в определении свойств:

```
class Lift{
        var speed = 10;
        var min_speed = 300;
        var max_speed = 5000;
        ...
}
```

В FLA-файле создадим экземпляр класса и попробуем перечислить его свойства:

```
var myLift:Lift = new Lift();
for (i in myLift) {
        trace(i); // Ничего не выводится!
}
```

Описанная особенность связана с тем, что статические члены класса сохраняются как свойства конструктора, в то время как нестатические наследуемые члены класса сохраняются в автоматически создаваемом объекте `prototype`, представляющем собой скрытое свойство конструктора.

Организация считывания и установки значений свойств. Определение *get/set*-методов

Как было показано, обращение к открытым свойствам класса может быть выполнено непосредственно по имени из кода, расположенного вне определения класса. Это плохо согласуется с принципом инкапсуляции и делает внешний код, использующий возможности класса, зависимым от названий переменных класса. В объектно-ориентированном программировании хорошим тоном считается, что обращение к свойствам класса извне должно выполняться с помощью специальных методов. Существует две разновидности таких методов:

- `get`-методы предназначены для извлечения значения свойства. Эти методы не принимают параметров и обязательно возвращают значение;
- `set`-методы предназначены для задания значения свойства. Эти методы должны принимать параметр и никогда не возвращают значения.

Методы, обеспечивающие доступ к свойствам класса, можно объявить точно так же, как обычные методы класса.

Следующий пример демонстрирует объявление методов для установки и извлечения значения защищенного свойства _currentFloor.

```
class Lift {
        private var _currentFloor:Number;
        ...
        // Метод для извлечения значения _currentFloor
        public function getCurrentFloor():Number{
        return _currentFloor;
        }
        // Метод для задания значения _currentFloor
        public function setCurrentFloor(floor:Number):Void{
        _currentFloor = floor;
        }
        ...
}
```

Теперь для обращения к свойству _currentFloor объекта Lift в FLA-файле используется следующий код:

```
var myLift:Lift = new Lift();
// Устанавливаем значение свойства
myLift.setCurrentFloor(5);
trace(myLift.getCurrentFloor()); // Выводит 5
```

Единственным неудобством описанного подхода является то, что для задания и извлечения значения свойства приходится использовать разные методы.

ActionScript предоставляет более удобную возможность объявления get/set-методов таким образом, чтобы доступ к ним из внешнего кода осуществлялся, как к свойствам объекта. Применение таких методов внешне очень похоже на непосредственное обращение к свойству, однако get/set-методы могут обладать собственной функциональностью и, наряду с обращением к свойству, выполнять определенные действия. Так, например, set-метод, задающий значение свойству _currentFloor, может проверять, является ли переданное ему в качестве параметра значение допустимым (например, есть ли в доме этаж с таким номером). Для объявления get/set-методов используются специальные ключевые слова get и set.

Объявление get-метода имеет следующий синтаксис:

```
function get имя ():тип{
блок кода
...
}
```

Объявление `set`-метода имеет следующий синтаксис:

```
function set имя (значение:тип):Void{
блок кода
...
}
```

Здесь *имя* — это идентификатор, который будет использоваться для обращения к свойству из внешнего кода. Этот идентификатор должен быть одинаковым в объявлении `get`-метода и `set`-метода. В этом случае, используя идентификатор *имя* как имя свойства, можно из внешнего кода осуществлять извлечение и присваивание значения внутреннего свойства класса. Обращение к свойству объекта с именем *имя* приведет к вызову соответствующего `get`-метода. Присваивание значения свойству с именем *имя* приведет к вызову соответствующего `set`-метода, которому в качестве параметра будет передано присваиваемое значение.

```
class Lift {
       // Создаем защищенное свойство
       private var _currentFloor:Number;
       ...
       // Метод для извлечения значения защищенного свойства
       functon get currentFloor():Number{
       return _currentFloor;
       }
       // Метод для задания значения защищенному свойству
       function set currentFloor(value:Number):Void{
       _currentFloor = value;
       }
       ...
}
```

В FLA-документе обращение к свойству `_currentFloor` может быть выполнено следующим образом:

```
var myLift:Lift = new Lift();
myLift.currentFloor = 6;
trace(myLift.currentFloor); // Выводит 6
```

Таким образом, мы видим, что `currentFloor` здесь используется в качестве свойства объекта `myLift`, инициируя вызов соответствующего `get`- или `set`-метода. Это с одной стороны обеспечивает удобство обращения к внутренним свойствам объекта, с другой — не нарушает инкапсуляцию.

Допускается использование статических `get/set`-методов. В этом случае при их объявлении необходимо использовать атрибут `static`. Статические `get/set`-методы обеспечивают возможность внешнего доступа к защищенным статическим свойствам класса.

Применение get/set-методов имеет два важных преимущества.

- Они дают возможность создания свойств, доступных только для чтения или только для записи.
- Они позволяют возвращать результат, являющийся результатом расчета, а не значением реального свойства класса, и накладывать ограничения на значения, задаваемые пользователем.

Для того чтобы создать свойство, доступное только для чтения, необходимо объявить его в классе, используя атрибут private, и создать только get-метод, позволяющий извлечь его значение. Если соответствующий set-метод создан не будет, присвоить свойству значение из внешнего кода будет нельзя (по крайней мере, используя set-свойство объекта). Рассмотрим простой пример:

```
class Lift{
        // Скорость лифта
        private var _speed:Number = 1000;
        // Нижнее граничное значение скорости, только для чтения
        private static var MIN_SPEED = 100;
        // Верхнее граничное значение скорости, только для чтения
        private static var MAX_SPEED = 5000;
        ...
        // Определение значения скорости
        public function get speed ():Number{
        return _speed;
        }
        // Установка значения скорости
        public function set speed (value:Number){
                // Вводим контроль заданного пользователем значения
                // скорости. Если оно будет выходить за допустимые границы,
                // присваиваем ему граничное значение
                _speed = Math.max(MIN_SPEED, Math.min(value, MAX_SPEED));
        }
        // Статический метод, позволяющий определить граничные значения
        // скорости. Поскольку лимит скорости действует для всех
        // экземпляров класса, данный get-метод объявлен как статический
        public static function get speedLimits ():Array{
                return [MIN_SPEED, MAX_SPEED];
        }
}
```

В приведенном классе статические свойства MIN_SPEED и MAX_SPEED имеют доступ только для чтения, поскольку для них не определен соответствующий set-метод.

Обращение к защищенному свойству _speed из внешнего кода осуществляется с помощью get/set-свойства speed.

```
var myLift = new Lift();
// Выводим граничные значения скорости, используя статическое
// get-свойство
trace(Lift.speedLimits); // Выводит 100,5000
// Задаем недопустимое значение скорости
myLift.speed = 6000;
trace(myLift.speed); // Выводит 5000
// Задаем допустимое значение скорости
myLift.speed = 2500;
trace(myLift.speed);// Выводит 2500
```

Обратите внимание, что хотя статические свойства MIN_SPEED и MAX_SPEED являются защищенными и для них не определен set-метод, тем не менее, их значение может быть изменено извне с помощью оператора []:

```
// Изменяем значение внутреннего свойства класса
Lift["MAX_SPEED"] = 7000;
// Выводим граничные значения скорости, используя статическое
// get-свойство
trace(Lift.speedLimits); // Выводит 100, 7000
// Задаем "недопустимое" значение скорости
myLift.speed = 6000;
trace(myLift.speed); /* Выводит 6000, так как внутреннее свойство было
изменено извне */
```

Динамические классы

Динамический (dynamic) класс — это класс, экземпляры и конструктор которого могут обладать свойствами и методами, не объявленными в определении класса.

По умолчанию все члены класса являются фиксированными. Это означает, что экземпляр класса и конструктор не могут содержать свойств, которые не были объявлены в определении класса. Попытка присвоить экземпляру класса свойство, отсутствующее в определении класса, приведет к ошибке компиляции. Так, если в определении класса Lift отсутствует объявление свойства currentFloor и статического свойства SPEED, следующий код в FLA-файле вызовет ошибку:

```
var myLift:Lift = new Lift();
myLift.currentFloor = 5; /* Ошибка, свойство экземпляра не объявлено
в определении класса */
Lift.SPEED = 2000; /* Ошибка, статическое свойство не объявлено
в определении класса */
```

При обнаружении подобной ошибки в панель **Output** выводится сообщение There is no property with the name '*имя свойства*'.

Примечание

Обращение к несуществующему свойству или вызов несуществующего метода объекта вызывают ошибку компиляции только при использовании строгой типизации. Если переменная, в которой сохраняется экземпляр объекта, не является строго типизированной, проверка наличия в классе объекта свойства или метода не выполняется. Это является еще одним аргументом в пользу применения строгой типизации. Тем не менее даже при использовании строгой типизации имеется возможность выполнить обращение к отсутствующему в определении класса свойству (методу), воспользовавшись оператором доступа к элементу массива `[]`.

Для того чтобы иметь возможность создавать произвольные свойства экземпляров и конструктора, необходимо объявить класс как динамический. Для этого в объявлении класса используется ключевое слово `dynamic`.

```
dynamic class имя класса {
...
}
```

Если класс объявлен с атрибутом `dynamic`, для его экземпляров можно создавать новые свойства и методы, не объявленные в определении класса:

```
myLift.currentFloor = 5; // Создано новое свойство экземпляра класса Lift
Lift.SPEED = 2000; // Создано новое свойство конструктора
myLift.traceMessage = function(mes){//Создан новый метод объекта Lift
trace(mes);
}
myLift.traceMessage("Привет, Flash!"); // Выводит Привет, Flash!
```

Свойства экземпляра динамического класса могут быть созданы прямо в конструкторе класса без их предварительного объявления. Выполнение подобной процедуры в классе, объявленном без применения атрибута `dynamic`, вызовет ошибку. Следует иметь в виду, что при инициализации собственных свойств экземпляра динамического класса в конструкторе необходимо использовать идентификатор `this`:

```
dynamic class Lift {
       function Lift() {
              this.currentFloor = 1;
              this.isBusy = false;
       }
}
```

При инициализации собственных свойств в конструкторе динамического класса можно не объявлять в этом классе одноименных наследуемых свойств. Однако если инициализация собственных свойств в конструкторе класса выполняется без применения идентификатора `this`, при создании экземпляра класса свойства будут поме-

щены не в него, а на _level0. Поэтому нужно либо объявлять в классе наследуемые свойства и затем инициализировать их для каждого экземпляра в конструкторе класса, либо, не объявляя наследуемых свойств, инициализировать собственные свойства в конструкторе с применением идентификатора this.

При адресации свойств экземпляра, созданных в конструкторе, из других членов класса также необходимо использовать идентификатор this.

Необходимо иметь в виду, что подкласс динамического класса также автоматически является динамическим. Исключение составляет только встроенный класс MovieClip. Класс MovieClip является динамическим, тем не менее, поскольку его очень часто используют в качестве суперкласса производных пользовательских классов, подклассы класса MovieClip по умолчанию не являются динамическими. Это дает возможность обеспечения более строгого контроля типов для пользовательских подклассов класса MovieClip, осуществляемого при компиляции. Если требуется создать динамический подкласс класса MovieClip, необходимо в его объявлении использовать атрибут dynamic.

Суперклассы и подклассы

Ранее мы выяснили, что объекты, порождаемые классом, наследуют члены, объявленные в определении этого класса. Это дает возможность использования общих свойств и методов любым количеством объектов без необходимости их копирования индивидуально в каждый экземпляр класса. Подобный подход может быть использован и для реализации наследования членов между разными классами, что позволяет существенно повысить эффективность создания комплексных программ.

При реализации наследования классы находятся в отношении "родитель — потомок", все члены класса-родителя являются доступными для членов дочернего класса и, соответственно, могут быть доступны для порождаемых им объектов. Родительский класс, который предоставляет свои свойства и методы называется *суперклассом* (superclass). Дочерний класс, наследующий свойства и методы суперкласса, называется *подклассом* (subclass). Поведение и данные, связанные с дочерним классом, всегда расширяют возможности суперкласса. Подкласс располагает всеми свойствами и методами суперкласса и при этом, как правило, содержит свои собственные свойства и методы, специфичные только для него. Таким образом, подкласс представляет собой более специализированную форму суперкласса. При моделировании определенной системы базовая функциональность, свойственная всем элементам этой системы, реализуется в суперклассе. Между тем каждый элемент системы может иметь свои собственные, характерные только для него, особенности проявления функциональности. Такой элемент может быть представлен подклассом, который, с одной стороны, будет наследовать всю базовую функциональность системы из суперкласса, с другой стороны, будет содержать функциональность, специфичную лично для него. Вернемся к нашему примеру с лифтом. Каждый лифт обладает такой базовой характеристикой, как возможность перемещаться к указанному этажу. Соответствующий метод, очевидно, имеет смысл определить в суперклассе так, чтобы он был автоматически доступен всем подклассам, представляющим разные типы лифтов: пассажирский, грузовой и т. д. В то же время в подклассе, представляющем

пассажирский лифт, может быть определен метод, позволяющий вызвать лифт с любого этажа.

Примечание

Подкласс наследует все члены суперкласса: как открытые, так и защищенные, поэтому члены подкласса всегда могут обращаться к членам суперкласса вне зависимости от заданного типа доступа.

Подкласс, в свою очередь, сам может являться суперклассом для класса, наследующего его данные и функциональное поведение. Причем наследование всегда транзитивно, то есть подкласс может наследовать не только от своего непосредственного суперкласса, но и от суперкласса, отстоящего от него на несколько уровней. Так, например, если класс `Lift` является суперклассом по отношению к классу `PassangerLift`, который в свою очередь является суперклассом для класса `OtisLift`, то класс `OtisLift` наследует все свойства и методы классов `PassangerLift` и `Lift` (и при этом может определять свои собственные, присущие только ему!).

Для того чтобы класс мог наследовать все члены другого класса, его необходимо объявить как подкласс. Для этого в объявлении класса используется ключевое слово `extends`.

```
class имя подкласса extends имя суперкласса {
...
}
```

Пользовательский класс может расширять встроенный класс ActionScript (то есть являться его подклассом). Это позволяет использовать возможности встроенного класса объектами, порожденными пользовательским классом.

Примечание

В ActionScript не допускается множественное наследование. Это означает, что у каждого подкласса может быть только один непосредственный суперкласс. Так, класс `C` может расширять класс `B`, который, в свою очередь, может расширять класс `A`. Однако класс `C` не может одновременно расширять и класс `B`, и класс `A`.

```
// Разрешается
class C extends B {}
class B extends A {}
// Не разрешается
class C extends A, B {}
```

Подкласс имеет доступ ко всем членам суперкласса: как открытым, так и защищенным, поэтому, если поведение наследуемого члена суперкласса в подклассе не должно быть модифицировано, объявлять его не нужно. Рассмотрим простой пример.

```
// Суперкласс
class A {
       public var prop:String = "Свойство суперкласса";
```

```
        public function showMessage(){
        trace("Метод суперкласса");
        }
}
// Подкласс
class B extends A {
}
// Код в FLA-документе
var myB:B = new B();
trace(myB.prop); // Выводит "Свойство суперкласса"
myB.showMessage(); // Выводит "Метод суперкласса"
```

Таким образом, мы видим, что свойства и методы, объявленные в суперклассе, доступны объектам, порожденным его подклассом. Обращение к свойствам и методам суперкласса также может быть выполнено из определения подкласса.

```
// Подкласс
class B extends A {
        public function displayProp(){
        trace (prop); // Выведет "Свойство суперкласса"
        }
}
```

Поскольку объявление свойства `prop` в классе `B` отсутствует, при использовании этого свойства в теле метода `displayProp()` выполняется обращение к соответствующему свойству суперкласса. Если бы в суперклассе также не было объявлено свойства с таким именем, интерпретатор выполнил бы обращение к суперклассу класса `A` (если бы таковой имелся). Если ни в одном из элементов этой цепочки не было бы обнаружено свойство `prop`, Flash выдал бы сообщение об ошибке (There is no property with the name '*имя свойства*).

Зачастую возникает необходимость в частичной или полной модификации наследуемого свойства или метода в подклассе с целью реализации специфической для данного подкласса функциональности. Так, например, в суперклассе `Lift` может содержаться объявление метода `call()`, выполняющего вызов свободного лифта на заданный этаж. В подклассе этого класса может потребоваться создать метод с таким же названием, который будет вызывать лифт не только в случае, если он свободен, но и в случае, если он занят, но находится над указанным этажом и движется вниз. Это называется *перекрытием* (overriding). Если в подклассе содержится объявление свойства или метода, имя которого совпадает с именем свойства или метода суперкласса, говорят, что элемент подкласса перекрывает соответствующий элемент суперкласса. При разрешении ссылок на члены класса интерпретатор в первую очередь проверяет, имеется ли соответствующий элемент в подклассе. Проверка наличия этого элемента в суперклассе осуществляется только в том случае, если в подклассе элемент не найден. Поэтому для того чтобы в подклассе изменить поведение

метода (или значение свойства), наследуемого из суперкласса, необходимо просто объявить метод (или свойство) с таким же именем. Продемонстрируем это на примере:

```
// Суперкласс
class A {
        public var prop:String = "Свойство суперкласса";
        public function showMessage(){
        trace("Метод суперкласса");
        }
}
// Подкласс
class B extends A {
        // Перекрытие метода showMessage() суперкласса
        public function showMessage(){
        trace("Метод подкласса");
        }
}
// Код в FLA-файле
var myB:B = new B();
trace(myB.prop); /* Выводит "Свойство суперкласса", так как свойство prop
наследуется из суперкласса */
myB.showMessage(); /* Выводит "Метод подкласса", так как метод
showMessage() определен в подклассе, то есть имеет место перекрытие */
```

Приведенный пример иллюстрирует полное перекрытие методов суперкласса. При полном перекрытии реализация метода, объявленного в суперклассе, полностью игнорируется.

В некоторых ситуациях необходимо создать член подкласса, который, обладая собственной функциональностью (специфичной только для подкласса), также реализует функциональность соответствующего члена суперкласса. Для того чтобы это сделать, нужно из члена подкласса непосредственно обратиться к суперклассу и вызвать его метод. Такое обращение осуществляется с использованием оператора `super`. При вызове метода суперкласса из тела метода подкласса используется следующий синтаксис оператора `super`:

`super.`*`метод суперкласса`* `(`*`п1`*`, `*`п2`*`, ..., `*`пN`*`)`

Здесь *`метод суперкласса`* — это вызываемый метод суперкласса.

Параметры *`п1`*, *`п2`*, ..., *`пN`* — это фактические параметры, передаваемые методу суперкласса при его вызове. Рассмотрим пример, иллюстрирующий возможность вызова метода суперкласса из тела метода подкласса.

```
class A {
        public function showMessage(){
```

```
        trace("Метод суперкласса");
        }
}
class B extends A {
        // Определяем метод подкласса, перекрывающий метод суперкласса
        public function showMessage(){
                trace("Метод подкласса");
                // Вызываем метод суперкласса
                super.showMessage();
        }
}
var myB:B = new B();
myB.showMessage(); // Выводит "Метод подкласса" "Метод суперкласса"
```

Оператор `super` также может быть использован в теле конструктора подкласса для вызова конструктора суперкласса. Это позволяет наряду с инициализацией собственных свойств, объявленных в подклассе, осуществить инициализацию свойств, объявленных в суперклассе. При вызове конструктора суперкласса из конструктора подкласса оператор `super` используется с применением следующего синтаксиса:

`super(`*п1*`, `*п2*`, ..., `*пN*`)`

Параметры *п1*, *п2*, ..., *пN* — это фактические значения параметров, передаваемые конструктору суперкласса.

Примечание

Вызов конструктора суперкласса из тела конструктора подкласса с помощью оператора `super` должен выполняться в первой строке конструктора подкласса. Невыполнение этого правила приведет к возникновению ошибки компиляции (The superconstructor must be called first in the constructor body).

В следующем примере показана процедура инициализации свойств в конструкторе подкласса `PassangerLift`, который наследует от класса `Lift`.

```
class Lift {
        private var _currentFloor:Number;
        private var _currentPosition:Number;
        // Инициализация свойств суперкласса
        function Lift (currentFloor:Number) {
                _currentFloor = currentFloor;
                _currentPosition = currentFloor;
        }
}
```

```
class PassangerLift extends Lift {
        private static Lift:Array = new Array();
        function PassangerLift (currentFloor:Number) {
                // Вызываем конструктор суперкласса для инициализации
                // значения свойств currentFloor и _currentPosition,
                // объявленных в суперклассе
                if (currentFloor == undefined) {
                        super(1);
                }
                else {
                        super(currentFloor);
                }
        // Инициализируем статическое свойство подкласса, помещая в
        // хранящийся в нем массив ссылку на создаваемый объект
        // PassangerLift
        _pLifts.push(this);
        }
}
```

В этом примере свойства `_currentFloor` и `_currentPosition` объявлены в суперклассе. Для их инициализации в теле конструктора подкласса осуществляется вызов конструктора суперкласса. При этом в конструкторе суперкласса также осуществляется инициализация свойства (в данном примере статического) подкласса.

Представление о внутреннем механизме реализации наследования

Как мы уже упоминали ранее, при компиляции конструкторы классов, сохраненных во внешних AS-файлах, сохраняются в объекте `_global` в качестве его свойств. Именно поэтому создание объекта с помощью конструктора в FLA-файле может быть выполнено в коде любой монтажной линейки документа. При использовании подкласса `_global` в объект помещаются оба конструктора: подкласса и суперкласса.

Формально конструктор представляет собой обычную функцию. При создании любой функции в ActionScript она (будучи объектом класса `Function`) автоматически наделяется свойством `prototype`, о котором мы уже говорили ранее в связи с созданием наследуемых свойств класса. Как было показано, все наследуемые свойства и методы класса сохраняются в объекте `prototype`, являющемся свойством конструктора. Используя свойство `prototype`, можно организовывать наследование непосредственно в сценарии FLA-файла. Так, определив в объекте `prototype` конструктора встроенного класса `MovieClip` метод `onPress()`, можно заставить все экземпляры клипов реагировать на нажатие. При этом метод `onPress()` не будет копироваться в каждый клип индивидуально, а будет просто наследоваться ими (подобный подход с некоторым забеганием вперед был использован в листинге

15.4). Следующий код приводит к тому, что при нажатии на любой клип его имя будет выведено в панель **Output**:

```
MovieClip.prototype.onPress = function(){
trace(this._name);
}
```

Каждый объект, порождаемый встроенным или пользовательским классом, автоматически наделяется двумя свойствами: `__proto__` и `constructor`. Эти свойства по умолчанию являются скрытыми, поэтому их нельзя перечислить с помощью цикла `for...in` и вывести в панель **Output**, используя команду **Debug>List Variables**. Тем не менее с помощью глобальной встроенной функции `ASSetPropFlags()` можно "открыть" эти свойства, как это показано далее.

```
// Создаем экземпляр класса A
var myA:A = new A();
// Снимаем защиту со свойств экземпляра myA
ASSetPropFlags(myA, null, null, true);
trace(myA.__proto__); /* Выводит [object Object], то есть свойство
существует */
```

Назначение свойства `__proto__` заключается в том, что оно хранит ссылку на объект `prototype` конструктора, создавшего данный экземпляр. В этом можно легко убедиться с помощью следующей проверки:

```
// Создаем экземпляр класса A
var myA:A = new A();
trace(myA.__proto__ == A.prototype); // Выводит true
```

Назначение свойства `constructor` состоит в том, что оно хранит ссылку на конструктор, создавший данный экземпляр. Снова подтвердим это с помощью следующей проверки:

```
trace(myA.constructor == A); // Выводит true
```

Механизм организации наследования заключается в том, что свойству `prototype` конструктора подкласса присваивается объект, порожденный суперклассом. В ActionScript 1.0 эта процедура выполнялась явным образом и представляла собой основной способ реализации наследования (при этом объявление классов осуществлялось непосредственно в FLA-документах). ActionScript 2.0 предоставляет внешне более формализованный и логичный способ организации наследования, однако внутренняя реализация наследования по-прежнему выполняется в соответствии со стилем ActionScript 1.0.

В основе внутренней реализации наследования лежит следующая процедура:

```
Конструктор подкласса.prototype = new Конструктор суперкласса()
```

В результате присваивания экземпляра суперкласса прототипу конструктора подкласса подкласс наследует все элементы суперкласса (как наследуемые, так и собственные, то есть инициализированные в конструкторе суперкласса).

Когда экземпляр объекта осуществляет обращение к наследуемому свойству или методу, интерпретатор просматривает объект prototype конструктора, с помощью которого был создан данный объект, в поисках соответствующего свойства (метода). Для обращения к прототипу конструктора интерпретатор использует свойство __proto__ (*экземпляр*.__proto__). Если в прототипе конструктора искомый элемент не найден, интерпретатор обращается к прототипу объекта prototype, ссылку на который предоставляет его свойство __proto__ (*экземпляр*.__proto__.__proto__) и т. д. Продемонстрируем описанную модель на примере:

```
// Суперкласс
class A {
       private var _speed:Number = 300;
       public function elevate(){
       }
}
// Подкласс
class B extends A {
       private var _isBusy:Boolean = false;
       public function call() {
       }
}
var myB:B = new B();
// Снимаем защиту со свойств прототипа конструктора и его прототипа
ASSetPropFlags(B.prototype, null, null, true);
ASSetPropFlags(B.prototype.__proto__, null, null, true);
// Подтвердим, что класс B наследует от класса A
trace(B.prototype.__proto__ == A.prototype); // Выводит true
// Выводим свойства, наследуемые от суперкласса
for(i in B.prototype.__proto__){
       trace(i); // Выводит _speed elevate ...
}
// Выводим все свойства подкласса, включая свои и наследуемые
for(i in B.prototype){
       trace(i); // Выводит _speed elevate ... _isBusy call
}
```

Связывание символа с пользовательским классом

Пользовательские классы могут расширять встроенные классы ActionScript, используя стандартный формальный синтаксис наследования. Наряду с этим в Flash MX

2004 имеется чрезвычайно удобная возможность, позволяющая связать экземпляр класса `MovieClip` с пользовательским классом и, таким образом, дать клипу возможность унаследовать все свойства и методы пользовательского класса.

Связывание символа с классом с помощью окна *Linkage Properties*

Для того чтобы реализовать эту процедуру, необходимо выполнить следующие действия.

1. Создать пользовательский класс, расширяющий класс `MovieClip`, во внешнем AS-файле и сохранить его в каталоге, указанном в глобальных или локальных настройках **classpath**.
2. В FLA-документе создать в библиотеке символ типа `MovieClip`.
3. Выделить этот символ в библиотеке и из контекстного меню выполнить команду **Linkage...**.
4. В появившемся диалоговом окне **Linkage Properties** установить флажок **Export for ActionScript**, в поле **Identifier:** ввести идентификатор связи для символа, а в поле **AS 2.0 class:** указать имя класса, свойства и методы которого должен унаследовать данный символ. Следует иметь в виду, что если класс входит в состав пакета, в поле **AS 2.0 class:** необходимо указать полный путь к классу (включая имя пакета).

После выполнения описанных процедур любой экземпляр этого клипа будет наследовать поведение, реализованное в связанном с ним пользовательском классе (так, как если бы он был порожден этим классом).

Поскольку пользовательский класс расширяет встроенный класс `MovieClip`, из его определения можно обращаться к любым встроенным свойствам и методам класса `MovieClip`. Создаваемый экземпляр клипа, будучи связан с пользовательским классом, автоматически включается в область видимости членов этого класса. Поэтому в классе, который связывается с символом, можно выполнить определение методов обработчиков событий клипов (`onEnterFrame()`, `onPress()` и т. д.). Экземпляры, связанные с этим классом, будут наследовать эти обработчики и, таким образом, получат возможность единообразно реагировать на соответствующие события. Приведем простой пример, иллюстрирующий возможности связывания символа с пользовательским классом.

Создадим класс `RandomColor`, который будет позиционировать клип на сцене случайным образом, задавать ему случайный цвет и заставлять перемещаться по круговой траектории. Затем в FLA-документе создадим символ, содержащий изображение круга, и свяжем его с классом `RandomColor` с помощью диалогового окна **Linkage Properties**. После этого вытащим на сцену требуемое количество экземпляров символа. В принципе, экземпляры можно было бы создать динамически с помощью `attachMovie()`, однако мы для большей наглядности намеренно хотим избежать создания сценария в FLA-документе. Исходный код класса `RandomColor` и реализующий его документ randomColorClip.fla находятся на сопровождающем книгу компакт-диске в папке Глава 25.

Класс `RandomColor`. Случайное позиционирование и окрашивание клипов. Перемещение по круговой траектории со случайной скоростью

```
class RandomColor extends MovieClip {
        // X-координата
        public var _x:Number;
        // Y-координата
        public var _y:Number;
        // Начальные значения координат
        var initX:Number;
        var initY:Number;
        // Цветовой объект
        private var col:Color;
        // Начальный угол
        private var a:Number;
        private var increment:Number;
        function RandomColor() {
                // Позиционируем клип случайным образом
                _x = Math.random() * Stage.width;
                _y = Math.random() * Stage.height;
                // Сохраняем начальные значения координат
                initX = _x;
                initY = _y;
                // Инициализируем объект Color (this - это ссылка на
                // объект, с которым связывается класс)
                col = new Color(this);
                // Применяем окрашивание
                col.setRGB(Math.random() * 0xFFFFFF);
                // Задаем случайный угол
                a = Math.random() * 2 * Math.PI;
                // Приращение угла
                increment = Math.random() * 0.5 + 0.1;
        }
        // Определяем обработчик onEnterFrame()
        public function onEnterFrame() {
                // Реализация перемещения по окружности
                _x = initX + 25 * Math.sin(a);
                _y = initY + 25 * Math.cos(a);
                // Приращение аргумента
```

```
                a += increment;
                if (a > 2 * Math.PI) {
                        a = 0;
                }
        }
}
```

Несмотря на то, что в FLA-файле не создано ни одной строчки кода, наши клипы ведут себя так, как и ожидалось, это почти волшебство!

Связывание символа с классом с помощью метода *Object.registerClass()*

Применение окна **Linkage Properties** для связи символа с классом — это удобный и простой способ, однако в некоторых случаях он не позволяет добиться желаемого результата. Он, например, не позволяет связывать определенный символ с классом, имя которого определяется динамически уже непосредственно в процессе воспроизведения фильма. Этот способ также не позволяет связать класс с символом, идентификатор связи которого формируется динамически. Такая ситуация будет иметь место в примере, рассматриваемом в *главе 27*, при обеспечении возможности создания пользовательских графических оболочек компонента.

В ActionScript имеется и более гибкий программный инструмент, позволяющий связать символ библиотеки с экземпляром класса. Для этих целей применяется статический метод класса `Object`, который называется `Object.registerClass()`. Данный метод имеет следующий синтаксис:

`Object.registerClass(`*идентификатор связи*`,` *класс*`)`

Параметр *идентификатор связи* — это строка, содержащая идентификатор связи символа, который требуется связать с классом. Этот параметр может задаваться не только литералом, но и выражением, возвращающим строку.

Параметр *класс* — это ссылка на конструктор класса, с которым связывается символ, указанный первым параметром. Данный параметр представляет собой идентификатор, то есть имя конструктора (без оператора вызова функции `()`) или имя переменной, содержащей ссылку на конструктор класса.

При создании экземпляра символа, связанного с внешним классом, Flash автоматически вызывает конструктор данного класса без параметров. Идентификатор `this`, используемый в конструкторе класса, возвращает ссылку на создаваемый экземпляр.

Необходимо иметь в виду, что связывание символа с классом с помощью метода `Object.registerClass()` оказывает влияние только на экземпляры, которые будут созданы после выполнения связывания. Экземпляры, созданные до вызова метода `Object.registerClass()`, не подвергаются никаким изменениям.

При попытке связать класс с символом, уже связанным с другим классом, предыдущая связь разрывается и вместо нее устанавливается новая связь. При этом все экземпляры, созданные к моменту установления новой связи, остаются связанными

со старым классом, а экземпляры, которые будут созданы в дальнейшем, будут управляться новым классом.

Для того чтобы разорвать связь символа с классом, необходимо вызвать метод `Object.registerClass()`, передав ему в качестве второго параметра значение `null`. При этом все ранее созданные экземпляры останутся связанными с предыдущим классом, а новые экземпляры будут по умолчанию связываться с классом `MovieClip`.

Для того чтобы реализовать приведенный выше пример с цветными шариками, не прибегая к настройкам окна **Linkage Properties**, необходимо поместить в кадр следующий код:

```
// Связываем символ с классом
Object.registerClass("mc", RandomColor);
// Создаем экземпляры клипов
for(i=0; i<20; i++){
       this.attachMovie("mc", "mc"+i, i);
}
```

Обратите внимание на то, что здесь экземпляры символа создаются динамически уже после осуществления связывания символа с классом `RandomColor`, поэтому каждый из них обладает поведением, реализованным в классе `RandomColor`. Если бы мы создали экземпляры символа в рабочей среде, просто поместив их на сцену, и в том же кадре поместили код, выполняющий связывание родительского символа с управляющим классом, это никак не отразилось бы на поведении экземпляров. Тем не менее экземпляры, которые мы бы создали далее программным образом с помощью метода `attachMovie()`, обладали бы поведением, реализованным в классе `RandomColor`.

Для того чтобы связать с пользовательским классом экземпляры символов, созданных в рабочей среде вручную, можно было бы поместить вызов метода `Object.registerClass()` в кадр, предшествующий кадру, в котором содержатся эти экземпляры. Такой способ позволит добиться требуемого результата, однако зачастую он оказывается неудобным, поскольку усложняет структурную организацию документа. Гораздо более эффективным подходом при решении подобных задач является применение директив `#initclip` и `#endinitclip`.

Директивы *#initclip* и *#endinitclip*

Директивы `#initclip` и `#endinitclip` позволяют сообщить компилятору, что снабженный ими блок кода необходимо выполнить перед выполнением кода кадра основной монтажной линейки. Эти директивы могут быть использованы только в сценарии, помещенном в первый кадр монтажной линейки клипа. Их нельзя поместить в кадр основной монтажной линейки (это вызовет ошибку компиляции). При использовании данных директив применяется следующий синтаксис:

```
#initclip номер
блок кода;
```

```
...
#endinitclip
```

Здесь *номер* — это целое положительное число, позволяющее задать порядок, в соответствии с которым будет происходить выполнение блоков кода, снабженных директивами `#initclip` и `#endinitclip`. Блок кода, снабженный директивой `#initclip` с меньшим значением порядкового номера, будет выполнен раньше, чем блок кода, снабженный директивой `#initclip` с большим значением. Так, если в первом кадре клипа `A` находится блок кода, снабженный директивой `#initclip` с порядковым номером 1, а в первом кадре клипа `B` находится блок кода, снабженный директивой `#initclip` с порядковым номером 2, то вначале будет выполнен код клипа `A`, а потом код клипа `B`.

Блок кода клипа, снабженный директивами `#initclip` и `#endinitclip`, будет выполнен непосредственно перед выполнением сценария кадра, в котором впервые появляется экземпляр этого клипа.

Если символ, на монтажной линейке которого имеются директивы `#initclip` и `#endinitclip`, экспортируется для ActionScript с помощью соответствующей настройки диалогового окна **Linkage Properties**, то его блок кода будет выполнен перед выполнением кода первого кадра фильма. Причем это будет иметь место вне зависимости от того, присутствуют ли на монтажной линейке фильма экземпляры данного символа или нет. Таким образом, директивы `#initclip` и `#endinitclip` позволяют выполнить сценарий клипа, даже если он просто находится в библиотеке. Возможность выполнения кода экспортируемого символа перед первым кадром фильма дает возможность инициализировать клип еще до того, как на монтажной линейке будут созданы его экземпляры. Так, например, для того чтобы связать экземпляры символа, имеющие идентификатор связи `mc` и помещенные в первый кадр монтажной линейки в рабочей среде, с классом `RandomColor`, нужно в первый кадр этого символа поместить следующий код:

```
#initclip
Object.registerClass("mc", RandomColor);
#endinitclip
```

Поскольку символ экспортируется для ActionScript, приведенный сценарий будет выполнен перед первым кадром фильма, поэтому, когда воспроизводящая головка войдет в первый кадр, содержащий экземпляры символа, они будут обладать поведением, реализованным в классе `RandomColor`. Именно такой механизм и лежит в основе обеспечения возможности связывания класса с символом с помощью диалогового окна **Linkage Properties**. Если поле **AS 2.0 Class:** этого окна содержит идентификатор класса, то при публикации Flash автоматически помещает в первый кадр символа, с которым связывается класс, код, аналогичный приведенному.

Если требуется выполнить инициализацию нескольких клипов, то в каждый из них можно поместить код, снабженный директивами `#initclip` и `#endinitclip`. Последовательность выполнения таких блоков кода будет определяться порядковыми номерами директивы `#initclip`.

Примечание

Необходимо помнить, что инициализация большого объема программного кода с помощью директив `#initclip` и `#endinitclip` перед первым кадром фильма может привести к значительной задержке в его загрузке. Причем на протяжении этой задержки предзагрузчик, помещенный в первый кадр фильма, работать, естественно, не будет.

Следует иметь в виду, что блок кода, снабженный директивами `#initclip` и `#endinitclip`, выполняется только один раз на протяжении воспроизведения документа. При возврате воспроизводящей головки на кадр клипа, содержащий данные директивы, блок кода повторно выполняться не будет. В связи с этим директивы `initclip` и `#endinitclip` применяются в основном для инициализации классов (в стиле ActionScript 1.0) и для связывания символов с классами.

Интерфейсы

Интерфейсы представляют собой своеобразное описание функциональности, реализуемой классом. Формально интерфейс — это перечень методов класса с указанием количества и типа принимаемых ими параметров, а также типа возвращаемого результата.

Создание интерфейса не является обязательной процедурой, его наличие или отсутствие никак не сказывается на работе класса. Тем не менее создание интерфейса имеет несколько очень существенных преимуществ (причем их значимость возрастает пропорционально увеличению объема создаваемого проекта и количеству разработчиков, принимающих участие в его создании).

- Интерфейс предоставляет удобную возможность документирования функциональных возможностей класса.
- Наличие интерфейса позволяет на этапе компиляции проверить, снабжен ли класс требуемой функциональностью.
- Наличие интерфейса позволяет стандартизировать функциональные возможности классов и обеспечить возможности их стыковки. Это особенно актуально, когда над различными классами работает не один человек, а целый коллектив.

Интерфейсы, как и классы, создаются в отдельных AS-файлах. Причем они, как и классы, могут объединяться в пакеты и импортироваться с помощью ключевого слова `import`. Определение интерфейса должно включать в себя следующие элементы.

- Объявление интерфейса.
- Перечень названий открытых нестатических методов с указанием количества и типа принимаемых ими параметров и типа возвращаемого значения.

В определении интерфейса запрещается размещать:

- реализацию методов. Интерфейс должен включать только объявление методов (без фигурных скобок), их реализация выполняется в классе;
- статические методы;
- защищенные методы, то есть методы с атрибутом `private`;

- атрибуты `get` и `set`;
- объявление свойств. Интерфейсы включают только объявления методов.

Для объявления интерфейса используется ключевое слово `interface`. Синтаксис объявления интерфейса напоминает объявление класса:

```
interface имя пакета.имя интерфейса {
...
}
```

Здесь *имя пакета* — это указание пакета, в котором сохранен файл интерфейса. Идентификатор *имя интерфейса* должен соответствовать правилам, предъявляемым к идентификаторам. Как и в случае с классом, имя интерфейса должно совпадать с именем AS-файла, в котором он сохраняется.

Приведем пример интерфейса для класса `Lift`, описание которого будет представлено в следующем разделе.

```
interface I_Lift {
       public function elevate(floor:Number):Void;
       public function onElevate():Void;
       public function onStop():Void;
}
```

Интерфейсы, как и классы, могут реализовывать наследование. Для этого в их объявлении нужно использовать ключевое слово `extend`.

```
interface I_Lift extend Liftable {
...
}
```

После определения интерфейса он может быть реализован конкретным классом. Для этого в объявлении класса необходимо использовать ключевое слово `implements`.

```
class Lift implements I_Lift{
...
}
```

В процессе компиляции Flash проверяет, реализованы ли в классе все методы, указанные в интерфейсе, и совпадают ли количество, тип параметров и тип возвращаемого результата у методов, реализованных в классе, и у методов, указанных в интерфейсе. При наличии расхождений Flash выведет сообщение об ошибке (The implementation of the interface method doesn't match its definition). Это позволяет обеспечить надежный контроль над тем, чтобы все методы были реализованы в классе в соответствии с разработанными при планировании правилами.

Один и тот же класс может одновременно реализовывать несколько интерфейсов. В этом случае после ключевого слова `implements` их следует перечислить через запятую:

```
class Lift implements I_Elevator, I_Hoist, I_Lift {
...
}
```

Подкласс также может реализовывать интерфейс, при этом ключевое слово `interface` указывается после ключевого слова `extends`:

```
class PassangerLift extends Lift implements I_Elevator, I_Lift {
...
}
```

Характерной особенностью реализации интерфейса является то, что интерфейс, подобно классу, способен определять тип данных. Так, если класс `Lift` реализует интерфейс `I_Lift`, то при создании экземпляра этого класса ссылку на него можно сохранить в переменной типа `I_Lift`.

```
var lift:I_Lift = new Lift();
```

Иногда может потребоваться в процессе воспроизведения фильма определить, реализует ли класс, породивший объект, или его суперкласс, определенный интерфейс. В зависимости от этого, например, можно выполнить вызов метода данного класса (если интерфейс реализован) или осуществить иные действия. Для выполнения такой проверки необходимо вызвать функцию, имя которой совпадает с именем интерфейса, и передать ей в качестве параметра выражение, возвращающее объект. Если данный объект является экземпляром класса, реализующего интерфейс, функция возвращает этот объект. В противном случае функция возвращает `null`. В следующем примере выполняется проверка, реализует ли класс (или его суперкласс), породивший объект `myObj`, интерфейс `I_Lift`. Если да, то объект `myObj` вызывает метод `elevate()`, указанный в определении интерфейса и реализованный в породившем его классе.

```
if(I_Lift(myObj) != null){
myObj.elevate();
}
```

Пример создания и применения пользовательских классов

В завершение темы создания пользовательских классов рассмотрим учебный пример, иллюстрирующий ее практическое применение. В качестве такого примера мы создадим модель системы пассажирских лифтов в многоэтажном здании.

Вначале приведем краткое описание функциональности, которая должна быть реализована в нашей модели лифтов.

- Количество лифтов в системе должно определяться уже непосредственно в процессе ее реализации. Таким образом, для небольшого дома может быть использовано меньшее количество лифтов, в то время как в небоскребе их число должно быть значительно больше. Это означает, что поведение каждого конкретного лифта должно быть полностью инкапсулировано в классе и не зависеть от фактического числа лифтов.
- Лифт должен обладать способностью перемещаться между этажами с заданной скоростью, причем у пользователя должна быть возможность задания временного

интервала, за который лифт перемещается между соседними этажами. Перемещение лифта должно выполняться плавно, то есть шаг дискретности его смещения должен зависеть от заданной пользователем скорости. Пользователь должен иметь возможность отслеживания процесса перемещения лифта извне (например, для того чтобы визуализировать его с помощью внешнего кода).

- Необходимо обеспечить возможность вызова лифта на любой этаж здания. На вызов должен реагировать только ближайший к этому этажу свободный лифт. Причем приоритет отдается лифтам, которые находятся выше (то есть если несколько лифтов находятся на одинаковом расстоянии от этажа, вызываться будет тот, который должен двигаться вниз). Вызов лифта на этаж может быть выполнен только в том случае, если на этом этаже уже нет лифта.
- Необходимо обеспечить возможность отправки свободного лифта на любой этаж (как если бы пассажир, войдя в лифт, поехал на нужный ему этаж).

Для обеспечения описанной функциональности мы создадим два класса. Класс `Lift` будет включать базовую функциональность, позволяющую выполнять перемещение лифта на заданный этаж и отслеживать это перемещение. Класс `PassangerLift` будет являться подклассом класса `Lift`, расширяя его функциональность. В классе `PassangerLift`, в частности, будет реализован метод `call()`, позволяющий осуществить вызов ближайшего к этажу незанятого лифта. Визуализацию лифтов и создание графического интерфейса, позволяющего управлять ими, мы выполним с помощью кода и графических элементов непосредственно в FLA-файле. Таким образом, классы `Lift` и `PassangerLift` обеспечат функциональность абстрактного Лифта, а ее физическое воплощение будет реализовано с помощью обычного сценария кадра. Далее будут приведены листинги, содержащие определения классов и программный код результирующего FLA-файла. Некоторые разъяснения внутри листинга будут даны обычным шрифтом. Исходные программные коды классов, снабженные подробными комментариями, находятся в файлах Lift.as и PassengerLift.as, а исходный файл конечного проекта сохранен с именем liftModel.fla. Все указанные файлы читатель найдет на сопровождающем книгу компакт-диске в папке Глава 25.

Класс `Lift`. Обеспечение базовой функциональности Лифта

Класс `Lift` реализует интерфейс `I_Lift`, определение которого было приведено в предыдущем разделе.

```
class Lift implements I_Lift{


// ******************** ОБЪЯВЛЕНИЕ СВОЙСТВ ****************************

        /* Свойство, хранящее номер этажа, к которому нужно переместить
        лифт. Данное свойство принимает конкретное числовое значение
        только при перемещении лифта. Если лифт стоит, оно принимает
        значение null */

        private var _floor:Number;

        /* Свойство, хранящее номер текущего этажа, на котором находится
        неподвижный лифт, или этаж, который покинул, еще не достигнув
        следующего, движущийся лифт */
```

```
private var _currentFloor:Number;
/* Текущее положение лифта на интервале от его исходного этажа до
этажа с номером _floor. Значение свойства _currentPosition будет
совпадать со значением свойства _currentFloor, когда лифт будет
находиться на этаже. В промежутках между этажами _currentPosition
будет иметь дробное значение. */
private var _currentPosition:Number;
```

Для реализации смещения лифта далее будет создан метод `moveLift()`. Перемещение лифта выполняется за счет периодического вызова метода `moveLift()` с помощью функции `setInterval()`. Поскольку смещение лифта должно выполняться плавно (а не с шагом, равным целому этажу), необходимо рассчитать шаг смещения, то есть расстояние, на которое будет смещен лифт при каждом вызове метода `moveLift()`. Эта величина должна определяться исходя из двух параметров:

- времени, за которое лифт должен переместиться между соседними этажами. Эта величина будет храниться в защищенном свойстве `_speed`;
- периодичности, с которой будет осуществляться вызов метода `moveLift()`, смещающего лифт. Значение частоты вызова этого метода будет храниться в защищенном свойстве `_updatesInterval`. Частота вызова метода `moveLift()` определяет, сколько раз лифт будет изменять свое положение при перемещении между двумя соседними этажами, то есть насколько плавным будет его перемещение.

Исходя из сказанного, величину единичного смещения лифта можно определить как частное от деления `_updatesInterval` на `_speed`. Для хранения значения шага смещения используется защищенное свойство `_increment`. Значение этого свойства не должно представлять собой бесконечную дробь, поэтому при его вычислении в дальнейшем (в определении подкласса) мы будем округлять результат до третьего знака (здесь этого делать не нужно, так как 50/500 = 0,1).

```
/* ID-номер интервала, установленного функцией setIntrval(),
с помощью которого выполняется перемещение лифта
*/
private var _moveInterval:Number;
// Количество миллисекунд, за которое лифт проходит один этаж
private static var _speed:Number = 500;
// Частота вызова метода moveLift(), смещающего лифт, в мсек
private static var _updatesInterval:Number = 50;
/* Величина единичного смещения лифта при движении с заданными
скоростью и частотой смещения
*/
private var _increment:Number =_updatesInterval /_speed;
```

```
/* Свойство, показывающее, занят ли лифт. Данное свойство
используется в качестве флага, меняя значение в зависимости
от состояния лифта. Лифт считаем занятым, если он движется (true),
и свободным (false)
*/
private var _isBusy:Boolean = false;

// ********** ОБЪЯВЛЕНИЕ КОНСТРУКТОРА *************

/* Конструктор, ему в качестве параметра передается этаж,
на котором лифт находится в момент создания объекта Lift.
*/
function Lift(currentFloor:Number) {
       _currentFloor = currentFloor;
       _currentPosition = currentFloor;
       //trace("Класс Lift загружен");
}
// ********** ОБЪЯВЛЕНИЕ getter/setter МЕТОДОВ *************
// Определение текущего этажа лифта
public function get currentFloor ():Number{
return _currentFloor;
}
// Определение текущего положения лифта
public function get currentPosition ():Number{
return _currentPosition;
}
// Определение того, занят ли лифт
public function get isBusy ():Boolean{
return _isBusy;
}
```

Все объявленные методы предоставляют только возможность извлечения значений свойств. Соответствующие `set`-методы не объявлены, поскольку свойства `currentFloor`, `currentPosition` и `isBusy` предназначены только для чтения и не должны быть заданы извне.

```
// ********** ОБЪЯВЛЕНИЕ МЕТОДОВ *************
```

В организации перемещения задействованы два метода. Открытый метод `elevate()` перемещает лифт на заданный этаж только в том случае, если он не занят или уже не находится на этом этаже. Метод `elevate()` доступен извне и представляет собой инструмент программного интерфейса, с помощью которого можно управлять лифтами. Собственно перемещение лифта реализуется с помощью защищенного метода

moveLift(). Для того чтобы в конечном проекте можно было отслеживать процесс перемещения лифта, в классе Lift будут объявлены методы onElevate() и onStop(). Эти методы исполняют роль обработчиков событий. Метод onElevate() будет вызываться из тела метода moveLift() при каждом смещении лифта. Метод onStop() будет вызываться в момент, когда лифт достигнет заданного свойством _floor этажа или при попытке перемещения лифта к этажу, на котором он уже находится. В определении класса методы onElevate() и moveLift() объявлены как пустые, то есть не содержат никаких инструкций. Для контроля событий onElevate и onStop необходимо определить методы с соответствующими именами для конкретных объектов, порожденных классом. Формально эта процедура похожа на использование методов обработчиков событий встроенных классов.

```
/* Метод, выполняющий перемещение лифта к заданному параметром
floor этажу
*/
public function elevate(floor:Number):Void {
        // Если лифт не занят...
        if(!isBusy){
                // Если лифт уже не находится на этаже floor...
                if(floor != _currentFloor){
                _floor = floor;
                // Инициируем перемещение лифта. Поскольку
                // вызывается не функция, а метод объекта, применяем
                // вторую форму синтаксиса функции setInterval()
                // (первый параметр - это ссылка на объект,
                // содержащий объявление вызываемого метода)
                _moveInterval = setInterval(this, "moveLift",↳
                _updatesInterval);
                // Переключаем флаг, поскольку лифт теперь занят
                _isBusy = true;
                }
                // Вызываем обработчик onStop()
                else onStop();
        }
}
// Внутренний метод, используемый для реализации перемещения лифта
private function moveLift():Void {
        // Вычисляем, какое расстояние должен пройти лифт
        var _distance:Number = _floor - _currentPosition;
        // Если лифт находится ниже этажа _floor, поднимаем его
        if (_distance > 0) {
                // Округляем значение _currentPosition до трех
                // десятичных знаков
```

```
            _currentPosition = ↳
            Math.round(_currentPosition*1000)/1000
            _currentPosition += _increment;
            // Проверяем, достаточно ли близко лифт подъехал к
            // очередному этажу, чтобы можно было считать, что
            // он его достиг
            if((Math.ceil(_currentPosition) - ↳
            _currentPosition)<_increment ){
            // Присваиваем свойству _currentFloor номер текущего
            // этажа
            _currentFloor = Math.ceil(_currentPosition);
            }
        }
        // Если лифт находится выше этажа _floor, опускаем его
        // с применением аналогичного алгоритма
        else if (_distance < 0) {
        _currentPosition = Math.round(_currentPosition*1000)/1000
        _currentPosition -= _increment;
            if((_currentPosition - ↳
            Math.floor(_currentPosition))<_increment ){
            _currentFloor = Math.floor(_currentPosition);
            }
        }
        // Вызываем метод onElevate, который будет определен для
        // объекта Lift в FLA-файле. Это дает возможность
        // контролировать процесс перемещения лифта, используя
        // onElevate как событие, например, для визуализации
        // движения лифта
        onElevate();
        // Если лифт достиг этажа _floor...
        if (Math.abs(_currentPosition - _floor) < _increment) {
            // Если текущее положение лифта дробное и не
            // соответствует строго номеру этажа _floor (то есть
            // лифт немного не доехал до этажа из-за неровных
            // дробных значений _increment), искусственно
            // "доводим" его до этажа _floor
            if(_currentPosition != _floor){
            _currentPosition = Math.round(_currentPosition);
            onElevate();
            }
            // Прекращаем перемещение лифта
            clearInterval(_moveInterval);
            _isBusy = false;
```

```
                    // Сбрасываем значение свойства _floor, так как лифт
                    // прибыл на заданный этаж
                    _floor = null;
                    // Вызываем метод onStop()
                    onStop();
                }
        }
        // Методы, которые могут быть определены в FLA-файле как
        // обработчики событий onElevate и onStop. В результате
        // определения собственных методов с соответствующими именами у
        // объектов, порожденных классом Lift (и его подклассами),
        // появляется возможность выполнять определенные действия на
        // протяжении перемещения лифта (onElevate) и в момент его
        // остановки (onStop)
        public function onElevate():Void {
        }
        public function onStop():Void {
        }
}
```

Класс `PassangerLift` является подклассом класса `Lift` и, наряду с его функциональностью, позволяет дополнительно реализовать следующие возможности.

- Задание скорости лифта путем передачи соответствующего значения конструктору, или используя специальное `get/set`-свойство программного интерфейса.
- Возможность вызова на заданный этаж ближайшего к нему лифта с помощью метода `call()`. Метод `call()` объявлен как статический, поскольку он вызывается сразу для всех экземпляров, определяя, какой из них находится ближе к заданному этажу.

Класс `PassangerLift`. Реализация регулирования скорости лифта и возможности его вызова на заданный этаж

```
class PassangerLift extends Lift {

// ****************** ОБЪЯВЛНИЕ СВОЙСТВ *********************
```

Поскольку при вызове лифта с помощью метода `call()` необходимо проверить положение всех лифтов, то есть всех объектов `PassangerLift`, необходимо сохранить ссылки на все эти объекты в специальном статическом свойстве `_pLifts`. Кроме того, каждый объект `PassangerLift` необходимо снабдить собственным свойством, позволяющим идентифицировать этот объект во внешнем сценарии. Для хранения идентификационного номера объекта `PassangerLift` мы используем защищенное свойство `_id`.

```
        /* Массив для хранения ссылок на все созданные объекты
        PassangerLift */
```

```
private static var _pLifts:Array = new Array();
// Идентификационный номер создаваемого объекта PassangerLift
private var _id:Number;
// Количество миллисекунд, за которое лифт проходит один этаж.
// Если в объекте не создается собственного свойства _speed, он
// использует значение, наследуемое из суперкласса
private var _speed:Number;
// Свойства задают нижний и верхний пределы для скорости лифта
private static var MIN_SPEED = 100;
private static var MAX_SPEED = 5000;

// ********** ОБЪЯВЛНИЕ КОНСТРУКТОРА *************
// Конструктору в качестве параметра передается этаж, на котором
// лифт находится в момент создания объекта PassangerLift и
// значение скорости лифта. Второй параметр является
// необязательным. В случае его отсутствия в качестве скорости
// объекта будет использовано значение свойства _speed
// суперкласса.
function PassangerLift(currentFloor:Number, speedValue:Number) {
       // Вызываем конструктор суперкласса для инициализации
       // значений всех наследуемых свойств
       if (currentFloor == undefined) {
              super(1);
       }
       else {
              super(currentFloor);
       }
       // Добавляем ссылку на текущий объект в массив _pLifts. Это
       // дает возможность контролировать все созданные объекты
       // PassengerLift из кода класса
       _pLifts.push(this);
       // Присваиваем значение идентификатору
       _id = _pLifts.length;
       // Устанавливаем скорость лифта
       if(speedValue != undefined){
       _speed = Math.max(MIN_SPEED, Math.min(speedValue,↳
       MAX_SPEED));
       // Вычисляем значение свойства _increment, объявленного в
       // суперклассе
       _increment = Math.round(_updatesInterval/_speed*1000)/1000;
       }
```

```
                //trace("Класс PassengerLift загружен");
        }
// ********** ОБЪЯВЛЕНИЕ getter/setter МЕТОДОВ *************
        // Определение идентификатора объекта
        public function get id():Object {
                return _id;
        }
        // Определение скорости объекта PassengerLift
        public function get speed ():Number{
                return _speed;
        }
        // Установка скорости объекта PassengerLift. Если для объекта не
        // задается собственное значение скорости (с помощью конструктора
        // или с применением set-свойства объекта), он наследует свойства
        // _speed и _increment из суперкласса, то есть эти свойства не
        // создаются в самом объекте
        public function set speed (value:Number){
                // Вводим контроль заданного пользователем значения
                // скорости. Если оно будет выходить за допустимые границы,
                // присваиваем ему граничное значение
                _speed = Math.max(MIN_SPEED, Math.min(value, MAX_SPEED));
                // Вычисляем значение наследуемого свойства _increment,
                // исходя из заданного значения скорости, и округляем
                // полученный результат до третьего знака
                _increment =
Math.round(_updatesInterval/_speed*1000)/1000;
        }
        // ********** ОБЪЯВЛНИЕ МЕТОДОВ *************
        // Метод call в качестве параметра принимает номер этажа, с
        // которого выполняется вызов лифта, и возвращает true в случае
        // успешного вызова и false, если вызов лифта не может быть
        // выполнен (все лифты заняты или лифт уже вызван на этот этаж)
        public static function call(fromFloor:Number):Boolean {
                // Если лифт уже вызван, прекращаем действие метода,
                // возвращаем false
                for (var i = 0; i < _pLifts.length; i++) {
                        if (_pLifts[i]._floor == fromFloor) {
                        trace("Лифт уже вызван на " + fromFloor + " этаж");
                        return false;
                        }
                }
```

Перед тем как начать перемещение лифта на заданный этаж, необходимо определить, какой из свободных лифтов (то есть объектов PassangerLift) находится к заданному этажу ближе всего. При этом следует иметь в виду, что приоритет отдается лифту, работающему на спуск. Так, например, если вызов выполняется с пятого этажа, а свободные лифты находятся на четвертом и на шестом, будет вызван лифт с шестого этажа. Поскольку свободный лифт всегда находится строго на этаже (это обеспечивается методом moveLift() суперкласса), расстояние от этажа fromFloor, с которого выполняется вызов, до *i*-го лифта можно оценить с помощью разности fromFloor - _pLifts[i]._currentFloor. Поскольку нам нужно не просто определить, какой свободный лифт находится ближе, но и отдать предпочтение вышерасположенному лифту, для выполнения сортировки используем следующий алгоритм.

1. Создадим два массива positiveDistArray и negativeDistArray, сохранив ссылки на них в локальных переменных.
2. Для каждого лифта, представляемого объектом PassangerLift (то есть элемента массива _pLifts), сформируем объект следующего вида: {d:dist, num:i}, где d — это количество этажей, отделяющих текущий этаж, на котором находится *i*-й лифт от этажа с номером fromFloor, num — порядковый номер лифта, соответствующий расположению элемента со ссылкой на этот лифт в массиве _pLifts.
3. В массив positiveDistArray помещаем сформированные объекты, соответствующие лифтам, расположенным под этажом fromFloor, а в массив negativeDistArray помещаем сформированные объекты, соответствующие лифтам, расположенным над этажом fromFloor.
4. Сортируем элементы в каждом из этих массивов по возрастающей по значению поля d (то есть по расстоянию до этажа fromFloor).
5. Сравниваем первые элементы отсортированных массивов positiveDistArray и negativeDistArray и на основании значения свойства d (а при равенстве этих значений учитывая, какой из лифтов находится над этажом) делаем вывод о том, какой лифт находится ближе. Номер соответствующего ему объекта PassangerLift в массиве _pLifts хранится в свойстве сформированного объекта num.

```
// --------------Сортировка---------------------------------

    // Массив для хранения объектов с информацией о лифтах,
    // находящихся под этажом fromFloor

    var positiveDistArray = new Array();

    // Массив для хранения объектов с информацией о лифтах,
    // находящихся над этажом fromFloor

    var negativeDistArray = new Array();

    // Проходим в цикле все лифты...

    for (var i = 0; i < _pLifts.length; i++) {

           // Если i-ый лифт уже на этаже и не занят

           if (_pLifts[i].currentFloor == fromFloor &&↵
           !_pLifts[i]._isBusy) {

                  trace("Лифт уже на " + fromFloor + " этаже");
```

```
                return false;
        }
        // Если i-й лифт занят, переходим к следующему лифту i+1
        if (_pLifts[i]._isBusy) {
                continue;
        }
        // Количество этажей, отделяющих текущий этаж, на котором
        // находится i-й лифт от этажа с номером fromFloor
        var dist:Number = fromFloor - _pLifts[i]._currentFloor;
        if (dist >= 0) {
                positiveDistArray.push({d:dist, num:i});
        }
        else {
                negativeDistArray.push({d:dist, num:i});
                }
        }// *Окончание тела цикла*
        // Если оба массива оказались пусты...
if (positiveDistArray.length == 0 && negativeDistArray.length==0){
        trace("В настоящий момент все лифты заняты");
        return false;
}
// Если в массиве более одного элемента, сортируем его по
// возрастающей
if (positiveDistArray.length > 1) {
        positiveDistArray.sortOn("d", Array.NUMERIC);
}
if (negativeDistArray.length > 1) {
// Поскольку в этом массиве значения отрицательные, сортируем его
// по убывающей (так как критерием сортировки является абсолютная
// величина расстояния без учета знака)
negativeDistArray.sortOn("d", Array.NUMERIC| Array.DESCENDING);
}
// Наименьшее расстояние до лифта с нижнего этажа
var posDist:Number = Math.abs(positiveDistArray[0].d);
// Наименьшее расстояние до лифта с верхнего этажа
var negDist:Number = Math.abs(negativeDistArray[0].d);
// Если все лифты находятся выше этажа fromFloor
if (positiveDistArray.length==0 && negativeDistArray.length != 0){
        // Номер ближайшего лифта
```

```
                var closestLift:Number = negativeDistArray[0].num;
        }
        // Если все лифты находятся ниже этажа fromFloor
        else if (positiveDistArray.length !== 0 &&↳
        negativeDistArray.length == 0) {
                var closestLift:Number = positiveDistArray[0].num;
        }
        else {
                if (posDist - negDist >= 0) {
                        var closestLift:Number = negativeDistArray[0].num;
                }
                else {
                        var closestLift:Number = positiveDistArray[0].num;
                }
        }
//------------------Окончание сортировки----------------------------
                // Отправляем ближайший свободный лифт на заданный этаж
                _pLifts[closestLift].elevate(fromFloor);
                return true;
        } //*Конец определения метода call()*
        // Метод для отправки свободного лифта на заданный этаж. Выполняет
        // те же функции, что и elevate(), но имеет более интуитивно
        // понятное (для пользователя) название
        public function send(toFloor:Number) {
                elevate(toFloor);
        }
}
```

На рис. 25.2 представлена практическая реализация системы, состоящей из 6 лифтов и 11-этажного здания, выполненная в FLA-файле.

Созданная система снабжена следующей интерактивной функциональностью. С помощью кнопок, соответствующих этажам, расположенных в левой части экрана, выполняется вызов лифта на этаж. Если лифт вызван на этаж с помощью одной из этих кнопок, на ней загорается индикация. До тех пор, пока лифт не прибудет на этаж, кнопка этого этажа остается неактивной.

При нажатии на покоящийся лифт в правой части экрана появляется соответствующая ему панель управления с кнопками, позволяющими отправить лифт на любой этаж. При нажатии кнопки в панели управления лифта панель убирается с экрана. Вызвать панель управления для движущегося лифта нельзя.

При перемещении лифта в нем указывается номер текущего этажа, который он миновал или на котором остановился.

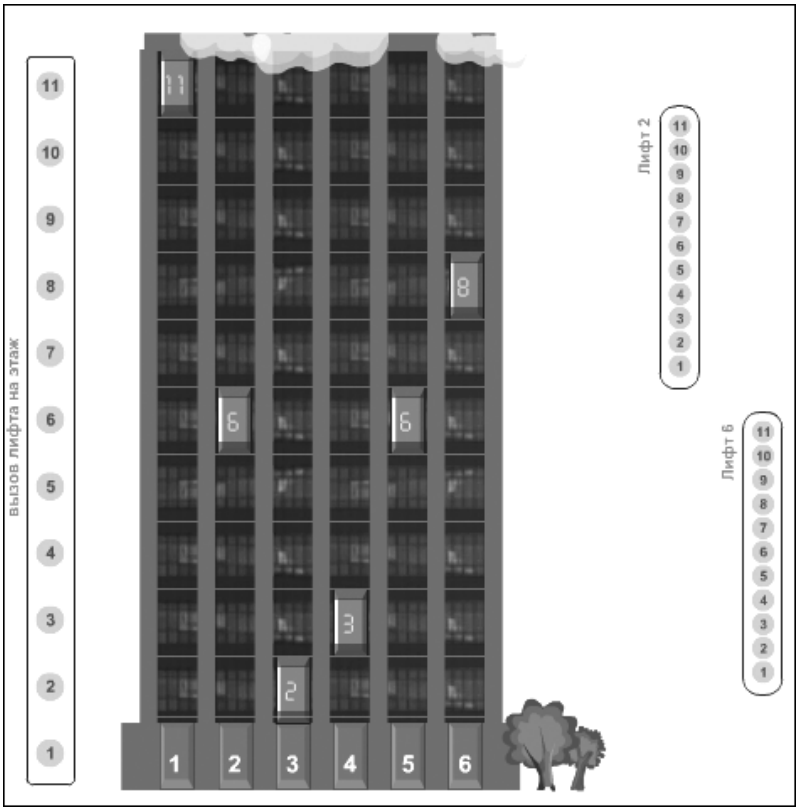


Рис. 25.2. Модель системы лифтов в действии

Программный код, реализующий описанную функциональность, снабжен комментариями и приведен далее в листинге. Приведем перечень используемых идентификаторов и особенностей их реализации.

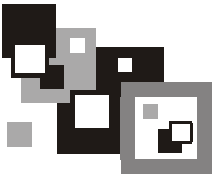
- Лифты, визуализирующие объекты `PassangerLift`, представлены клипами с именами `lift1`, `lift2`, ..., `lift6`. На монтажной линейке такого клипа находится текстовое поле с именем `txt`, предназначенное для вывода текущего этажа, на котором находится лифт.
- Кнопки в левой части экрана, соответствующие этажам, имеют имена `callBut1`, `callBut2`, ..., `callBut11`. Клип, представляющий такую кнопку, содержит два кадра с метками "`_free`" и "`_busy`". В первом кадре находится изображение активной кнопки. Во втором кадре находится изображение кнопки с индикацией, свидетельствующей о том, что вызов был уже выполнен.
- Панели управления лифтов оформлены в виде клипов с именами `liftPanel1`, `liftPanel2`, ..., `liftPanel6`. На монтажной линейке клипа, представляющего панель управления лифтом, находятся кнопки с именами `sendBut1`, `sendBut2`, ..., `sendBut11`.

Сценарий кадра в FLA-файле, реализующий графический интерфейс и визуализацию системы лифтов

```
var num:Number = 6; // Число лифтов
var floors:Number = 11; // Число этажей
var scale:Number = 40; // Коэффициент масштаба (для перемещения клипов)
var _this = this; // Ссылка на текущую монтажную линейку
for (var i = 1; i <= num; i++) {
        // Создаем лифты, позиционируя их на 6 этаже и задавая для всех
        // одинаковую скорость. Для каждого клипа, визуализирующего лифт,
        // определяем метод обработчика onPress, так чтобы при нажатии на
        // лифт выводилась панель с кнопками, позволяющими отправить его
        // на любой этаж
        this["pLift" + i] = new PassangerLift(6, 700);
        this["lift"+i]._y=450 - this["pLift" + i].currentPosition *
        scale;
        this["lift" + i].txt.text = this["pLift" + i].currentFloor;
        this["lift" + i].onPress = function() {
                _this["liftPanel" + this._name.substr(4)]._visible = true;
                this.enabled = false;
        };
        // По умолчанию отключаем видимость всех панелей управления
        // лифтов
        this["liftPanel" + i]._visible = false;
        // При перемещении лифта
        this["pLift" + i].onElevate = function() {
                // Отключаем возможность вызова лифта и видимость его
                // панели управления, так как лифт занят. Свойство id - это
                // идентификационный номер объекта PassangerLift,
                // позволяющий связать клип, представляющий лифт, с
                // соответствующим ему объектом. Поскольку каждый объект
                // PassangerLift имеет свой уникальный id, такое
                // соответствие всегда будет однозначным.
                _this["lift" + this.id].enabled = false;
                _this["liftPanel" + this.id]._visible = false;
                // Визуализируем перемещение лифта, изменяя положение
                // соответствующего клипа
                _this["lift"+this.id]._y=450-this.currentPosition * scale;
                // Выводим номер текущего этажа в лифте, после того как
                // лифт прибыл на него (или миновал)
                _this["lift" + this.id].txt.text = this.currentFloor;
        };
        // Когда лифт прибыл на заданный этаж
        this["pLift" + i].onStop = function() {
```

```
                // Активизируем кнопку вызова лифта на этом этаже и
                // отключаем ее индикацию
                _this["callBut" + this.currentFloor].enabled = true;
                _this["callBut" + this.currentFloor].gotoAndStop("_free");
                // Включаем возможность вызова панели управления лифта
                _this["lift" + this.id].enabled = true;
        };
        // При нажатии кнопки в панели управления лифтом отправляем его на
        // соответствующий этаж и отключаем отображение панели
        for (var k = 1; k <= floors; k++) {
                this["liftPanel" + i]["sendBut" + k].onPress = function()
{
                        var num = this._parent._name.substring(9);
                        var floor = Number(this._name.substring(7));
                        _this["pLift" + num].send(floor);
                        this._parent._visible = false;
                };
        }
}
// При нажатии кнопки вызова этажа на этаж инициируем вызов статического
// метода call(), отправляющего на этаж ближайший свободный лифт
for (var i = 1; i <= floors; i++) {
        this["callBut" + i].onPress = function() {
                var floor = Number(this._name.substr(7));
                // Если вызов выполнен успешно...
                if (PassangerLift.call(floor)) {
                        // Делаем кнопку на этаже неактивной, так как
                        // лифт отправлен туда
                        this.enabled = false;
                        // Включаем индикацию кнопки
                        this.gotoAndStop("_busy");
                }
        };
}
```

Глава 26



Трехмерное моделирование и анимация

Все законы — имитация реальности.

Метазакон Лилли

Flash — это программа, предназначенная для работы с двумерной графикой. Тем не менее, используя возможности ActionScript в сочетании с определенным математическим аппаратом, можно смоделировать трехмерные объекты и визуализировать их в Flash Player. Конечно, поскольку возможности проигрывателя достаточно скромны, невозможно программным образом сконструировать в Flash полноценную трехмерную сцену с освещением, текстурами и т. д. Однако использование более простых динамически созданных трехмерных элементов может найти широкое применение при реализации разнообразных декоративных анимационных эффектов, анимированных элементов навигации и создании игр.

В этой главе мы дадим общее представление о базовых понятиях машинной графики, связанных с трехмерными преобразованиями и проекциями. Затем на основе этих теоретических сведений мы создадим класс, позволяющий задавать положение точек в трехмерном пространстве, визуализировать их и выполнять с этими точками преобразования поворота, переноса и масштабирования. В конце главы будут приведены примеры практической реализации возможностей созданного класса, демонстрирующие организацию интерактивного управления трехмерным объектом с помощью клавиатуры и мыши.

Понятие о матрицах

Математический аппарат, используемый в машинной графике для вывода и преобразования точек на плоскости и в трехмерном пространстве, основывается на операциях с матрицами. В связи с этим в начале данной главы мы кратко рассмотрим несколько базовых понятий, связанных с матрицами, владение которыми будет необходимо для понимания излагаемого далее материала.

Определение матрицы и ее представление в ActionScript

Матрицей называется таблица, составленная из произвольного набора величин, которые располагаются на пересечениях ее строк и столбцов. Величины, составляющие матрицу, называются элементами матрицы. Совокупность элементов, расположенных на одной горизонтальной прямой, составляет *строку* матрицы. Совокупность элементов, расположенных на одной вертикальной прямой, составляет *столбец* матрицы. Каждый элемент матрицы имеет фиксированное положение, которое определяется номером строки и столбца, на пересечении которых он находится. Местоположение элемента обозначается с помощью индекса, состоящего из двух чисел. Согласно общепринятым обозначениям элемент матрицы обозначается a_{ij}, где i — номер строки, j — номер столбца, на пересечении которых находится элемент. Нумерация элементов матрицы начинается с единицы. Приведем пример матрицы, состоящей из 12 элементов:

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{pmatrix}.$$

Размер матрицы обозначается, как произведение числа строк на число столбцов. Так, приведенная выше матрица имеет размер 4×3. Матрица, у которой количество строк совпадает с количеством столбцов, называется квадратной. Количество строк квадратной матрицы называется ее порядком.

Две матрицы называются согласованными, если количество столбцов первой матрицы совпадает с количеством строк второй. Таким образом, матрицы размером $m \times n$ и $n \times p$ являются согласованными.

Каждую строку матрицы можно рассматривать как матрицу размером $1 \times n$. Строки матрицы с числом элементов n могут использоваться для хранения координат некоторых точек в пространстве размерностью n. В этом случае они часто называются векторами строк.

Для оперирования матрицами в ActionScript удобно использовать многомерные массивы *(см. разд. "Многомерные массивы" главы 9)*. Здесь и далее для представления матриц мы будем использовать двумерные массивы, элементами которых являются массивы, представляющие векторы строк. Поясним это на конкретном примере. Пусть у нас имеется квадратная матрица, имеющая порядок 4, следующего вида:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Для того чтобы сконструировать эту матрицу, в Flash сначала создадим массивы для хранения векторов строк:

```
var v1:Array = [1, 0, 0, 0];
var v2:Array = [0, 1, 0, 0];
var v3:Array = [0, 0, 0, 0];
var v4:Array = [0, 0, 0, 1];
```

Теперь создадим двумерный массив, представляющий исходную матрицу, поместив в него массивы векторов строк.

```
var M:Array = [v1, v2, v3, v4];
```

В дальнейшем, используя термин "матрица" в контексте ActionScript, мы будем подразумевать двумерный массив, элементами которого являются массивы векторов строк.

Умножение матриц

Главной (и единственной) матричной операцией, с которой нам придется иметь дело при выполнении рассматриваемых далее пространственных преобразований, является умножение. Ввиду чрезвычайной важности этой операции приведем правила выполнения матричного умножения.

Матрица, стоящая в выражении умножения слева, считается *первой*, матрица, стоящая в выражении справа, считается *второй*. Для выполнения умножения двух матриц необходимо, чтобы они являлись согласованными, то есть, чтобы количество столбцов первой матрицы совпадало с количеством строк второй. Так, при умножении матриц $n_1 \times m_1 \cdot n_2 \times m_2$, необходимо выполнение условия $m_1 = n_2$. Количество строк результирующей матрицы всегда совпадает с количеством строк первой матрицы, а количество столбцов результирующей матрицы всегда совпадает с количеством столбцов второй.

Вычисление элемента c_{ij} матрицы, полученной в результате умножения матрицы A размерностью $m \times n$ на матрицу B размерностью $m \times p$, выполняется по следующей формуле:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj},$$

где $i = 1, 2, 3, \ldots, m$ и $j = 1, 2, 3, \ldots, p$.

В словесной форме правило умножения двух матриц можно сформулировать следующим образом. Для получения элемента i-й строки и j-го столбца результирующей матрицы необходимо умножить каждый элемент i-й строки первой матрицы на каждый элемент j-го столбца второй матрицы и сложить полученные результаты.

Проиллюстрируем это правило следующим примером:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} =$$

$$= \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \\ a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31} & a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} & a_{41}b_{13} + a_{42}b_{23} + a_{43}b_{33} \end{pmatrix}.$$

При умножении нескольких матриц умножение выполняется попарно слева направо. Сначала первая матрица умножается на вторую, затем результирующая матрица умножается на третью и т. д.

Чрезвычайно важно иметь в виду, что операция умножения матриц не является коммутативной. Это означает, что произведение матриц $(A)\times(B)$ в общем случае не равно произведению $(B)\times(A)$. Таким образом, при умножении матриц существенную роль играет последовательность умножения матриц.

Для матричного умножения выполняется ассоциативный закон. Это означает, что очередность выполнения умножения не оказывает влияния на его результат. Произведение трех матриц $(AB)C$ равно произведению $A(BC)$. Таким образом, неважно, в какой очередности выполняется произведение матриц, однако чрезвычайно важна последовательность расположения матриц в выражении матричного умножения.

Класс *Calculus*

При выполнении пространственных преобразований нам придется многократно выполнять некоторые однотипные математические вычисления. В связи с этим имеет смысл поместить все процедуры, используемые для выполнения вычислений, в отдельный класс. Этот класс мы назовем `Calculus`.

Основной вычислительной операцией, с которой нам придется иметь дело, будет умножение матриц. В большинстве случаев мы будем вычислять результат умножения двух матриц. Тем не менее практические задачи могут потребовать вычисления произведения нескольких матриц. Умножение нескольких матриц представляет собой просто последовательное перемножение двух матриц, поэтому вначале мы создадим метод `matrixMultiply2()`, вычисляющий произведение двух матриц, а затем создадим более общий метод `matrixProduct()`, предназначенный для вычисления произведения произвольного числа матриц. Программный код с подробными комментариями представлен в следующем листинге. Исходный файл класса читатель сможет найти в файле Calculus.as, расположенном в папке Глава 26 на сопровождающем книгу компакт-диске.

Класс `Calculus`. Математические операции, используемые для выполнения трехмерных преобразований

```
class Calculus {
// Метод для умножения двух согласованных матриц произвольного размера.
// В качестве параметров принимает массивы, представляющие исходные
// матрицы. Произведение выполняется в той же последовательности, в какой
// передаются параметры.
public static function matrixMultiply2(ar1:Array, ar2:Array):Array {
        //Сохраняем размеры исходных матриц в переменных
        var r1 = ar1.length;//Кол-во строк левой матрицы
        var c1 = ar1[0].length;//Кол-во столбцов левой матрицы
        var r2 = ar2.length;//Кол-во строк правой матрицы
        var c2 = ar2[0].length;//Кол-во столбцов правой матрицы
        //trace("Размер левой м-цы: "+r1+"x"+c1);
        //trace("Размер правой м-цы: "+r2+"x"+c2);
        //Проверка выполнения условия допустимости умножения матриц
        if (c1 != r2) {
                trace("Ошибка при умножении матриц! Количество столбцов
                левой матрицы не совпадает с количеством строк правой");
                return null;
        }
        // Массив для хранения матрицы, полученной в результате
        // умножения двух исходных матриц
        var product = new Array();
        //Заполнение этого массива пока пустыми векторами строк
        for (var row = 0; row < r1; row++) {
                product.push([]);
        }
        //Строки результирующей матрицы
        for (var i = 0; i < r1; i++) {
                //Столбцы результирующей матрицы
                for (var j = 0; j < c2; j++) {
                // Пока помещаем в массив нулевое значения с тем,
                // чтобы далее его можно было модифицировать с помощью
                // составной операции для вычисления значения элемента
                // результирующей матрицы
                        product[i][j] = 0;
                        // Этот цикл позволяет вычислять элементы
                        // результирующей матрицы, поэлементно умножая
                        // строку первой матрицы на столбец второй.
```

```
                    for (var col = 0; col < c2; col++) {
                        product[i][j] += ar1[i][col] * ar2[col][j];
                    }
                }
        }
        //Возвращаем результирующую матрицу
        return (product);
}
```

Теперь, используя созданный выше метод `matrixMultiply2()`, создадим метод `matrixProduct()`, позволяющий вычислять произведение произвольного количества матриц (при условии, что они являются согласованными). Метод `matrixProduct()` в качестве параметров будет принимать двумерные массивы, представляющие матрицы, которые должны быть перемножены. В объявлении этого метода отсутствуют формальные параметры, поскольку число фактических параметров не ограничено. Принцип действия метода заключается в том, что он последовательно слева направо вычисляет произведение переданных ему в качестве фактических параметров матриц, используя метод `matrixMultiply2()`. Матрица, полученная в результате умножения матриц, переданных двумя первыми параметрами, умножается на матрицу, переданную третьим параметром, и т. д.

```
// Метод для умножения произвольного количества согласованных матриц
// произвольного размера
public static function matrixProduct():Array {
        var product:Array = arguments[0];
        for (var i = 1; i < arguments.length; i++) {
                product = matrixMultiply2(product, arguments[i]);
        }
        return product;
}
```

Наряду с методами для выполнения умножения матриц мы поместим в класс `Calculus` еще несколько вспомогательных методов, выполняющих вычисления, которые нам потребуются в дальнейшем. Их код очевиден и не нуждается в дополнительном описании.

```
// Метод для вычисления синуса заданного в градусах угла
public function sin(angle:Number):Number {
        return (Math.sin(angle/180*Math.PI));
}
// Метод для вычисления косинуса заданного в градусах угла
public function cos(angle:Number):Number {
        return (Math.cos(angle/180*Math.PI));
}
```

```
/* Метод для приведения значения, переданного ему в качестве параметра
угла, в диапазон [0, 360). Создание этого метода обусловлено
необходимостью исключения из значения угла полных оборотов, а также тем,
что пользователю проще ориентироваться на положительные значения угла.
В результате действия данного метода значение угла -40° будет
преобразовано в +320°, а значение 725° будет преобразовано в 5°. */
public function normalize360(angle:Number):Number{
        if (angle >= 360) {
                angle = angle % 360;
        }
        if (angle < 0) {
                if (angle <= -360) {
                        angle = 360 + angle % 360;
                }
                else {
                        angle = 360 + angle;
                }
        }
        return angle;
}
}
```

Изображение и преобразование трехмерных точек

В трехмерном пространстве точка представляется тремя координатами x, y, z. Эти координаты могут быть определены, как элементы матрицы размерностью 1×3 следующего вида:

$$\begin{pmatrix} x & y & z \end{pmatrix}.$$

Для определения координат нескольких точек можно объединить соответствующие им векторы строк в общую матрицу, каждая строка которой будет содержать координаты определенной точки. Так, координаты точек $A(1, 0, 0)$, $B(0, 1, 0)$, $C(0, 0, 1)$ могут быть представлены с помощью квадратной матрицы 3-го порядка:

$$X_{A,\,B,\,C} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Матричное представление позволяет изящно выразить многие физические операции, связанные с преобразованиями координат. В общем случае процедура преобразования координат точек в матричном представлении состоит в следующем. Имеется матрица X, содержащая исходные координаты точек. Имеется матрица T, элементами которой являются числовые значения, определенные таким образом, что умножение матрицы X на матрицу T дает результирующую матрицу, строки которой со-

держат преобразованные координаты исходных точек. Матрица T называется матрицей преобразования и выступает в роли геометрического оператора. Выполнение преобразования координат точек осуществляется путем умножения матрицы X с их исходными координатами на матрицу T соответствующего преобразования. Полученная в результате такого умножения матрица X'' будет содержать координаты, соответствующие положению точек после выполнения преобразования. Так, допустим, у нас имеется матрица X, содержащая исходные координаты точек. Требуется повернуть эти точки на некоторый угол вокруг оси Y. Для выполнения этой операции можно воспользоваться матрицей поворота вокруг этой оси T_{Ry}. Тогда преобразование можно записать следующим образом:

$$X \cdot T_{Ry} = \begin{pmatrix} x_1 & y_1 & z_1 \\ \dots & \dots & \dots \\ x_n & y_n & z_n \end{pmatrix} \times T_{Ry} = \begin{pmatrix} x'_1 & y'_1 & z'_1 \\ \dots & \dots & \dots \\ x'_n & y'_n & z'_n \end{pmatrix} = X''.$$

Для различных преобразований можно определить соответствующие матрицы преобразований. Здесь мы рассмотрим выполнение следующих трехмерных преобразований:

- повороты вокруг координатных осей;
- масштабирование по координатным осям;
- перенос вдоль координатных осей.

При выполнении всех операций с трехмерными объектами мы будем использовать правую систему координат, изображение которой представлено на рис. 26.1. Положительное направление поворота определяется в соответствии с правилом правой руки и отмечено на рис. 26.1. Правило правой руки заключается в следующем: необходимо отставить большой палец правой руки и совместить его с положительным направлением оси, вокруг которой выполняется поворот. После этого необходимо согнуть пальцы правой руки. Их круговое движение будет указывать положительное направление поворота вокруг данной координатной оси.

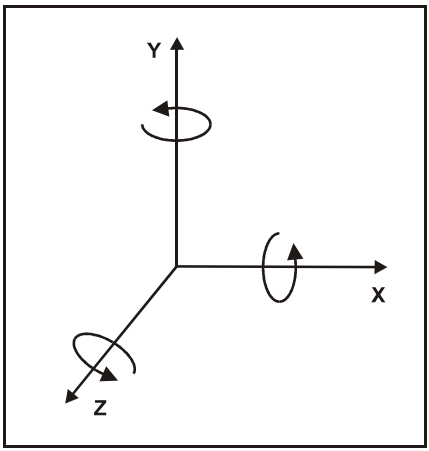


Рис. 26.1. Правая система координат и положительные направления поворотов

Как уже было сказано, координаты трехмерной точки задаются вектором, состоящим из трех компонентов. Поэтому число столбцов в матрице с координатами точек равно трем. Для выполнения преобразований поворота и масштабирования используются квадратные матрицы третьего порядка. Однако выполнение пространственного переноса с помощью матрицы преобразования размерностью 3×3 оказывается невозможным (объяснение этого факта заинтересованный читатель сможет найти в литературе по проективной геометрии или в специальной литературе по машинной графике).

Решение этой проблемы заключается в переходе к так называемым однородным координатам. Любая точка A с координатами (x, y, z) может быть представлена в однородных координатах, имеющих следующий вид:

$$(x', y', z', w),$$

где $x = x'/w$, $y = y'/w$, $z = z'/w$, а w — любое ненулевое вещественное число.

Трехмерные декартовы координаты точки получаются из однородных делением на множитель w.

Возникновение однородных координат исторически было связано с необходимостью выполнения преобразования координат из пространства размерностью n в пространство размерностью $n + 1$. Для любой точки в пространстве размерностью n существует неограниченное количество значений однородных координат, соответствующих точке в пространстве размерности $n + 1$. Для большей наглядности приведем пример с двумерной точкой на плоскости. Если точка A имеет декартовы координаты (2, 5), то ее однородные координаты могут совершенно эквивалентно задаваться тройками чисел (2, 5, 1), (4, 10, 2), (6, 15, 3), (2000, 5000, 1000) и т. д. Приведенные тройки значений можно рассматривать как координаты двумерной точки A в трехмерном пространстве. Как видно, однородные координаты не сохраняют однозначности, поскольку одной и той же точке A соответствует множество значений однородных координат.

При решении задач, связанных с выполнением трехмерных преобразований, как правило, в качестве однородных координат некоторой точки $A(x, y, z)$ берется набор однородных координат следующего вида:

$$(x, y, z, 1).$$

Здесь значение множителя w равно единице, поэтому остальная тройка координат соответствует исходным декартовым координатам и не требует дополнительных вычислений. Таким образом, для представления координат точек в однородных координатах достаточно просто добавить в матрицу, содержащую соответствующие векторы строк, дополнительный четвертый столбец, состоящий из единиц. Так, матрица $X_{A,\,B,\,C}$, приведенная в начале данного раздела, в однородных координатах будет записываться следующим образом:

$$X'_{A,\,B,\,C} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Введение однородных координат позволяет выполнять преобразования переноса, однако приводит к увеличению размерности исходной матрицы и матриц преобра-

зования, что, в свою очередь, оборачивается увеличением объема вычислений. Для большей эффективности и быстродействия программы имеет смысл оперировать с обычными координатами до тех пор, пока не возникнет необходимость в переходе к однородным. Для выполнения чистых поворотов и масштабирования однородные координаты не требуются. Тем не менее, если эти преобразования выполняются в сочетании с преобразованиями переноса, переход к однородным координатам необходим. В дальнейшем изложении и в приведенном далее примере мы будем сразу же использовать однородные координаты как более общий и универсальный (хотя в некоторых случаях менее эффективный с точки зрения быстродействия) способ.

Матрицы различных трехмерных преобразований представлены в табл. 26.1.

Таблица 26.1. *Матрицы трехмерных преобразований в однородных координатах*

Преобразование	**Матрица**
Поворот вокруг оси X на угол φ	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Поворот вокруг оси Y на угол φ	$\begin{pmatrix} \cos\varphi & 0 & -\sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Поворот вокруг оси Z на угол φ	$\begin{pmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Масштабирование по осям X, Y, Z Коэффициенты S_x, S_y, S_z задают масштаб по соответствующим осям. Для увеличения масштаба значение коэффициента должно быть больше единицы	$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Перенос вдоль осей X, Y, Z Коэффициенты l_x, l_y, l_z задают расстояние переноса вдоль соответствующих осей	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l_x & l_y & l_z & 1 \end{pmatrix}$

Следует иметь в виду, что при выполнении поворотов чрезвычайно важной является последовательность их выполнения. Поскольку трехмерное вращение осуществляет-

ся путем перемножения матриц, которое не обладает свойством коммутативности, преобразования поворотов также некоммутативны. В связи с этим, например, для того, чтобы восстановить исходное состояние повернутого объекта, необходимо выполнить повороты в обратном направлении в последовательности, обратной к последовательности выполнения поворотов.

При необходимости выполнения сразу нескольких преобразований можно вначале вычислить общую матрицу преобразований и затем умножить на нее исходную матрицу с координатами точек. Такой подход позволит уменьшить объем вычислений.

Проецирование трехмерных точек на плоскость

При визуализации трехмерного объекта зритель видит его на двумерной плоскости экрана монитора (или иного выводного устройства). Для формирования двумерного представления трехмерного объекта необходимо выполнить процедуру, которая называется проецированием. В результате проецирования трехмерного объекта получается плоская геометрическая проекция. Для выполнения проецирования можно воспользоваться матрицей проецирования, подобно тому, как мы использовали матрицу преобразования для выполнения пространственных преобразований. Так, для того чтобы выполнить проецирование трехмерных точек, координаты которых содержатся в матрице X, на плоскость $z = 0$, нужно умножить эту матрицу на матрицу проецирования P_z.

$$X \times P_z = \begin{pmatrix} x_1 & y_1 & z_1 & 1 \\ \dots & \dots & \dots & \dots \\ x_n & y_n & z_n & 1 \end{pmatrix} \times P_r = \begin{pmatrix} x'_1 & y'_1 & 0 & 1 \\ \dots & \dots & \dots & \dots \\ x'_n & y'_n & 0 & 1 \end{pmatrix} = X'_p$$

Здесь X'_p — это матрица с координатами точек проекции.

В результате умножения на матрицу проецирования всегда получается матрица, один из первых трех столбцов которой является нулевым. Двумерные координаты спроецированных точек находятся в ненулевых столбцах. В приведенном примере плоскостью проекции являлась плоскость $z = 0$, поэтому в результирующей матрице нулевым оказался столбец, соответствующий координате z. Первые два столбца здесь содержат двумерные координаты проекции исходных трехмерных точек.

Механизм получения плоской геометрической проекции демонстрируется на рис. 26.2. Проецирование выполняется с помощью прямолинейных лучей, называемых проекторами, идущих из центра проекции через каждую точку объекта до пересечения с плоскостью проекции. Плоская проекция получается в результате пересечения проекторов с плоскостью проекции. Различают перспективную и параллельную проекции. Перспективная проекция получается в случае, если центр проекции находится на конечном расстоянии от плоскости проекции и проекторы пересекаются в центре проекции (см. рис. 26.2). Параллельная проекция получается, когда центр проекции находится в бесконечно удаленной точке трехмерного пространства и все проекторы являются параллельными (рис. 26.3).

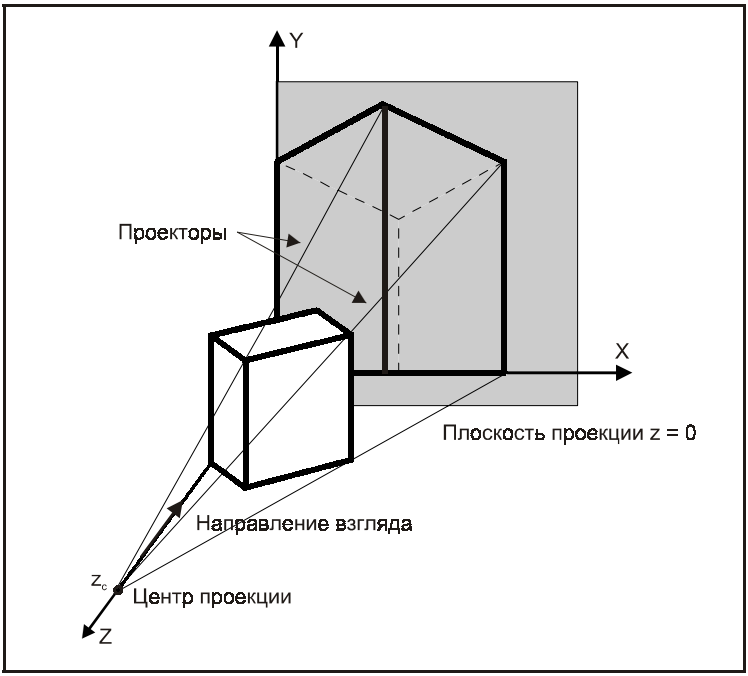


Рис. 26.2. Получение плоской геометрической проекции

Существует несколько видов проекций, каждый из которых обладает определенными специфическими характеристиками и имеет свою область применения. Мы воспользуемся самой простой из параллельных проекций, которая называется ортографической и широко применяется в черчении. *Ортографическая проекция* — это проекция на одну из координатных плоскостей. Мы будем использовать ортографическую проекцию на плоскость $z = 0$. Матрица ортографического проецирования на плоскость $z = 0$ имеет следующий вид:

$$P_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

На рис. 26.3 представлены ортографические проекции куба со срезанным углом на координатные плоскости, перпендикулярные различным координатным осям.

Как видно из приведенного рисунка, одна ортографическая проекция не может дать представления о форме объекта. Для получения четкого представления о форме трехмерного объекта необходимо выполнить несколько его поворотов и/или перемещений так, чтобы были видны хотя бы три грани. Затем результат проецируется на видовую плоскость с помощью матрицы ортографической проекции, то есть с центром проекции, расположенным в бесконечности. Это называется аксонометрическим проецированием. В рассматриваемом нами далее примере предполагается

постоянное выполнение преобразований трехмерных точек (анимация вращения и переноса) с последующим проецированием. Это даст прекрасную возможность рассмотреть объект и составить четкое представление о его форме.

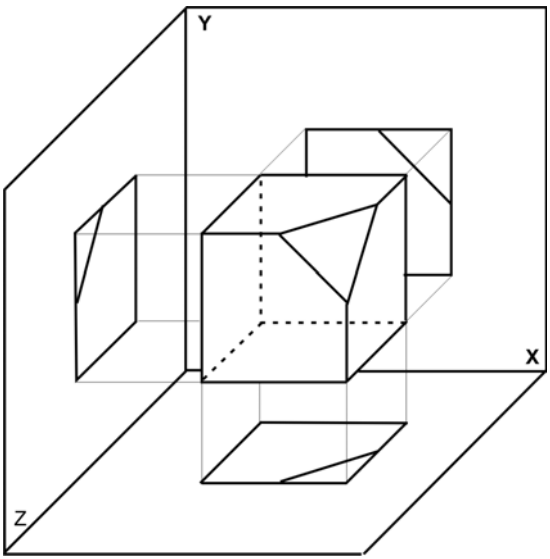


Рис. 26.3. Ортографические проекции куба со срезанным углом на плоскости $y = 0$, $z = 0$ и $x = 0$

Особенностью человеческого зрения является перспектива. Видимый нами мир отличается от реально существующего, поскольку видимые размеры объектов зависят от их удаленности от глаза наблюдателя. Это объясняется тем, что человек видит предметы, когда исходящие от них прямолинейные лучи сходятся в глазу. Ортографическая проекция предполагает бесконечную удаленность центра проецирования и, соответственно, не учитывает эффект перспективного искажения размеров объектов.

Для того чтобы придать изображению бóльшую реалистичность, можно выполнить перспективное преобразование координат спроецированных точек. Таким образом, визуализация трехмерного объекта на двумерной плоскости будет выполняться в два этапа.

1. Вначале мы выполняем ортографическое (параллельное) проецирование на видовую плоскость, используя для этого приведенную выше матрицу.
2. После этого мы преобразуем полученные двумерные координаты проекции, чтобы добиться эффекта перспективного искажения.

Описанная процедура называется перспективным проецированием. Перспективное преобразование предполагает, что центр проекции удален от плоскости проекции на конечное расстояние. При этом все параллельные прямые объекта, не параллельные плоскости проекции, пересекаются в точке, называемой точкой схода. В зависимо-

сти от положения объекта относительно плоскости проекции количество точек схода может быть различным: от 1 до 3. Все точки схода лежат на одной прямой линии, называемой линией горизонта. В зависимости от числа точек схода различают одно-, двух- и трехточечную перспективную проекцию. Для выполнения одноточечного перспективного проецирования на плоскость $z = 0$ можно воспользоваться следующими формулами:

$$x' = x/(r \times z + 1);$$
$$y' = y/(r \times z + 1).$$

Здесь x и y — это двумерные координаты точки, полученные в результате ортографического проецирования, z — это исходная координата точки по оси z. Параметр r называется одноточечным перспективным множителем. Его значение определяет расстояние от центра проекции до плоскости проекции (в данном случае до плоскости $z = 0$) в соответствии со следующей зависимостью:

$$r = -1/z_c,$$

где z_c — это расстояние от центра проекции до плоскости проекции $z = 0$.

Отметим, что с увеличением z_c множитель r стремится к нулю. При нулевом значении r эффект перспективы отсутствует (это означает, что центр проекции находится на бесконечно удаленном расстоянии, в результате чего мы получаем обычную параллельную проекцию). Та часть объекта, которая лежит в плоскости проекции, отображается с истинными размерами и формой без перспективного искажения. Все остальные части объекта искажаются. Если объект и плоскость проекции находятся по одну сторону от центра проекции, то элементы объекта, расположенные дальше от центра проекции, имеют меньший размер (что соответствует нашему физическому восприятию). Если объект и плоскость проекции находятся по разные стороны от центра проекции, изображение оказывается перевернутым. Все эти закономерности читатель может проверить самостоятельно с помощью примера на сопровождающем книгу компакт-диске, сохраненного в файле 26.1.swf, описание реализации которого приведено далее в этой главе.

Двух- и трехточечные перспективные преобразования могут выполняться путем предварительного поворота трехмерного объекта вокруг соответствующих координатных осей и последующего проецирования с одноточечным перспективным преобразованием.

В качестве краткого резюме перечислим последовательность операций, используемых для визуализации трехмерного объекта.

1. Задаем трехмерные координаты точек объекта, формируя исходную матрицу.
2. При необходимости выполнения преобразований (поворотов, переносов и т. д.) умножаем исходную матрицу на соответствующую матрицу преобразования.
3. Выполняем ортографическое проецирование результирующей матрицы на плоскость $z = 0$.
4. Выполняем одноточечное перспективное преобразование двумерных координат спроецированных точек для получения перспективного искажения.

Создание класса *Vertex3D*

В этом разделе мы рассмотрим процесс создания класса `Vertex3D`, позволяющего размещать точки в трехмерном пространстве, выполнять с ними пространственные преобразования, проецировать точки на плоскость и визуализировать их. Возможности, предоставляемые описываемым классом, могут быть применены при создании разнообразных декоративных анимационных эффектов, элементов навигации и логотипов.

Вначале кратко сформулируем постановку задачи и опишем функциональность, которой должен будет обладать класс `Vertex3D`. Основные функциональные требования к классу следующие.

- Экземпляр класса `Vertex3D` представляет собой набор из произвольного количества трехмерных точек.
- Координаты трехмерных точек задаются в единицах, которые затем масштабируются с помощью заданных пользователем коэффициентов. Так, например, если точка должна быть помещена в координаты (100, 100, 100), ее исходные координаты задаются как (1, 1, 1), а затем вводится масштабирующий коэффициент 100. Это обеспечивает удобство при задании координат точек, поскольку пользователю значительно проще оперировать простыми числами, а не реальными размерами. Важно только, чтобы координаты всех точек задавались в едином масштабе, поскольку для всех точек набора используется единый масштабирующий коэффициент. Задание координат точек осуществляется с применением обычных координат. Переход к однородным координатам будет выполняться в классе автоматически.
- Возможность выполнения трехмерного вращения набора точек вокруг координатных осей *X*, *Y* и *Z* на заданный угол в соответствии с правилом правой руки.
- Возможность выполнения трехмерного переноса точек вдоль осей *X*, *Y* и *Z* на заданное расстояние.
- Возможность масштабирования набора точек по осям *X*, *Y* и *Z*.
- Возможность восстановления исходного состояния точек.
- Выполнение ортографического проецирования преобразованных точек на плоскость $z = 0$ с применением одноточечного перспективного преобразования. Коэффициент перспективного преобразования задается пользователем.
- Спроецированные трехмерные точки могут быть визуализированы с помощью заданного пользователем символа, расположенного в библиотеке документа. При этом порядок размещения экземпляров символа, изображающих точки, в стеке графических объектов должен определяться их удаленностью от плоскости проекции. Клип, соответствующей точке, расположенной ближе к зрителю, должен выводиться на переднем плане. Масштаб клипов также должен соответствовать их удаленности от зрителя (эффект перспективы).
- Класс должен обладать возможностью, позволяющей соединить любые спроецированные точки набора прямой линией заданного цвета. Линии проводятся из заданной точки к указанным точкам набора для текущего расположения точек.

Приведенный далее программный код класса Vertex3D снабжен пояснениями, которые даны обычным шрифтом. Исходный код с подробными комментариями читатель сможет найти в файле Vertex3D.as, расположенном в папке Глава 26 на сопровождающем книгу компакт-диске.

Класс Vertex3D. Трехмерные преобразования и визуализация точек

Класс Vertex3D наследует методы класса Calculus, поскольку для осуществления трехмерных преобразований требуется вычислять произведение матриц. В классе Calculus также содержатся методы sin() и cos(), используемые для вычисления функций синуса и косинуса, принимающие в качестве параметров значение в градусах. Это избавляет нас от необходимости применения более громоздкой записи вызова методов класса Math и преобразования градусов в радианы.

```
class Vertex3D extends Calculus {
// ********** ОБЪЯВЛЕНИЕ СВОЙСТВ *************
```

Свойство init3D будет использоваться для хранения массива с исходными координатами точек. Используя это свойство, можно будет легко восстановить первоначальное расположение точек.

```
private var init3D:Array;
```

Свойство transformed3D используется для хранения массива с преобразованными координатами точек. После каждого преобразования (поворот, перенос, масштабирование) новые координаты точек будут записываться в массив transformed3D. Для визуализации преобразования точки, хранящиеся в массиве transformed3D, должны быть спроецированы на плоскость.

```
public var transformed3D = new Array();
```

Свойство projection используется для хранения массива с двумерными координатами спроецированных точек. Значения, хранящиеся в этом свойстве, используются при позиционировании клипов, визуализирующих точки.

```
public var projection:Array = new Array();
```

Свойство zPerspective хранит значение одноточечного перспективного коэффициента. Это значение определяет, на каком расстоянии от плоскости проекции находится центр проекции, расположенный на оси *Z*. Значение этого свойства должно быть отрицательным, поскольку центр проекции должен находиться на положительной полуоси *OZ*. При задании значения перспективного множителя необходимо задавать его так, чтобы центр проекции находился на определенном расстоянии от ближайшей к нему точке в положительном направлении оси *Z*. Напомним соотношение, позволяющее определить одноточечный перспективный множитель, исходя из заданного положения центра проекции:

$$\texttt{zPerspective} = -1 / Z_c,$$

где Z_c — положение центра проекции на оси *Z*.

Нулевое значение свойства zPerspective соответствует отсутствию перспективного преобразования. Это значение будем использовать по умолчанию

```
private var zPerspective:Number = 0;
```

Свойство `skinOn` — это флаг, определяющий, используется ли визуализация точек с применением символов библиотеки, или нет.

```
private var skinOn = false;
```

Как уже говорилось, координаты точек задаются в единицах, которые затем масштабируются с требуемым коэффициентом. Свойства `xscale` и `yscale` задают масштабные коэффициенты для координат точек по осям *X* и *Y* соответственно. Масштабный коэффициент по оси *Z* будет рассчитываться как среднее арифметическое от значений `xscale` и `yscale`.

```
private var xscale:Number;
private var yscale:Number;
```

Поворот точек будет выполняться на заданный угол относительно нулевого угла. Поскольку после выполнения первого неполного поворота текущий угол точек будет отличен от нуля, для их повторного поворота до заданного угла фактический поворот должен быть выполнен на угол, представляющий собой разность заданного и текущего. Следующие свойства будут хранить значения, соответствующие фактическим поворотам вокруг координатных осей.

```
private var thetaX:Number = 0;
private var thetaY:Number = 0;
private var thetaZ:Number = 0;
```

Аналогично следующие свойства будут хранить значения фактических смещений точек вдоль координатных осей.

```
private var distX:Number = 0;
private var distY:Number = 0;
private var distZ:Number = 0;
```

Свойство `container` — это ссылка на клип, в котором будут размещены все графические элементы, используемые для визуализации точек. Этот клип может быть создан в рабочей среде или динамически. Значением этого свойства также может быть `_root` или `_level0`.

```
private var container:Object;
```

```
// ********** ОБЪЯВЛЕНИЕ КОНСТРУКТОРА *************
```

Конструктор принимает четыре параметра, первые три из которых являются обязательными.

- Параметр `holder` — это ссылка на клип-контейнер, в котором будут размещены все графические элементы, используемые для визуализации точек. Этот клип должен быть создан и позиционирован в FLA-документе самим пользователем. Точка регистрации клипа `holder` будет являться проекцией начала трехмерной системы координат.
- Параметры `scaleX` и `scaleY` — это масштабирующие коэффициенты по *X* и по *Y*, на основании которых будет выполняться масштабирование заданных значе-

ний координат точек. Масштабирование будет выполняться только на этапе проецирования. Все преобразования будут выполняться с исходными координатами. Здесь не требуется задание масштабирующего коэффициента по оси *Z*, так как ось *Z* перпендикулярна плоскости проекции, а при проецировании значение имеют только координаты *X* и *Y*. При выводе информации о фактических значениях смещения точек будем рассчитывать масштабный коэффициент по оси *Z*, как среднее из значений `scaleX` и `scaleY`.

- Параметр `perspective` задает коэффициент одноточечного перспективного преобразования. Этот параметр является необязательным. В случае его отсутствия будет использовано значение по умолчанию 0 (отсутствие перспективы). Значение этого параметра не должно быть положительным (положительные значения приведут к эффекту обратной перспективы), поэтому в конструкторе мы инициализируем свойство `zPerspective`, присваивая ему отрицательное значение, по модулю соответствующее значению параметра `perspective`.

```
function Vertex3D(holder, scaleX, scaleY, perspective) {
    //trace("Класс Vertex3D загружен");
    container = holder;
    xscale = scaleX;
    yscale = scaleY;
    if(perspective){
        zPerspective = -Math.abs(perspective);
    }
```

Для того чтобы каждый объект `Vertex3D` мог представлять индивидуальный набор точек, необходимо в конструкторе инициализировать собственное свойство `init3D`. При дальнейшем задании координат точек массив `init3D` будет заполнен этими значениями.

```
    init3D = new Array();
}
```

Используя `get`-методы, создадим свойства, предназначенные только для чтения и позволяющие определить фактические значения поворотов и смещений точек. Данные свойства являются элементами внешнего программного интерфейса и предоставляют сведения о том, на какой угол или на какое расстояние были смещены или повернуты точки.

```
// ***************** ОБЪЯВЛЕНИЕ getter/setter МЕТОДОВ ******************
public function get rotationX():Number {
    return thetaX;
}
public function get rotationY():Number {
    return thetaY;
}
public function get rotationZ():Number {
```

```
        return thetaZ;
}
public function get offsetX():Number {
        return distX * xscale;
}
public function get offsetY():Number {
        return distY * yscale;
}
public function get offsetZ():Number {
        return Math.round(distZ * (xscale + yscale) / 2);
}

// ********** ОБЪЯВЛЕНИЕ МЕТОДОВ *************
```

Метод `construct3D()` предназначен для "конструирования" набора точек в трехмерном пространстве. В качестве параметров этому методу должны быть переданы векторы-строки, соответствующие координатам точек. Поскольку количество точек в наборе, представляемом объектом `Vertex3D`, может быть произвольным, в объявлении этого метода формальные параметры не указываются. Фактические параметры (передаваемые методу при вызове) должны являться массивами, состоящими из трех элементов. В принципе, можно было бы возложить функции этого метода на конструктор, однако это привело бы к усложнению его формы в связи с увеличением количества разнородных параметров. Поэтому было решено задачу конструирования набора точек возложить на самостоятельный метод.

```
public function construct3D():Array {
        for (var i = 0; i < arguments.length; i++) {
                // Проверка корректности задания параметров
                if (arguments[i].length != 3) {
                        trace("Для " + i + "-й точки задано неверное↵
                        количество координат. Требуется ровно три.");
                        return [];
                }
```

Поскольку наш класс предполагает возможность выполнения с точками преобразований переноса, это означает необходимость перехода к однородным координатам. В частности, матрица исходных координат точек (массив `init3D`) должна быть дополнена четвертым столбцом, состоящим из единиц. Требовать задания координат точек с применением однородных координат изначально неразумно, поскольку их назначение может оставаться для пользователя абсолютно неочевидным. Поэтому переход к однородным координатам мы выполним автоматически, увеличив размерность матрицы, переданной с помощью параметров в цикле. В принципе, применение однородных координат необходимо только при выполнении пространственного переноса. Для выполнения поворотов вокруг координатных осей можно оперировать обычными координатами, используя матрицы меньшей размерности. Это позволило

бы уменьшить объем вычислений. Тем не менее здесь для простоты мы сразу вводим однородные координаты, предполагая постоянную потребность в выполнении переноса.

```
            // Представляем переданные в качестве параметров координаты
            // точек в однородных координатах
            arguments[i].push(1);
            init3D.push(arguments[i]);
        }
```

Для каждой *i*-ой точки создаем в клипе `container` пустой клип с именем, состоящим из слова `vertexClip` и порядкового номера, соответствующего номеру точки. Нумерация точек осуществляется на основании порядка задания их координат. Так, первая точка — это точка, чей координатный вектор представлен первой строкой матрицы (первым параметром), вторая точка — это точка, координаты которой задаются вторым параметром, и т. д. В каждый из этих клипов в дальнейшем может быть помещен графический объект из библиотеки для визуализации точки. В каждом из этих клипов также можно будет нарисовать линию для визуализации ребер объекта. Поскольку графическое представление каждой точки будет находиться в отдельном клипе, можно легко изменять порядок наложения этих клипов (перемещая их на определенный уровень) в соответствии с положением точек (клип, соответствующий дальней точке, должен быть помещен на задний план). Аналогично — с ребрами. Все ребра, исходящие из одной вершины, будут помещены в соответствующий ей клип $\texttt{vertexClip}_i$.

```
            for (var i = 0; i < init3D.length; i++) {
                container.createEmptyMovieClip("vertexClip" + i,↳
                container.getNextHighestDepth());
            }
            // Инициализируем transformed3D, так как это свойство
            // применяется при проецировании. До выполнения
            // преобразований transformed3D будет содержать массив с
            // исходными координатами точек. Это необходимо для
            // проецирования и отображения исходного положения точек
            transformed3D = init3D;
            // Проецируем точки, представленные массивом transformed3D
            project3D();
            // Возвращаем массив с исходными координатами
            return transformed3D;
}
```

Следующие методы применяются для выполнения поворотов набора точек относительно координатных осей. Поворот выполняется на переданное в качестве параметра значение в градусах относительно нулевого положения. Это означает, что при вызове метода с определенным значением параметра все точки в наборе будут повернуты так, чтобы угол их поворота относительно нулевого положения соответствовал переданному в качестве параметра значению. При выполнении поворота не-

обходимо учитывать текущий угол поворота. Фактический поворот будет выполнен на значение, представляющее собой разность заданного угла и угла, соответствующего текущему положению точек.

Для вычисления координат точек в результате поворота умножаем матрицу `transformed3D`, хранящую текущие координаты точек, на матрицу поворота, хранящуюся в локальной переменной. Результирующую матрицу снова записываем в свойство `transformed3D`. Затем выполняем проецирование точек и визуализацию поворота.

```
// Поворот вокруг оси X до заданного угла
public function rotateX(theta:Number):Array {
        // Нормализуем переданное значение так, чтобы оно попадало в
        // диапазон [0, 360)
        theta = normalize360(theta);
        // Вычисляем, на какой угол относительно текущего значения
        // поворота нужно повернуть точки до достижения заданного угла
        var thetaDelta = theta - thetaX;
        // Обновляем текущее значение угла поворота
        thetaX += thetaDelta;
        // Матрица поворота вокруг X (в однородных координатах)
        var r1:Array = [1, 0, 0, 0];
        var r2:Array = [0, cos(thetaDelta), sin(thetaDelta), 0];
        var r3:Array = [0, -sin(thetaDelta), cos(thetaDelta), 0];
        var r4:Array = [0, 0, 0, 1];
        var Rx = [r1, r2, r3, r4];
        // Результирующая матрица с преобразованными координатами
        transformed3D = matrixProduct(transformed3D, Rx);
        // Выполняем проецирование и визуализацию точек
        project3D();
        return transformed3D;
}
// Поворот вокруг оси Y до заданного угла
public function rotateY(theta:Number):Array {
        theta = normalize360(theta);
        var thetaDelta = theta - thetaY;
        thetaY += thetaDelta;
        var r1:Array = [cos(thetaDelta), 0, -sin(thetaDelta), 0];
        var r2:Array = [0, 1, 0, 0];
        var r3:Array = [sin(thetaDelta), 0, cos(thetaDelta), 0];
        var r4:Array = [0, 0, 0, 1];
        var Ry = [r1, r2, r3, r4];
        transformed3D = matrixProduct(transformed3D, Ry);
```

```
        project3D();
        return transformed3D;
}
// Поворот вокруг оси Z до заданного угла
public function rotateZ(theta:Number):Array {
        theta = normalize360(theta);
        var thetaDelta = theta - thetaZ;
        thetaZ += thetaDelta;
        var r1:Array = [cos(thetaDelta), sin(thetaDelta), 0, 0];
        var r2:Array = [-sin(thetaDelta), cos(thetaDelta), 0, 0];
        var r3:Array = [0, 0, 1, 0];
        var r4:Array = [0, 0, 0, 1];
        var Rz = [r1, r2, r3, r4];
        transformed3D = matrixProduct(transformed3D, Rz);
        project3D();
        return transformed3D;
}
```

Локальное масштабирование вдоль координатных осей выполняется с помощью соответствующей матрицы преобразования. Масштабные коэффициенты для масштабирования вдоль осей *X*, *Y* и *Z* передаются в качестве параметров.

```
// Локальное масштабирование вдоль координатных осей
public function scale(scaleX:Number,scaleY:Number, scaleZ:Number):Array {
        var s1:Array = [scaleX / 100, 0, 0, 0];
        var s2:Array = [0, scaleY / 100, 0, 0];
        var s3:Array = [0, 0, scaleZ / 100, 0];
        var s4:Array = [0, 0, 0, 1];
        var S = [s1, s2, s3, s4];
        transformed3D = matrixProduct(transformed3D, S);
        project3D();
        return transformed3D;
}
```

Следующие методы применяются для реализации пространственных переносов в направлении осей *X*, *Y* и *Z*. Параметры, принимаемые этими методами, определяют местоположение, в которое должны быть перенесены точки по отношению к началу отсчета координат. Для удобства применения величины смещений, передаваемые параметрами, задаются отмасштабированными значениями. Так, например, если координаты точек задавались в единичной системе координат, то значения смещений будут задаваться для координат, к которым было применено масштабирование, с применением заданных масштабных коэффициентов. Пользователю при переме-

щении удобнее оперировать реальными размерами (например, 178), чем исходными (например, 0,178).

При формировании матрицы преобразования переданные пользователем значения смещений нужно масштабировать в ту же размерность, в которой задавались координаты точек. Фактическое смещение, как и поворот, будет выполнено на расстояние, соответствующее разности заданного пользователем значения и текущего местоположения точек.

```
public function shiftX(dist) {
        dist /= xscale; // Масштабируем
        var distDelta = dist - distX;
        distX += distDelta;
        // Матрица переноса по X
        var t1 = [1, 0, 0, 0];
        var t2 = [0, 1, 0, 0];
        var t3 = [0, 0, 1, 0];
        var t4 = [distDelta, 0, 0, 1];
        var T = [t1, t2, t3, t4];
        transformed3D = matrixProduct(transformed3D, T);
        project3D();
}
public function shiftY(dist) {
        dist /= yscale;
        var distDelta = dist - distY;
        distY += distDelta;
        var t1 = [1, 0, 0, 0];
        var t2 = [0, 1, 0, 0];
        var t3 = [0, 0, 1, 0];
        var t4 = [0, distDelta, 0, 1];
        var T = [t1, t2, t3, t4];
        transformed3D = matrixProduct(transformed3D, T);
        project3D();
}
public function shiftZ(dist) {
        dist /= (xscale + yscale) / 2;
        var distDelta = dist - distZ;
        distZ += distDelta;
        var t1 = [1, 0, 0, 0];
        var t2 = [0, 1, 0, 0];
        var t3 = [0, 0, 1, 0];
```

```
        var t4 = [0, 0, distDelta, 1];
        var T = [t1, t2, t3, t4];
        transformed3D = matrixProduct(transformed3D, T);
        project3D();
}
```

Метод `reset()` позволяет переместить точки в исходное положение. Для этого достаточно просто использовать матрицу, хранящуюся в свойстве `init3D`.

```
public function reset() {
        transformed3D = init3D;
        // Обнуляем текущие значения поворотов и смещений
        thetaX = thetaY = thetaZ = distX = distY = distZ = 0;
        project3D();
}
```

Метод `setNodeSkin()` активизирует визуализацию точек, используя для этого символ библиотеки, идентификатор связи которого передается методу в качестве параметра. Действие этого метода заключается в том, что он динамически присоединяет символ из библиотеки, помещая его экземпляр в каждый клип `vertexClip`, соответствующий точке из набора. Каждый присоединенный экземпляр получает имя `graphicHolder`. Конфликта имен не происходит, поскольку каждый клип `graphicHolder` находится в своем клипе `vertexClip`.

```
public function setNodeSkin(graphicHolder:String) {
        // Если визуализация еще не выполнена...
        if (!skinOn) {
                for (var i = 0; i < init3D.length; i++) {
                        container["vertexClip" + i].attachMovie(↳
                        graphicHolder, "graphicHolder", 1);
                        }
                skinOn = true;
                // Позиционируем клипы vertexClip, соответствующие точкам,
                // и устанавливаем масштаб прикрепленных к ним клипов
                // graphicHolder
                render3D();
        }
}
```

Метод отключает визуализацию точек, удаляя клипы `graphicHolder` из всех клипов `vertexClip`.

```
public function removeNodeSkin() {
        // Если визуализация активна
        if (skinOn) {
```

```
        for (var i = 0; i < init3D.length; i++) {
        container["vertexClip" + i].graphicHolder. removeMovieClip();
        }
        skinOn = false;
        }
}
```

Метод `joinNodes()` используется для рисования линии, соединяющей точку, номер которой задается первым параметром с точками, номера которых задаются другими параметрами. Нумерация точек осуществляется в соответствии с порядком задания их координат с помощью метода `construct3D()`. Точка, координаты которой были переданы методу `construct3D()` первым параметром, имеет номер 0; точка, координаты которой были переданы вторым параметром, имеет номер 1, и т. д.

Метод `joinNodes()` принимает следующие параметры:

- `node` — номер точки, от которой проводятся линии;
- `lineThickness`, `rgb`, `alpha` — соответственно, толщина, цвет и прозрачность линии;
- остальные (фактические) параметры, которые будут переданы данному методу при его вызове, — это числа, указывающие номера точек, с которыми должна быть соединена точка с номером `node`. Поскольку точка с номером может быть соединена с произвольным количеством точек, формальные параметры в объявлении метода отсутствуют. Для обращения к фактическим параметрам, начиная с пятого, в теле метода используется объект `arguments`.

Линии, исходящие из точки с номером `node`, рисуются в соответствующем этой точке клипе `vertexClip`. Поскольку взаимным наложением клипов `vertexClip` можно легко управлять, задавая их уровень в стеке в соответствии с их удаленностью от плоскости проекции, это автоматически дает возможность сделать так, чтобы все линии, исходящие из точки, расположенной ближе к зрителю, располагались перед линиями, исходящими из точки, расположенной дальше. Это позволяет придать изображению бóльшую реалистичность.

Метод `joinNodes()` соединяет точки линиями только для их конкретного взаимного расположения. Если расположение точек меняется, данный метод должен быть вызван снова для выполнения перерисовки. Таким образом, принцип его использования похож на принцип применения методов динамического рисования.

```
public function joinNodes(node:Number, lineThickness:Number, rgb:Number,↳
alpha:Number) {
        // Очищаем клип vertexClip для очередного рисования
        container["vertexClip" + node].clear();
        for (var i = 4; i < arguments.length; i++) {
                container["vertexClip" + node].moveTo(0, 0);
                container["vertexClip" + node].lineStyle(lineThickness,↳
                rgb, alpha);
```

```
            // Соединяем точку с номером node с точкой с номером
            // arguments[i]. Точки соединяются так, что концы линий
            // располагаются в точках регистрации клипов vertexClip
            container["vertexClip" + node].lineTo(container[
            "vertexClip" + arguments[i]]._x - container["vertexClip" +
            node]._x, container["vertexClip" + arguments[i]]._y -
            container["vertexClip" + node]._y);
            }
    }
```

Метод `disjoinAllNodes()` удаляет все соединительные линии, просто проходя в цикле все клипы `vertexClip` и очищая их динамически нарисованное содержимое.

```
    public function disjoinAllNodes() {
            for (var i = 0; i < init3D.length; i++) {
                    container["vertexClip" + i].clear();
            }
    }
```

Метод `render3D()` используется для позиционирования клипов `vertexClip` и задания масштаба клипов `graphicHolder`, визуализирующих точки. Координаты клипов `vertexClip` задаются на основании значений, полученных в результате проецирования. Координаты спроецированных точек хранятся в двух первых столбцах матрицы `projection`.

```
private function render3D() {
    for (var i = 0; i < init3D.length; i++) {
            // Позиционирование
            container["vertexClip" + i]._x = projection[i][0];
            container["vertexClip" + i]._y = -projection[i][1];
```

Наряду с позиционированием клипов `vertexClip` на экране необходимо определить порядок их наложения в соответствии с удаленностью представляемых ими точек от центра проекции. Поскольку в нашем случае плоскость проекции перпендикулярна оси *Z*, достаточно разместить клипы в стеке графических объектов в соответствии со значениями *z*-координаты представляемых ими точек. Клип `vertexClip`, соответствующий точке с меньшим значением *z*-координаты, должен быть помещен на нижележащий уровень по отношению к уровню, на котором находится клип `vertexClip`, соответствующий точке с бóльшим значением *z*-координаты. Значения *z*-координаты точек хранятся в третьем столбце матрицы `transformed3D` (элемент с номером 2).

```
            container["vertexClip" + i].swapDepths(Math.round(
            transformed3D[i][2] * 1000) - i);
            if (skinOn) {
```

Для более реалистичной передачи эффекта перспективы необходимо установить масштаб клипов `graphicHolder`, визуализирующих точки в соответствии с удален-

ностью последних от центра проекции. Для вычисления значений масштаба мы воспользовались формулами одноточечного перспективного преобразования. Полученные значения представлены в исходной размерности. Для приведения их к реальным размерам необходимо умножить полученные значения на масштабные коэффициенты `xscale` и `yscale`. Результирующие значения мы берем по модулю, поскольку задание отрицательных значений свойствам `_xscale` и `_yscale` приведет к эффекту отражения, что в нашем случае совершенно нежелательно.

```
            container["vertexClip" + i]["graphicHolder"]._xscale =↳
            Math.abs(xscale/(zPerspective*transformed3D[i][2]+1));
            container["vertexClip" + i]["graphicHolder"]._yscale =↳
            Math.abs(yscale/(zPerspective*transformed3D[i][2]+1));
                }
            }
}
```

Метод `project3D()` выполняет ортографическое проецирование на плоскость $Z = 0$ с последующим одноточечным перспективным преобразованием и затем визуализирует результат, вызывая метод `render3D()`.

```
private function project3D() {
        // Матрица ортографического проецирования
```

Третий и четвертый столбцы реальной матрицы ортографического проецирования не оказывают влияние на искомые координаты X и Y проекции, поэтому, мы отбросили эти столбцы в целях облегчения вычислений. В наших вычислениях будет использоваться только часть истинной матрицы размерностью 4×2.

```
        var Pz:Array = [[1, 0], [0, 1], [0, 0], [0, 0]];
```

В результате произведения матрицы `transformed3D` на матрицу проецирования `Pz` мы получим матрицу размерностью 4×4, в которой первые два столбца будут соответствовать координатам спроецированных точек на плоскости проекции по осям X и Y. Остальные столбцы не представляют для нас интереса.

```
        projection = matrixProduct(transformed3D, Pz);
```

Теперь необходимо только выполнить одноточечное перспективное преобразование координат, представленных первыми двумя столбцами полученной матрицы `projection`. Результирующие значения и будут окончательными координатам проекций исходных трехмерных точек на плоскость $Z = 0$.

```
        for (var i = 0; i < projection.length; i++) {
            projection[i][0] *= xscale / (zPerspective *↳
            transformed3D[i][2] + 1);
            projection[i][1] *= yscale / (zPerspective *↳
            transformed3D[i][2] + 1);
            }
        // Визуализируем результаты
```

```
        render3D();
        return projection;
        }
}
```

На рис. 26.4 представлен алгоритм, отражающий функциональность класса `Vertex3D`.

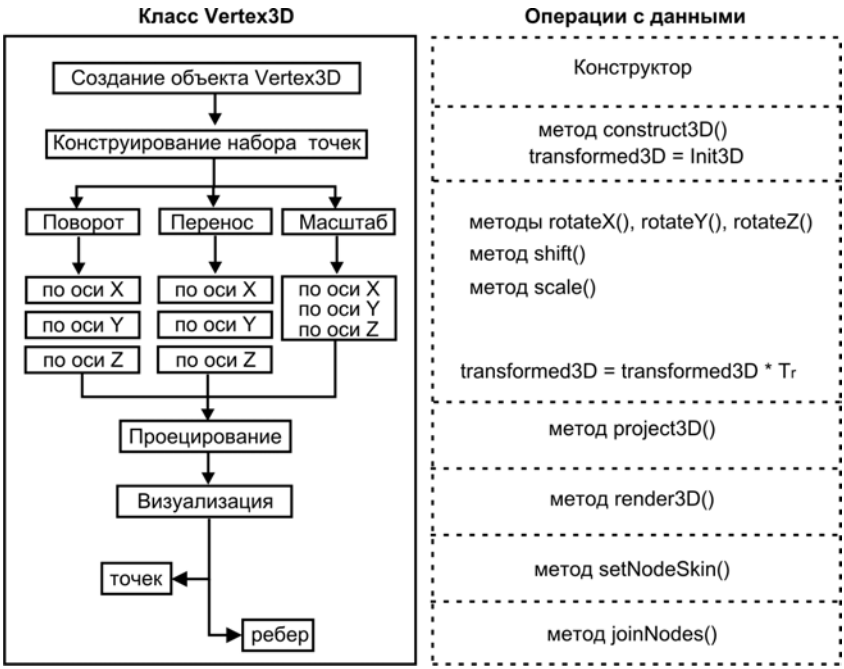


Рис. 26.4. Функциональная схема класса `Vertex3D`

Пример реализации: управление центрированным кубом со срезанным углом с помощью клавиатуры

В качестве демонстрации применения созданного нами класса `Vertex3D` создадим набор точек, соответствующих вершинам куба с отсеченным углом, и обеспечим возможности интерактивного управления этим кубом с помощью клавиатуры и элементов интерфейса.

Вначале сформулируем основные функциональные требования к нашему примеру.

- Пользователь должен иметь возможность осуществлять вращение куба вокруг координатных осей с помощью клавиатуры и задавая точные значения углов по-

ворота по соответствующим осям. Фактические значения углов поворота должны выводиться в текстовые поля.

- Пользователь должен иметь возможность осуществлять перенос куба вдоль координатных осей с помощью клавиатуры и задавая точные значения смещений вдоль соответствующих осей. Фактические значения смещений должны выводиться в текстовые поля.
- Пользователь должен иметь возможность включить/отключить визуализацию вершин и ребер куба.
- Пользователь должен иметь возможность восстановить исходное положение куба.

Для обеспечения описанной функциональности нам потребуется создать в FLA-файле следующие элементы интерфейса (рис. 26.5).

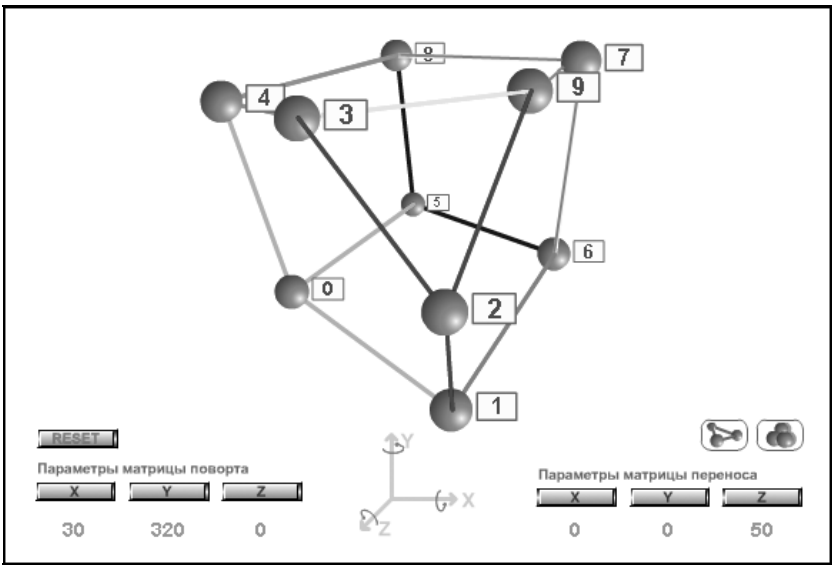


Рис. 26.5. Пример, демонстрирующий возможности управления трехмерными точками и их визуализации

- Текстовые поля, в которых будут выводиться текущие значения угла поворота вокруг осей *X*, *Y* и *Z*. Эти поля также будут использоваться для ввода значений угла поворота пользователем, поэтому данные поля должны иметь тип **input**. Данным текстовым полям мы присвоим имена `xRotTxt`, `yRotTxt` и `zRotTxt`.
- Текстовые поля, предназначенные для вывода текущих значений смещений вдоль осей *X*, *Y* и *Z* и задания этих значений пользователем. Данным текстовым полям мы присвоим имена `xOffsetTxt`, `yOffsetTxt` и `zOffsetTxt`.
- Для того чтобы введенное пользователем в текстовое поле значение угла поворота или смещения вступило в силу и отразилось на внешнем виде куба, для каждого поля будет создана кнопка, нажатие на которую вызовет соответствующие

изменения положения точек и визуализацию этих изменений. Кнопки, соответствующие текстовым полям, связанным с поворотом вокруг осей, мы назовем xRotationBut, yRotationBut и zRotationBut. Кнопки, соответствующие текстовым полям, связанным со смещением вдоль осей, мы назовем xOffsetBut, yOffsetBut и zOffsetBut.

- Для восстановления исходного положения куба создадим кнопку с именем resetBut.
- Для обеспечения возможности включения и отключения визуализации точек создадим клип-кнопку skinBut. Данная кнопка будет использоваться в качестве флага, поэтому на ее монтажной линейке мы создадим два кадра, снабдив их метками "_on" и "_off". Кадр с меткой "_off" будет содержать изображение кнопки в состоянии, соответствующем отключению визуализации точек, а кадр с меткой "_on" — изображение кнопки при активизации визуализации.
- Для обеспечения возможности включения и отключения визуализации ребер, соединяющих точки, аналогичным образом создадим клип-кнопку, также используемую в качестве флага и состоящую из двух кадров с метками "_on" и "_off".
- В библиотеке документа необходимо создать символ _node_, который будет использоваться для визуализации точек. Этому символу мы присвоим идентификатор связи node. Данный символ может содержать любое изображение или анимацию. Важно иметь в виду, что этот символ будет позиционирован так, что его центр регистрации будет совпадать с координатами спроецированной точки. В нашем примере в качестве содержимого такого символа будет использовано изображение круга, окрашенного радиальным градиентом. Для того чтобы рядом с изображением точки вывести ее порядковый номер, на монтажной линейке клипа мы создадим динамическое текстовое поле с именем nodeTxt. Как было сказано ранее, символ, используемый для визуализации точек, будет динамически помещен в клип-контейнер, соответствующий данной точке. Имя клипа-контейнера состоит из слова vertexClip и порядкового номера, соответствующего номеру представляемой им точки. Поэтому для того, чтобы в текстовом поле nodeTxt выводился номер точки, которую визуализирует символ _node_, на его монтажную линейку достаточно поместить следующий код:

```
nodeTxt.text = _parent._name.substring(10);
```

Теперь можно приступить к написанию кода, реализующего поставленную нами задачу, который мы поместим в первый кадр основной монтажной линейки FLA-документа. Готовый пример читатель сможет найти на сопровождающем книгу компакт-диске в файле 26.1.fla, расположенном в папке Глава 26.

Листинг 26.1. Управление центрированным кубом со срезанным углом с помощью клавиатуры

```
// Создаем клип, в который будут помещены объекты, визуализирующие точки
this.createEmptyMovieClip("holder", this.getNextHighestDepth());
holder._x = 275;
```

```
holder._y = 150;
// Создаем объект, представляющий точки в пространстве
var cube3D:Vertex3D = new Vertex3D(holder, 120, 120, -.5);
```

Теперь необходимо задать координаты точек, соответствующих вершинам куба со срезанным углом. Центр нашего куба будет совпадать с началом координат. Для простоты возьмем единичный куб (ширина грани которого составляет 1). Такой единичный центрированный куб со срезанным углом изображен на рис. 26.6. Нумерация вершин куба выполнена в соответствии с порядком их дальнейшей передачи методу `construct3D()` в качестве параметров.

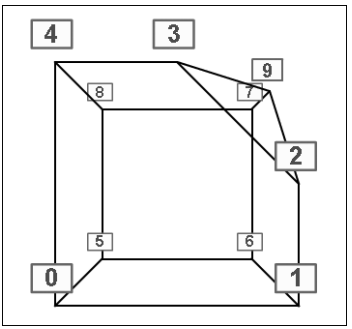


Рис. 26.6. Единичный центрированный куб со срезанным углом.

```
// Сохраняем в переменных координаты точек единичного куба со срезанным
// углом
var v1:Array = [-.5, -.5, .5];
var v2:Array = [.5, -.5, .5];
var v3:Array = [.5, 0, .5];
var v4:Array = [0, .5, .5];
var v5:Array = [-.5, .5, .5];
var v6:Array = [-.5, -.5, -.5];
var v7:Array = [.5, -.5, -.5];
var v8:Array = [.5, .5, -.5];
var v9:Array = [-.5, .5, -.5];
var v10:Array = [.5, .5, 0];
```

Теперь создадим функцию `join()`, которая будет рисовать ребра для нашего куба. Поскольку метод `joinNodes()` соединяет точки только для их текущего местоположения, его необходимо вызывать после каждого изменения положения точек. В связи с этим целесообразно один раз создать функцию, которая выполняет соединение вершин, и затем вызывать ее после каждого очередного преобразования. Для большей наглядности мы нарисуем ребра разными цветами. Это подчеркнет пространственный эффект.

```
function join() {
        // Соединяем нулевую вершину с вершинами 1, 4, 5 зелеными линиями
        cube3D.joinNodes(0, 3, 0x40CC20, 100, 1, 4, 5);
        cube3D.joinNodes(4, 3, 0xFF00FF, 100, 8, 3);
        cube3D.joinNodes(2, 3, 0x0000FF, 100, 1, 3, 9);
        cube3D.joinNodes(7, 3, 0xFFFF00, 100, 6, 8, 9);
        cube3D.joinNodes(5, 3, 0x0F0F0F, 100, 6, 8);
        cube3D.joinNodes(3, 3, 0x00FFFF, 100, 9);
        cube3D.joinNodes(1, 3, 0xFF0000, 100, 6);
}
```

Теперь все готово для того, чтобы вывести изображение куба на экран. Для этого мы вначале должны "сконструировать" его, передав предварительно подготовленные координаты вершин методу `construct3D()`.

```
//Формируем набор трехмерных точек, соответствующих вершинам куба
cube3D.construct3D(v0, v1, v2, v3, v4, v5, v6, v7, v8, v9);
```

Теперь наши точки реально существуют в трехмерном пространстве. Для того чтобы увидеть их, выполним визуализацию с помощью метода `setNodeSkin()`.

```
//Визуализируем вершины, используя клип с идентификатором связи "node"
cube3D.setNodeSkin("node");
```

Теперь наши точки спроецированы на плоскость и визуализированы с помощью графической оболочки, представленной символом библиотеки. Остается только соединить вершины куба ребрами. Для этого вызываем написанную выше функцию:

```
join();
```

Теперь изображение на экране должно выглядеть примерно так, как показано на рис. 26.6. Следующая задача заключается в том, чтобы обеспечить возможности управления точками. Для этого мы введем переменную, устанавливающую шаг поворота и переноса:

```
var speed = 10; // Шаг поворота и переноса
```

Создадим функцию `render()`, которая будет выполнять обновление значений поворота и смещения по осям в соответствующих текстовых полях и переменных. Функция `render()` также будет выполнять отрисовку ребер, если данный режим будет включен.

```
function render() {
        // Вывод параметров поворотов и переносов в текстовые поля
        xRotTxt.text = cube3D.rotationX;
        yRotTxt.text = cube3D.rotationY;
        zRotTxt.text = cube3D.rotationZ;
        xOffsetTxt.text = cube3D.offsetX;
        yOffsetTxt.text = cube3D.offsetY;
```

```
        zOffsetTxt.text = cube3D.offsetZ;
        //Соединение вершин линиями при активном режиме их отображения
        if (!joinoff) join();
}
```

Для осуществления поворотов и переносов по осям на основании введенных пользователем в текстовые поля значений используется код вида:

```
// Поворот вокруг оси X на заданный угол относительно нулевого угла на
// основании значений, указанных в текстовых полях
xRotationBut.onPress = function() {
        cube3D.rotateX(Number(xRotTxt.text));
        render();
};


// Перенос относительно оси X на заданное расстояние на основании
// значений, указанных в текстовых полях
xOffsetBut.onPress = function() {
        cube3D.shiftX(Number(xOffsetTxt.text));
        render();
};
```

Код для выполнения преобразований по остальным осям здесь не приводится, поскольку он аналогичен. Обратите внимание, что после вызова метода, выполняющего преобразование, следует вызов функции `render()`, обновляющей информацию о выполненном преобразовании и перерисовывающей ребра куба.

Для восстановления исходного положения куба нужно вызвать метод `reset()` при нажатии кнопки `resetBut`.

```
resetBut.onPress = function() {
        cube3D.reset();
        render();
};
```

Для реализации поворота и переноса с помощью клавиатуры используются те же самые методы преобразований, которым в качестве параметров достаточно передать текущее значение параметра соответствующего преобразования, увеличенное или уменьшенное на величину шага `speed`. Для выполнения преобразований с помощью клавиатуры используется код следующего вида:

```
this.onEnterFrame = function() {
// Поворот вокруг оси Y по часовой стрелке
        if (Key.isDown(Key.RIGHT)) {
                cube3D.rotateY(speed + cube3D.rotationY);
                render();
        }
```

```
// Поворот вокруг оси Y против часовой стрелки
        if (Key.isDown(Key.LEFT)) {
                cube3D.rotateY(cube3D.rotationY-speed);
                render();
        }
// Перенос вдоль оси X в отрицательном направлении
        if (Key.isDown(100)) {
                cube3D.shiftX(cube3D.offsetX-speed);
                render();
        }
// Перенос вдоль оси X в положительном направлении
        if (Key.isDown(102)) {
                cube3D.shiftX(speed + cube3D.offsetX);
                render();
        }
};
```

Остальные преобразования поворота и переноса выполняются аналогичным образом.

Теперь мы имеем возможность управлять положением куба в пространстве. Следует помнить, что пространственные повороты некоммутативны (вследствие некоммутативности умножения матриц). Это означает, что при выполнении трехмерных поворотов важна последовательность, в которой они производятся. В связи с этим далеко не всегда можно вернуть куб в исходное положение, просто введя нулевые значения в поля, соответствующие поворотам вокруг осей. Для возврата к исходному состоянию необходимо выполнить все преобразования поворотов в обратном порядке. Однако если пользователь выполнял одновременное вращение вокруг разных осей, удерживая нажатыми несколько клавишей, то восстановление исходного положения будет исключено, поскольку повороты вокруг разных осей попеременно выполнялись с шагом в `speed` градусов и восстановить их очередность для выполнения обратного преобразования не представляется возможным. Задав нулевые значения в полях поворотов, пользователь просто выполнит повороты на `-rotationX`, `-rotationY` и `-rotationZ` градусов, однако это вовсе не означает, что куб автоматически вернется в исходное состояние. Для такого случая предусмотрены кнопка `resetBut` и метод `reset()`, обсуждавшиеся ранее.

Теперь нам остается только обеспечить возможность отключения/включения визуализации точек и соединяющих их линий с помощью кнопок `skinBut` и `joinBut`. Соответствующий код приводится далее:

```
// Включение/отключение визуализации вершин
skinBut.onPress = function() {
        if (!skinoff) {
                cube3D.removeNodeSkin();
                skinoff = true;
```

```
                this.gotoAndStop("_off");
        }
        else {
                cube3D.setNodeSkin("node");
                skinoff = false;
                this.gotoAndStop("_on");
        }
};
// Включение/отключение отображение ребер куба
joinBut.onPress = function() {
        if (!joinoff) {
                cube3D.disjoinAllNodes();
                joinoff = true;
                this.gotoAndStop("_off");
        }
        else {
                join("node");
                joinoff = false;
                this.gotoAndStop("_on");
        }
};
```

Пример реализации: управление трехмерной плоскостью с произвольным числом точек с помощью мыши

В предыдущем примере при размещении точек в пространстве мы задавали их координаты вручную. Набор координат, соответствующих вершинам и точкам поверхностей для различных правильных геометрических тел, может быть сформирован с помощью определенного алгоритма. Это оказывается удобным, если количество точек достаточно велико и задавать их координаты вручную нецелесообразно. В качестве иллюстрации такого подхода мы сформируем набор точек, лежащих в одной плоскости в трехмерном пространстве.

Поскольку управление трехмерными точками с помощью клавиатуры уже было описано в предыдущем примере, теперь покажем, как можно организовать управление точками с помощью мыши.

Приведем краткую формулировку задачи.

- Все точки должны лежать в одной плоскости, образуя двумерную матрицу.

- Формирование координат точек должно выполняться исходя из следующих параметров:
  - количества точек по горизонтали и по вертикали;
  - расстояния между точками по горизонтали и по вертикали.
- Графические элементы (клипы), используемые для визуализации точек, должны быть окрашены в цвет, выбранный случайным образом.
- Плоскость, в которой лежат точки, должна вращаться вокруг осей *X* и *Y*, причем скорость и направление ее вращения задаются путем протаскивания курсора с одновременным нажатием левой кнопки мыши. Чем дальше от центра плоскости находится курсор, тем быстрее скорость ее вращения.

Программный, код, реализующий поставленную задачу, приведен в листинге 26.2. Внешний вид трехмерной плоскости, сформированной 35 точками, представлен на рис. 26.7.

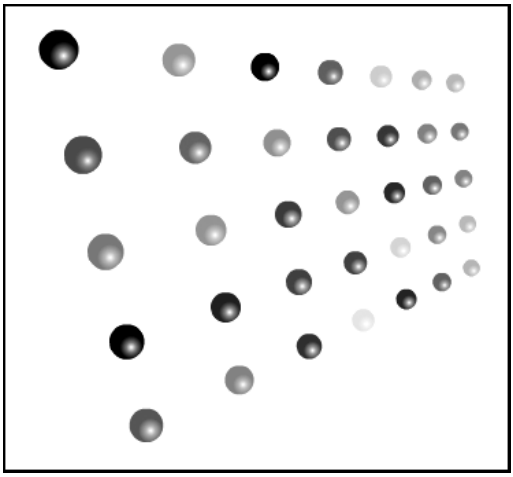


Рис. 26.7. Плоскость, сформированная трехмерными точками

Листинг 26.2. Управление трехмерной плоскостью с заданным числом точек с помощью мыши

```
// Создаем клип, в который будут помещены объекты, визуализирующие точки
this.createEmptyMovieClip("holder", 1);
holder._x = 275;
holder._y = 200;
// Создаем объект, представляющий точки в пространстве
var plane3D = new Vertex3D(holder, 150, 150, -.5);
```

```
var numX = 5; //Количество точек плоскости по горизонтали
var numY = 7; //Количество точек плоскости по вертикали
var distX = .25; //Расстояние между точками по X (задается от 0 до 1)
var distY = .25;//Расстояние между точками по У (задается от 0 до 1)
var xSpeed = ySpeed = 10; //Начальные скорости поворота вокруг осей X и Y
```

Для вычисления координат набора точек плоскости мы создадим функцию `setPlane()`, которая будет формировать массивы с трехмерными координатами точек, исходя из заданных параметров: количества точек по осям X и Y и расстояний между ними. Возвращаемым значением этой функции будет массив, элементами которого являются массивы с вычисленными координатами точек плоскости. Исходные координаты точек плоскости задаются так, что плоскость параллельна плоскости проекции $z = 0$. Центр плоскости находится в начале координат. Координаты точек вычисляются, начиная с левой верхней точки сверху вниз по строкам (сначала вычисляются координаты всех точек первого столбца, затем координаты всех точек второго столбца и т. д.). Порядок нумерации точек представлен на рис. 26.8.

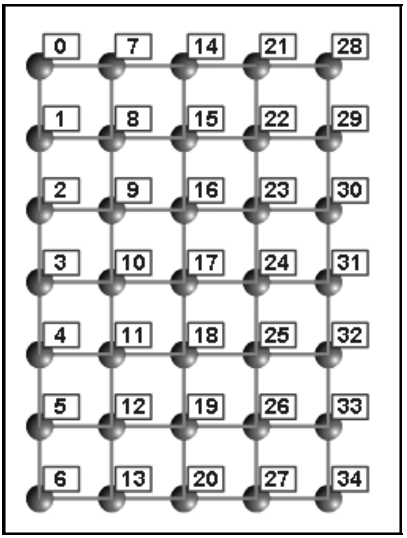


Рис. 26.8. Порядок нумерации точек при формировании плоскости

```
function setPlane(numX, numY, distX, distY):Array {
        var ar:Array = new Array();
        //Расстояние между крайними точками плоскости по оси X
        var sideX = (numX-1)*distX;
        //Расстояние между крайними точками плоскости по оси У
```

```
        var sideY = (numY-1)*distY;
        for (i=0; i<numX; i++) {// По столбцам
                for (j=0; j<numY; j++) { // По строкам
                        // Помещаем координаты точки j-ой строки и i-го
                        // столбца в массив
                        ar.push([-sideX/2+i*distX, -sideY/2+j*distY, 0]);
                }
        }
        // Возвращаем массив, заполненный координатами точек плоскости
        return (ar);
}
```

Наряду с заданием координат точек нам потребуется реализовать возможность их соединения линиями, подобно тому, как это делалось в предыдущем примере. Для этого создадим функцию `join()`, которую нужно будет вызывать после каждого выполнения поворота точек для перерисовывания соединительных линий. На рис. 26.8 хорошо видно, что каждая точка плоскости, кроме точек нижнего ряда и правого столбца, должна быть соединена с соседними точками справа и снизу, точки крайнего справа столбца должны быть соединены с соседними снизу точками, а точки нижнего ряда — с соседними справа. Соединение точек выполняется в соответствии с порядком их нумерации.

```
function join() {
        for (j=0; j<numX; j++) {//По столбцам
                // По рядам, кроме последнего (нижнего)
                for (i=j*numY; i<j*numY+numY-1; i++) {
                // Для крайнего правого столбца
                if(j == numX-1){
                plane3D.joinNodes(i, 2, 0x999999, 100, i+1);
                }
                // Для остальных точек, кроме точек последнего ряда
                else plane3D.joinNodes(i, 2, 0x999999, 100, i+1, i+numY);
                        //Для последнего (нижнего) ряда
                        if (i == j*numY+numY-2) {
                        plane3D.joinNodes(i+1, 2, 0x999999, 100, i+numY+1);
                        }
                }
        }
}
```

Теперь все готово для того, чтобы сконструировать нашу плоскость и визуализировать составляющие ее точки.

```
// Генерируем координаты точек, составляющих плоскость, и сохраняем их в
// массиве.
var points:Array = setPlane(numX, numY, distX, distY);
```

Поскольку методу `construct3D()` в качестве параметров должны быть переданы массивы с трехмерными координатами точек, а у нас все эти массивы сохранены в массиве `points` в качестве его элементов, необходимо использовать метод `Function.apply()`, позволяющий передать параметры функции или методу в виде массива. В качестве первого параметра методу `apply()` будет передана ссылка на объект `plane3D`, поскольку именно из этого объекта вызывается метод `construct3D()`.

```
//Создаем трехмерный объект
plane3D.construct3D.apply(plane3D, points);
//Визуализируем вершины, используя клип с идентификатором связи "node"
plane3D.setNodeSkin("node");
// Соединяем точки линиями при активизации соответствующего режима
if(!joinoff)join();
```

Теперь наша плоскость визуализирована. Для того чтобы заставить ее вращаться вокруг осей *X* и *Y* одновременно, используем следующий код:

```
this.onEnterFrame = function(){
plane3D.rotateX(plane3D.rotationX + xSpeed;
plane3D.rotateY(plane3D.rotationY + ySpeed);
if(!joinoff)join();
};
```

Остается только обеспечить возможность изменения скорости вращения вокруг осей путем перемещения курсора с одновременным нажатием левой кнопки мыши. Горизонтальное смещение курсора будет влиять на скорость и направление поворота вокруг оси *Y*, а вертикальное смещение будет влиять на скорость и направление поворота вокруг оси *X*. Для того чтобы пользователю было проще ориентироваться в этой системе управления, в качестве вспомогательного средства во время нажатия кнопки мыши мы будем выводить прямую линию, соединяющую центр плоскости с курсором и определяющую направление поворота.

```
// При нажатии левой кнопки мыши...
this.onMouseDown = function(){
        // При перемещении курсора с нажатой кнопкой мыши...
        this.onMouseMove = function(){
                // Рисование вспомогательной линии
                this.clear();
```

```
                this.lineStyle(0, 0x996666);
                this.moveTo(this.holder._x, this.holder._y);
                this.lineTo(this._xmouse, this._ymouse);
                // Определение значений скоростей поворота вокруг осей Y и
                // X в зависимости от удаленности курсора от центра
                // плоскости (то есть от точки регистрации клипа holder)
                ySpeed = holder._xmouse * Stage.width/7200;
                xSpeed = holder._ymouse * Stage.height/7200;
                updateAfterEvent();
        };
};
// При отпускании кнопки мыши...
this.onMouseUp = function(){
        // Удаляем вспомогательную линию
        this.clear();
        // Прекращаем действие обработчика onMouseMove
        delete this.onMouseMove;
};
```

Как и в предыдущем примере с кубом, мы можем добавить элементы управления, позволяющие включить/отключить визуализацию точек и соединительных линий. Для обеспечения этих возможностей достаточно просто перенести кнопки `skinBut` и `joinBut` с соответствующим фрагментом кода из листинга 26.1.

Для того чтобы клип с идентификатором связи `node`, используемый для визуализации точки плоскости, был окрашен в случайный цвет, мы прямо на его монтажной линейке (в библиотеке) разместим следующий код:

```
col = Math.random()*0xffffff;
var c1 = new Color(mc1);
c1.setRGB(col);
```

Здесь `mc1` — это клип, расположенный на монтажной линейке клипа с идентификатором связи `node`. В нашем примере `mc1` — это круг, закрашенный радиальным градиентом, левый порог которого (центр) полностью прозрачен. Под клипом `mc1` находится круг белого цвета, который просвечивает сквозь прозрачную область клипа `mc1`, окрашенного случайным цветом, и таким образом создает эффект светового блика.

Клип, который находится в библиотеке и используется для визуализации трехмерных точек, может обладать и более сложным поведением. Так, например, если связать клип, закрашенный радиальным градиентом (в примере этот символ в библиотеке имеет имя graphic), расположенный в клипе с идентификатором связи `node` с классом `RandomColor`, описанным в *главе 25*, то можно получить довольно интерес-

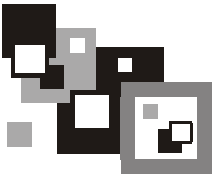
ный эффект (см. файл 26.3.swf на сопровождающем книгу компакт-диске). Здесь клип не просто "привязан" к проекции трехмерной точки, а перемещается вокруг нее по круговой орбите (уже на двумерной плоскости) со случайной скоростью. Это локальное движение вместе с общим движением точек плоскости дает интересный эффект. Единственное изменение, которое было внесено в класс `RandomColor` для данного примера, заключалось в удалении из его конструктора следующих строк:

```
// Позиционируем клип случайным образом
_x = Math.random() * Stage.width;
_y = Math.random() * Stage.height;
```

Это нужно было сделать для того, чтобы центром вращения клипа являлась не просто случайная точка, а проекция соответствующей ему трехмерной точки.

После связывания клипа `graphic`, расположенного в клипе, используемом для визуализации точки, с классом `RandomColor` код с монтажной линейки последнего может быть удален, поскольку случайный цвет клипа задается непосредственно в классе.

Глава 27



Создание пользовательских компонентов

Неточно спланированная программа требует
в три раза больше времени, чем предполагалось;
тщательно спланированная — только в два раза.

Закон Голуба

При создании Flash-проектов разработчику часто приходится использовать типовые функциональные элементы. Эти элементы могут иметь графическое представление, создаваемое в рабочей среде (части интерфейса, визуальные эффекты и т. д.), или представлять собой программные модули, которые обеспечивают заданную функциональность и осуществляют визуализацию за счет использования программных средств. Для многократного использования типовой программной функциональности применяются классы. Однако, как правило, приходится использовать типовые классы, обеспечивающие функциональность в сочетании с графическими элементами, позволяющими ее реализовать. Так, в рассмотренных нами ранее примерах весь графический интерфейс и визуализация поведения, реализуемого классом, выполнялись непосредственно в FLA-файле. Такой подход, безусловно, оправдывает себя при создании нестандартных, единичных (или, как в нашем случае, учебных) проектов. Если же поставленная задача требует многократного применения элемента, объединяющего программную функциональность и графический интерфейс, то одним из решений этой проблемы является создание компонента. Преимущества применения компонентов заключаются в их хорошей переносимости, возможностях настройки параметров работы и внешнего вида при полном сокрытии внутренней реализации.

В этой главе мы рассмотрим общий процесс создания пользовательского компонента DustAndScratches, позволяющего реализовать характерный визуальный эффект старой киноленты с мелькающими пылинками и царапинами. В отличие от предыдущих глав, где теоретический материал предшествовал описанию примеров его практического применения, настоящая глава будет построена как поэтапное описание процесса создания конкретного компонента. Теоретические сведения, относящиеся к определенному этапу, будут приводиться непосредственно по ходу изложения.

Представление о компонентах

Компонент — это элемент рабочей среды Flash, применение которого имеет много общего с использованием обычных символов типа MovieClip. Как и символы, компоненты помещаются в библиотеку документа, а на монтажной линейке располагаются их экземпляры. В действительности компонент представляет собой не обычный символ, а *компилированный клип* (compiled clip), то есть SWF-документ, включающий в свой состав элементы, которые обеспечивают его функциональность и визуальное представление. Тот факт, что компонент, по сути, является SWF-документом, исключает возможность проникновения в него, как это можно легко сделать при работе с обычным символом. Таким образом, все подробности внутренней реализации и графические элементы компонента скрыты от пользователя.

Важнейшей особенностью компонентов является возможность настройки параметров их работы с помощью интерфейса рабочей среды Flash. Для этих целей может быть задействована панель **Properties** или специально предназначенная для работы с компонентами панель **Component Inspector**. Открыть последнюю можно с помощью команды **Window>Development Panels>Component Inspector** (<Alt>+<F7>). При выделении экземпляра компонента на сцене во вкладке **Parameters** панели **Component Inspector** выводится перечень всех параметров, используемых для настройки компонента. Аналогично, при выделении экземпляра компонента на панели **Properties** появляется специальная вкладка **Parameters**, перейдя на которую можно увидеть перечень основных параметров компонента. Создатель компонента может по своему выбору указать, какие параметры должны быть доступны из панели **Properties**. В панели **Component Inspector** выводятся все "открытые" параметры компонента. Таким образом, компонент — это своеобразный "черный ящик". Его содержимое скрыто от пользователя, а результат работы зависит от заданных пользователем значений параметров данного компонента.

Еще одна особенность компонентов заключается в использовании режима предварительного просмотра (live preview), который позволяет непосредственно в рабочей среде отображать компонент таким же, как он будет выглядеть и в конечном фильме. Все изменения параметров компонента, осуществляемые с помощью панелей **Properties** или **Component Inspector**, сразу же отражаются на его внешнем виде. Это делает процесс настройки компонента интуитивно понятным для пользователя. Для включения или отключения режима предварительного просмотра компонента используется команда главного меню **Control>Enable Live Preview**.

Наряду с возможностью настройки компонента посредством интерфейса рабочей среды Flash пользователь может устанавливать значения его параметров и управлять им программным образом, используя программный интерфейс компонента. Реализация программного интерфейса компонента осуществляется его создателем. Автор компонента решает, какие параметры сделать доступными для изменения извне и какой программный инструментарий может потребоваться для эффективного использования компонента.

Экземпляру компонента так же, как и обычному клипу, может быть присвоен идентификатор (instance name), позволяющий программным образом управлять компонентом и настраивать параметры его работы из внешнего кода.

Пользователь не может изменять идентификатор связи компонента, который задается разработчиком на этапе создания компонента.

В отличие от экземпляра символа экземпляру компонента нельзя назначить цветовой эффект. Тем не менее это можно сделать, поместив компонент на монтажную линейку символа и применив цветовую трансформацию к экземпляру этого символа.

В комплект поставки Macromedia Flash MX 2004 входит целый набор встроенных компонентов. Увидеть их можно, открыв панель **Components** с помощью команды главного меню **Window>Development Panels>Components**. Все встроенные компоненты Flash разделены на три категории.

- **Data Components** (Компоненты для работы с данными) — предназначены для управления данными, участвуют в организации связи между компонентами и позволяют реализовывать взаимодействие с серверными службами.
- **User interface (UI) components** (Компоненты пользовательского интерфейса) — полный набор стандартных элементов интерфейса, используемых при создании типовых приложений (например, флажки, переключатели, выпадающие меню, предзагрузчики и т. д.).
- **Media components** (Компоненты для работы с данными) — компоненты, позволяющие осуществлять загрузку и управлять воспроизведением потокового звука в формате MP3 и видео в формате FLV.

Функциональность встроенных компонентов реализована с применением классов, созданных разработчиками Macromedia, полный набор которых также включен в комплект поставки Flash MX 2004 Professional. Классы компонентов находятся в папке mx, которая расположена примерно по такому адресу: **C:\Program files\Macromedia\Flash MX 2004\en\First Run\Classes\mx**. Архитектура классов встроенных компонентов чрезвычайно обширна. Ее описание, как и описание встроенных компонентов, выходит за рамки нашей книги, посвященной возможностям ActionScript, а не описанию созданного с его помощью программного инструментария.

Встроенные компоненты предоставляют широкие функциональные возможности. Тем не менее они имеют существенный недостаток, зачастую приводящий к невозможности их применения — встроенные компоненты занимают довольно большой объем. Это связано с тем, что в архитектуре компонентов широко используется наследование, в связи с чем один компонент может использовать несколько различных классов, которые, в свою очередь, используют возможности других классов и т. д. В результате этого в код компонента включается множество элементов, которые не используются им непосредственно, что приводит к увеличению объема файла. Так, например, включение в свой проект встроенного компонента `CheckBox`, выполняющего роль флажка, увеличит объем файла на 27 Кбайт! Такое значительное увеличение объема является платой за использование "готового решения", в которое включена разнообразная функциональность. Между тем на практике, как правило, требуется, чтобы элемент выполнял достаточно узкий набор функций, поэтому большая часть возможностей компонента просто не используется, и объем файла увеличивается неоправданно. Правда, использование сразу нескольких близких по назначению встроенных компонентов приводит к меньшему суммарному увеличению объема, чем применение каждого из них по отдельности. Это связано с тем, что многие компоненты используют общие классы.

Использование встроенных компонентов оправдывает себя при создании проектов, требующих одновременного и многократного применения типовых элементов со сложной функциональностью, когда минимизация конечного объема файла не является приоритетной задачей. Тем не менее зачастую типовые элементы должны обладать достаточно простой функциональностью или разработчик просто не может себе позволить существенное увеличение объема файла. Это является весомым аргументом против применения встроенных компонентов.

Общий процесс создания компонента

В самом общем случае компонент состоит из двух элементов: символа типа MovieClip, представляющего собой "материальную" оболочку компонента, и связанного с ним класса ActionScript, обеспечивающего функциональность компонента.

При создании пользовательского компонента его базовую функциональность можно обеспечить, используя встроенные классы компонентов, поставляемые вместе с Flash. Этот подход, с одной стороны, позволит легко снабдить пользовательский компонент функциональностью, свойственной встроенным компонентам. С другой стороны, размер такого компонента может оказаться довольно большим, что далеко не всегда допустимо.

Мы рассмотрим создание пользовательского компонента "с нуля". Это позволит лучше понять основные принципы создания компонентов и обеспечит минимальный объем результирующего файла. Единственное исключение будет сделано для встроенного класса `EventDispatcher`, который мы используем для обеспечения возможности генерирования и трансляции событий.

Ключевые этапы создания компонента

Перед тем как приступить к созданию нашего компонента, приведем краткий перечень ключевых этапов создания пользовательского компонента.

1. Создание клипа компонента, который в дальнейшем будет конвертирован в компилированный клип. Это единственная часть работы, осуществляемая непосредственно в рабочей среде с применением монтажной линейки и рисующих инструментов.
2. Объявление класса, реализующего функциональность компонента. Класс компонента должен расширять класс `MovieClip` или другой класс, который, в свою очередь, является подклассом класса `MovieClip`. Это позволит использовать стандартные возможности, свойственные клипам (позиционирование, масштабирование, динамическое создание экземпляров и т. д.) по отношению к компоненту. Компонент может использовать возможности нескольких классов. В этом случае перед объявлением каждого класса все используемые в нем внешние классы должны быть импортированы с помощью ключевого слова `import`. Клипы, входящие в состав компонента, также могут быть связаны с определенными классами, наделяющими их требуемой функциональностью.

3. Объявление свойств класса. Некоторые свойства компонента могут быть скрыты от пользователя, а другие могут выступать в роли параметров, с помощью которых пользователь выполняет настройку компонента. Для определения того, какие свойства будут открыты пользователю, применяются стандартные режимы доступа к членам класса `public` и `private`.
4. Объявление конструктора. В конструктор необходимо, как минимум, поместить вызов метода `init()`, осуществляющего масштабирование и "сборку" компонента из составляющих его элементов.
5. Объявление `get/set`-методов.
6. Для того чтобы открытое свойство отображалось в панели **Properties** и/или в панели **Component Inspector**, служат специальные элементы, которые называются *тегами метаданных* (metadata tags). Теги метаданных размещаются непосредственно внутри программного кода, хотя сами не являются инструкциями ActionScript.
7. Объявление метода `init()`, осуществляющего "сборку" компонента. Данный метод будет вызываться единожды для каждого экземпляра из его конструктора (то есть при каждом создании очередного экземпляра компонента). В обязанности этого метода входит:
   - вызов метода `setSize()`, выполняющего масштабирование экземпляра компонента в соответствии с размерами, установленными пользователем в рабочей среде;
   - вызов метода `createChildren()`, выполняющего присоединение графических элементов, визуализирующих компонент, из библиотеки;
   - вызов метода `arrange()`, выполняющего позиционирование динамически присоединенных элементов компонента в соответствии с их назначением;
   - сокрытие ограничивающей рамки компонента;
   - если компонент должен генерировать и транслировать события, в данный метод можно поместить код, инициализирующий компонент в качестве источника событий;
   - иные действия, которые необходимо выполнить сразу же при создании экземпляра компонента (например, инициализация его собственных свойств).
8. Объявление метода `setSize()`. Этот метод должен обязательно присутствовать у компонента именно с таким именем и стандартным набором параметров. Назначение метода `setSize()` заключается в установке свойств компонента, отвечающих за его размер, в значения, заданные пользователем. Данный метод играет важную роль в обеспечении функционирования режима предварительного просмотра (live preview).
9. Объявление метода `createChildren()`. Как правило, элементы, формирующие графическую оболочку компонента, помещают не непосредственно на его монтажную линейку, а в библиотеку файла компонента. Данный метод присоединяет графические элементы, входящие в состав компонента, из библиотеки с помощью метода `attachMovie()`.

10. Объявление метода `arrange()`. Этот метод позиционирует элементы графической оболочки компонента. В принципе, позиционирование может выполнять и метод `createChildren()`, однако разумнее возложить эту обязанность на самостоятельный метод хотя бы потому, что при изменении размеров компонента далеко не всегда требуется заново присоединять составляющие его элементы.
11. Объявление методов, используемых для генерирования и трансляции событий компонента, реализация которых содержится в классе `EventDispatcher`. Использование "готового" класса в данном случае не приводит к существенному увеличению объема файла, так как его код довольно компактен (менее ста строк).
12. Если компонент использует возможности динамического рисования (для создания собственных графических элементов или иных целей), соответствующий программный код можно вынести в отдельный метод `draw()`.
13. Объявление остальных методов, обеспечивающих функциональность компонента.
14. Объявление метода `onUpdate()`, выполняющего действия, специфичные только для режима предварительного просмотра в рабочей среде Flash.
15. Экспортирование клипа компонента в формат SWC или создание Flash-расширения для дальнейшего распространения готового компонента.

Следует иметь в виду, что, создавая пользовательский компонент, не обязательно использовать именно те имена методов, которые приведены в данном перечне. Имена методов не оказывают влияние на функциональность компонента. Единственное исключение составляют методы `setSize()` и `onUpdate()`, поскольку они задействованы в обеспечении возможности предварительного просмотра компонента в рабочей среде. Реализация метода `setSize()` должна обязательно присутствовать в классе, связанном с компонентом, в то время как реализация метода `onUpdate()` необходима только в том случае, если в режиме предварительного просмотра компонент должен обладать какими-то специфическими возможностями. Используемые нами имена некоторых свойств и большинства методов заимствованы из архитектуры встроенных компонентов, где они выполняют схожие функции.

Проектирование компонента

Процесс создания компонента мы будем рассматривать на примере компонента, который мы назвали `DustAndScratches` (пыль и царапины). Данный компонент используется для имитации характерного эффекта старой кинопленки с мелькающими точками пылинок и царапинами. Подобный эффект находит широкое применение при создании визуального оформления имиджевых и художественных проектов и достаточно часто эксплуатируется, например, в видеоклипах или телевизионных заставках.

Готовый компонент в действии изображен на рис. 27.1. Принцип его работы заключается в создании эффекта хаотического мелькания точек и царапин случайных размеров, формы и цвета (черный или белый). Точки и царапины ведут себя поразному. Точки просто случайным образом появляются и исчезают, в то время как царапины случайным образом появляются, перемещаются сверху вниз, исчезают за нижней границей, после чего вновь появляются у верхней границы в точке, коорди-

наты которой определяются случайным образом. Такое поведение позволяет придать эффекту большую реалистичность. Оценить работу этого эффекта читатель может, открыв файл 27.1.swf в папке Глава 27 на сопровождающем книгу компакт-диске.

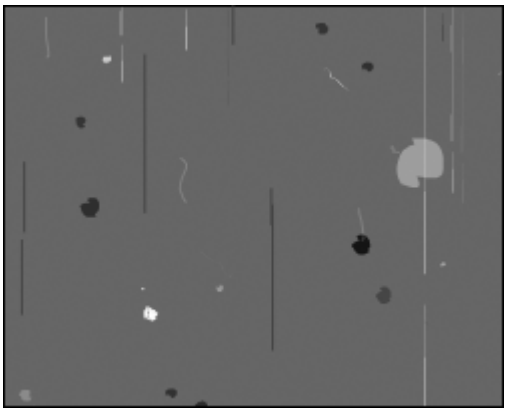


Рис. 27.1. Компонент `DustAndScratches`. Эффект старой кинопленки

Приведем набор основных параметров настройки нашего компонента, которые пользователь сможет задать непосредственно в рабочей среде Flash.

- Количество элементов отдельно для пылинок и царапин. Значения задаются качественными характеристиками: "`Отсутствуют`", "`Мало`", "`Средне`", "`Много`", "`Очень много`".
- Размер элементов отдельно для пылинок и царапин. Значения также задаются качественными характеристиками: "`Мелкие`", "`Средние`", "`Крупные`".
- Максимальная непрозрачность элементов. Значения задаются числом от 0 до 100. Данный параметр определяет максимальное значение свойства `_alpha` для всех элементов. Так, если пользователь задает значение 80, это означает, что свойство `_alpha` для всех пылинок и царапин будет случайным образом изменяться в диапазоне от 0 до 80.
- Частота обновления, определяющая частоту мерцания пылинок и скорость движения царапин. Задается числом, определяющим временной интервал в миллисекундах, через который будет выполняться изменение состояния элементов. Чем меньше скорость этого параметра, тем быстрее мелькание элементов (и, соответственно, нагрузка на ресурсы компьютера).

Кроме описанных параметров настройки наш компонент должен обладать возможностью использования внешних символов библиотеки в качестве пользовательских элементов, представляющих пылинки и царапины. С этой целью компонент необходимо снабдить еще двумя параметрами, позволяющими пользователю задать идентификаторы связи тех символов, которые он хочет использовать вместо стандартного изображения пылинки и царапины. Это открывает для пользователя широкие возможности, позволяющие применить данный компонент для создания разнооб-

разных декоративных эффектов. В качестве пользовательских графических оболочек можно использовать символы, содержащие на своей монтажной линейке несколько кадров. Кадры будут экспонироваться в случайном порядке, поэтому, поместив в разные кадры разные изображения, пользователь сможет добиться большего разнообразия.

Материальную основу нашего компонента будут создавать четыре символа типа MovieClip. В их число входят следующие символы (здесь указаны имена элементов библиотеки, а не экземпляров):

- `DustAndScratches` — исходный символ самого компонента;
- `boundingBox_mc` — символ ограничивающей рамки компонента;
- `dust_mc` — символ с изображением пылинки. На его монтажной линейке имеется несколько кадров с разными изображениями черного и белого цвета;
- `scratch_mc` — символ с изображением царапины. На его монтажной линейке имеется несколько кадров с разными изображениями черного и белого цвета.

Обеспечение функциональности компонента реализуется с помощью трех классов:

- `DustAndScratches` — базовый класс компонента. В нем содержится объявление всех свойств и методов, перечисленных в предыдущем разделе. Этот класс обеспечивает возможности настройки параметров компонента с помощью пользовательского интерфейса, осуществляет присоединение и начальное позиционирование символов `dust_mc` и `scratch_mc`, генерирует событие `onUpdate` с частотой, равной значению заданной пользователем частоты обновления;
- классы `AnimateDust` и `AnimateScratch` — классы, которые связываются с символами `dust_mc` и `scratch_mc`, наделяя их экземпляры заложенной в классе функциональностью. Эти классы реализуют поведение пылинок и царапин, то есть, собственно, отвечают за анимацию.

Кратко опишем общий принцип работы нашего компонента. В рабочей среде пользователь задает значения параметров компонента. При создании экземпляра компонента вызывается метод `setSize()`, масштабирующий компонент в соответствии с заданными пользователем в рабочей среде размерами. Затем вызывается метод `createChildren()`, который динамически связывает символы `dust_mc` и `scratch_mc` с классами `AnimateDust` и `AnimateScratch` соответственно и создает заданное количество клипов, представляющих пылинки и царапины. Таким образом, клипы пылинок и царапин уже сразу снабжаются функциональностью, заложенной в связанных с ними классах. Далее вызывается метод `arrange()`, случайным образом задающий начальное положение пылинок и царапин в пределах области, занимаемой экземпляром компонента. Каждый экземпляр компонента генерирует пользовательское событие, которое мы назвали `onUpdate`. Для этих целей применяется глобальная функция `setInterval()`, которая через заданные промежутки времени (соответствующие заданному пользователем значению параметра частоты обновления) инициирует вызов определенного в классе `DustAndScratches` метода `setUpdate()`, транслирующего данное событие всем зарегистрированным на его получение объектам. Таким образом, наш компонент постоянно генерирует событие подобно тому, как это происходит с событием `onEnterFrame`. Различие заключается в том, что частота генерирования события `onUpdate` определяется не скоростью

воспроизведения монтажной линейки, а заданным пользователем значением временного интервала. Для того чтобы реализовать анимацию клипов пылинок и царапин, достаточно зарегистрировать их в качестве приемников события `onUpdate` и в связанных с ними классах реализовать метод, обрабатывающий данное событие. Регистрация экземпляров символов `dust_mc` и `scratch_mc` в качестве приемников события `onUpdate`, генерируемого компонентом, осуществляется в методе `createChildren()`, то есть непосредственно после их создания.

Принципиальная схема работы компонента `DustAndScratches` представлена на рис. 27.2.

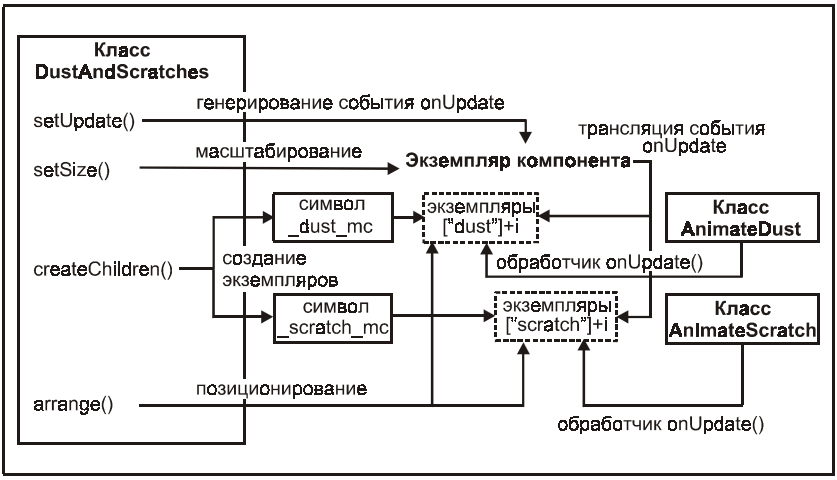


Рис. 27.2. Принципиальная схема работы компонента `DustAndScratches`

Создание клипа компонента

Создание компонента начинается с создания в библиотеке обычного символа типа `MovieClip`. Этот символ всегда содержит два кадра, назначение которых описывается далее. Клип для нашего будущего компонента мы назвали `DustAndScratches`. Исходный клип компонента `DustAndScratches` читатель сможет найти в библиотеке файла DustAndScratches.fla на сопровождающем компакт-диске.

1. В первый кадр помещается клип с прямоугольной ограничивающей рамкой, размер которой соответствует размеру экземпляра компонента, используемому по умолчанию. Левый верхний угол ограничивающей рамки должен совпадать с точкой регистрации клипа компонента. Несмотря на то, что ограничивающая рамка не оказывает влияния на функциональность компонента, ее назначение очень важно. Дело в том, что в первом кадре компонента не должно находиться никакой другой графики, поэтому ограничивающая рамка придает компоненту "материальные" очертания и размер. Когда пользователь вытаскивает из библиотеки (или из панели **Components**) экземпляр компонента к себе на сцену, он

принимает размеры, соответствующие размерам ограничивающей рамки. Если бы ограничивающей рамки не было, эффект был бы такой же, как и при создании на сцене экземпляра клипа, у которого в первом кадре отсутствует графическое содержимое. В этом случае пользователь был бы лишен возможности видеть экземпляр компонента и изменять его размеры с помощью инструмента **Free Transform**. Если у пользователя включен режим предварительного просмотра компонентов (**Control>Enable Live Preview**), он будет видеть реальный облик компонента. При выключенном режиме пользователь будет видеть только ограничивающую рамку компонента.

Поскольку ограничивающая рамка компонента не должна быть видна во время его работы (и в режиме **Live Preview**) в коде метода `init()` (который вызывается из конструктора класса компонента) размещаются инструкции, которые скрывают эту рамку и уменьшают ее размер до нуля.

2. Во второй кадр помещаются экземпляры всех символов, использующихся для визуализации компонента. Так, в нашем случае во второй кадр нужно будет поместить экземпляры клипов `dust_mc` и `scratch_mc`. Графическая оболочка компонентов, как правило, всегда формируется программным образом. Составляющие ее элементы (клипы) динамически присоединяются из библиотеки с помощью кода в управляющем классе компонента. Какие-то элементы (например, рамки) могут создаваться средствами динамического рисования (что активно используется во встроенных компонентах). Для чего же тогда нужно размещать экземпляры элементов библиотеки на монтажной линейке? Главной причиной этого служит необходимость включения всех графических элементов, используемых компонентом, в состав компилированного клипа, в который на конечном этапе будет преобразован исходный клип компонента. Если какие-то элементы библиотеки не будут непосредственно помещены на монтажную линейку компонента, Flash не станет включать их в состав компилированного клипа.

   Поскольку содержимое второго кадра непосредственно не используется компонентом, в первом кадре необходимо поместить вызов функции `stop()`. Это, с одной стороны, предотвратит переход воспроизводящей головки с первого кадра на второй (что совершенно нежелательно). С другой стороны, не будет требовать дополнительных ресурсов на преждевременную инициализацию составляющих элементов компонента.

Клипу с изображением ограничивающей рамки, который был помещен в первый кадр, мы присвоили имя `_boundingBox`. Это необходимо сделать, поскольку нам потребуется обратиться к данному клипу из кода класса, чтобы скрыть его.

Далее необходимо создать символы `dust_mc` и `scratch_mc` с изображением пылинки и царапины (экземпляры которых необходимо поместить во второй кадр монтажной линейки клипа компонента). Данные символы будут содержать несколько кадров, в каждый из которых будет помещено свое изображение. Это позволит добиться большего разнообразия, поскольку кадры будут экспонироваться в случайном порядке. Символам `dust_mc` и `scratch_mc` мы присвоили идентификаторы связи `_dust_mc` и `_scratch_mc` соответственно. Это также необходимо сделать, поскольку их экземпляры будут создаваться программным образом с помощью метода `attachMovie()`. В принципе, эти символы можно было бы сразу связать с управляющими ими классами `AnimateDust` и `AnimateScratch` с помощью диалогового

окна **Linkage Properties** (как в основном и поступают на практике), однако мы не станем этого делать, так как связывание будет осуществляться динамически с помощью метода `Object.registerClass()`. Причиной этого является то, что наш компонент предполагает возможность присоединения вместо стандартных элементов `dust_mc` и `scratch_mc` пользовательских графических оболочек. Очевидно, для этого необходимо связывать классы `AnimateDust` и `AnimateScratch` с символами, идентификаторы связи которых задаются пользователем. Поэтому в нашей ситуации обойтись без метода `Object.registerClass()` не удастся.

На этом работа с монтажной линейкой и векторным инструментарием заканчивается. Теперь необходимо присвоить клипу компонента идентификатор связи и связать его с управляющим классом. Для этого нужно воспользоваться командой библиотеки **Linkage...** и в диалоговом окне **Linkage Properties** указать соответствующие значения. Идентификатор связи символа компонента, как правило, задается в соответствии с его именем (в нашем случае `DustAndScratches`). В поле **AS 2.0. Class** необходимо ввести полный путь к классу. Следует иметь в виду, что изменить значение идентификатора связи у готового компонента пользователь не сможет.

И в заключение необходимо в библиотеке, щелкнув на клипе компонента правой кнопкой мыши, из контекстного меню выбрать команду **Component Definition...**. В появившемся вслед за этим одноименном диалоговом окне нужно в поле **AS 2.0. Class** задать полный путь к управляющему компонентом классу. В результате этого действия пиктограмма символа компонента в библиотеке примет вид стандартной иконки компонента , используемой по умолчанию.

Примечание

На стадии создания компонента возникает необходимость в тестировании тех или иных аспектов его функциональности. Для этого может потребоваться превратить исходный клип компонента в компилированный клип. Чтобы выполнить данную процедуру, необходимо правой кнопкой мыши щелкнуть на символе компонента в библиотеке и из контекстного меню выбрать команду **Convert to Compiled Clip**. Получившийся компилированный клип будет обладать всей функциональностью компонента, реализованной к моменту выполнения компиляции. Следует иметь в виду, что для тестирования изменений, внесенных в код управляющего компонентом класса, необходимо после каждого изменения перекомпилировать исходный клип компонента.

Объявление класса компонента

Объявление управляющего класса компонента выполняется во внешнем AS-файле с применением стандартного синтаксиса ActionScript 2.0. В зависимости от организации структуры кода управляющий компонентом класс может являться подклассом другого класса, реализующего более общую функциональность.

Класс компонента, имеющий графическое представление, всегда является подклассом класса `MovieClip`. Это дает ему возможность использовать базовые программные инструменты для работы с графическими объектами (такие как, например, свойства `_x`, `_y`, `_width`, `_height` и т. д.). В основе архитектуры встроенных компонентов пользовательского интерфейса Flash лежат два базовых класса `UIObject` и

UIComponent. Класс UIObject является подклассом класса MovieClip. Класс UIComponent является подклассом класса UIObject. Управляющие встроенными компонентами классы наследуют от классов UIObject и UIComponent (и, таким образом, от класса MovieClip). При создании собственного компонента с применением архитектуры встроенных компонентов необходимо сделать управляющий им класс подклассом UIComponent (или UIObject).

Поскольку мы не будем использовать встроенные классы компонентов, наш класс DustAndScratches будет наследовать непосредственно от класса MovieClip.

Перед объявлением класса необходимо с помощью ключевого слова import импортировать все используемые в нем внешние классы. Для обеспечения возможности генерирования и трансляции события нам потребуется импортировать встроенный класс EventDispatcher, входящий в состав пакета mx.events. В файловой системе он находится примерно по такому адресу: **E:\Program files\Macromedia\Flash MX 2004\en\First Run\Classes\mx\events\EventDispatcher.as**. Поскольку в коде класса нам придется динамически связать экземпляры символов пылинок и царапин с классами AnimateDust и AnimateScratch, их также нужно будет импортировать. Кроме того, перед объявлением класса размещаются теги метаданных, указывающие среде Flash, какие события генерирует компонент и в каком файле находится изображение иконки компонента (о тегах метаданных подробнее будет сказано далее).

Объявление нашего класса будет иметь следующий вид (исходный программный код класса DustAndScratches можно найти в файле DustAndScratches.as, расположенном в папке Глава 27 на сопровождающем книгу компакт-диске):

```
import mx.events.EventDispatcher;
import AnimateDust
import AnimateScratch
 [Event("onUpdate")] // Событие, генерируемое компонентом
 [IconFile("DustAndScratches.png")] // Файл с пиктограммой компонента
class DustAndScratches extends MovieClip {
...
}
```

Объявление свойств

Как правило, свойства управляющего класса компонента, которые объявляются с помощью ключевого слова var, имеет смысл скрыть от внешнего доступа с помощью атрибута private. Эти свойства будут использоваться во внутренней реализации компонента. Для обеспечения возможности доступа к свойствам компонента извне (со стороны пользователя) следует использовать get/set-методы. Такой подход лучше согласуется с концепцией инкапсуляции данных, лежащей в основе объектно-ориентированного программирования. Исключение могут составлять статические свойства, предоставляющие информацию, связанную со всеми экземплярами компонентов (например, статическое свойство, хранящее номер версии компонента).

В этом разделе приведен фрагмент кода, содержащий объявление скрытых свойств, используемых для обеспечения функциональности компонента. Для обеспечения возможности обращения к этим свойствам извне для настройки параметров компонента мы будем использовать `get/set`-методы. Тем не менее некоторым свойствам мы сразу присвоим значения, используемые по умолчанию. Это нужно сделать для того, чтобы при динамическом создании экземпляра компонента (с помощью `attachMovie()`) его параметры сразу были установлены в соответствующие значения.

Свойства `__width` и `__height` используются для внутреннего хранения размеров экземпляров компонента. Два символа нижнего подчеркивания в начале имени используются для того, чтобы отличить эти свойства от встроенных свойств `_width` и `_height`.

```
private var __width:Number;
private var __height:Number;
```

Поскольку во время инициализации компонента нам потребуется спрятать его ограничивающую рамку, представленную клипом `_boundingBox`, необходимо объявить его идентификатор в качестве члена класса. В противном случае попытка обратиться к нему из кода метода вызовет ошибку.

```
private var _boundingBox:MovieClip;
```

Как уже было сказано, экземпляры клипов, представляющих пылинки и царапины, будут присоединяться динамически. Для этих целей будет создан пустой клип с именем `holder`, который будет исполнять роль контейнера. Для того чтобы к нему можно было обратиться по имени из определения класса, необходимо объявить соответствующее свойство:

```
private var holder:MovieClip;
```

Для того чтобы пылинки и царапины не выходили за границы компонента, мы будем применять динамическое маскирование. Для этих целей будет создан клип, используемый в качестве маски:

```
private var mask:MovieClip;
```

Свойство `__alpha` (два символа нижнего подчеркивания) предназначено для внутреннего хранения значения максимальной непрозрачности пылинок и царапин. Для считывания и установки значения данного свойства будут использоваться `get/set`-методы. Параметр, соответствующий этому свойству, будет выведен на панель **Properties**. По умолчанию будет использоваться значение 100.

```
private var __alpha:Number = 100;
```

Поскольку наш компонент предполагает возможность применения пользовательских графических оболочек, необходимо объявить свойства для хранения идентификаторов связи пользовательских клипов, представляющих пылинки и царапины. По умолчанию в качестве значений этих свойств мы будем использовать идентификаторы связи символов `dust_mc` и `scratch_mc`. Для считывания и установки значений данных свойств будут использоваться `get/set`-методы. Параметры, соответствующие этим свойствам, будут выведены на панель **Properties**.

```
private var _customDustSkin:String = "_dust_mc";
private var _customScratchSkin:String = "_scratch_mc";
```

Пользователь должен иметь возможность задать размер и количество элементов. Для большего удобства имеет смысл реализовать установку этих значений, используя не количественные показатели (числа), а качественные (слова). Следующие свойства предназначены для хранения массивов, содержащие допустимые качественные определения характеристик размера и количества элементов. Поскольку для всех экземпляров компонента будет использоваться одинаковый набор значений параметров, эти свойства следует сделать статическими. Обязанностью нашего класса будет перевод этих значений в числовую форму:

```
private static var _magnitudeValues:Array = ["Мелкие", "Средние",↳
"Крупные"];
private static var _quantityValues:Array = ["Отсутствуют", "Мало",↳
"Средне", "Много", "Очень много"];
```

Следующие свойства хранят массивы с количественными значениями размеров и количества элементов, соответствующими указанным качественным. Размеры задаются не абсолютной величиной, а коэффициентами, которые можно рассматривать, как доли от габаритов экземпляра компонента. Поскольку в общем случае пылинки и царапины должны иметь разные размеры (и пропорции), эти значения будут использованы с поправочными коэффициентами, которые вводятся в коде классов, обеспечивающих анимацию самих объектов. Количественные значения числа элементов, соответствующие указанным качественным, говорят о том, сколько экземпляров клипов, представляющих пылинки и царапины, будет создано.

```
private static var _magnitudeFigures:Array = [.07, 0.13, .2];
private static var _quantityFigures:Array = [0, 3, 5, 10, 20];
```

Следующие скрытые свойства предназначены для хранения текущих значений количества и размеров клипов, представляющих пылинки и царапины. Для считывания и установки значений этих свойств будут использоваться `get/set`-методы. Параметры, соответствующие этим свойствам, будут выведены на панель **Properties**.

```
// Текущее количество клипов
private var _dustNum:Number;
private var _scratchNum:Number;
// Текущие коэффициенты размеров клипов
private var _dustSize:Number;
private var _scratchSize:Number;
```

Следующее свойство предназначено для хранения частоты обновления, то есть продолжительность временного интервала (в миллисекундах), с которым наш компонент будет генерировать событие `onUpdate`. По умолчанию используется значение 50 мс. Для считывания и установки значений данного свойства будут использоваться `get/set`-методы. Параметр, соответствующий этому свойству, будет выведен на панель **Properties**.

```
private var _frequency:Number = 50;
```

Наш компонент будет предусматривать возможность отключения и включения работы эффекта. Следующее свойство будет хранить булеву величину, определяющую, включен или выключен эффект.

По умолчанию эффект включен.

```
private var _active:Boolean = true;
```

Для реализации возможности генерирования и трансляции событий нам необходимо определить три свойства, соответствующие методам класса `EventDispatcher`. Это необходимо сделать для того, чтобы использовать данные методы в коде класса и вызывать их в конечном фильме для экземпляра компонента. Реализация данных методов содержится в классе `EventDispatcher` (применение класса `EventDispatcher` будет подробнее обсуждаться далее).

```
private var dispatchEvent:Function;
public var addEventListener:Function;
public var removeEventListener:Function;
```

Применение тегов метаданных

Специфической особенностью класса, управляющего компонентом, является наличие в нем тегов метаданных. Теги метаданных (metadata tags) — это текстовые элементы, которые размещаются разработчиком непосредственно в коде класса компонента (хотя не являются инструкциями ActionScript) и используются средой Macromedia Flash для получения разнообразных сведений о компоненте. На основании этих сведений Flash выводит параметры компонента в панелях **Properties** и **Component Inspector** (см. рис. 27.4), а также выполняет ряд других операций, связанных с интеграцией компонента в рабочую среду Flash.

В зависимости от назначения теги метаданных могут быть связаны с отдельным элементом (свойством) или с классом целиком. В первом случае тег должен быть помещен внутри определения класса непосредственно перед соответствующим элементом. Во втором случае тег помещается перед объявлением класса. Тег метаданных заключается в квадратные скобки и в общем случае имеет следующий синтаксис:

```
[Имя тега (атрибут1, атрибут2, ..., атрибутN)]
```

Теги метаданных играют очень важную роль в обеспечении возможности настройки параметров компонента с помощью панелей **Properties** или **Property Inspector**.

Тег *Inspectable*. Описание параметров внешнего интерфейса компонента

Для того чтобы сделать объявленное в классе компонента свойство доступным для настройки в рабочей среде, необходимо непосредственно перед его объявлением поместить тег `Inspectable`, снабдив его набором требуемых атрибутов. Свойство, перед объявлением которого помещен данный тег, автоматически становится параметром компонента, доступ к которому пользователь может получить с помощью панелей **Properties** или **Property Inspector**, выделив экземпляр компонента на сцене. С помощью тега `Inspectable` разработчик может определить, каким образом пользователь сможет задать значение тому или иному параметру. В зависимости от типа значения параметра и поставленных задач значение может вводиться вручную, вы-

бираться из списка заданных значений, задаваться в виде массива и даже выбираться из каталога цветов (при установке значения цвета).

Тег `Inspectable` может быть размещен непосредственно перед объявлением свойства или перед объявлением `get/set`-методов, реализующих возможности доступа к этому свойству. Второй подход является более предпочтительным (и удобным) по следующим причинам.

- Зачастую изменение значения свойства, осуществляемое пользователем извне, должно сопровождаться определенными действиями. Например, если пользователь меняет количество элементов, необходимо вызвать метод `createChildren()`, заново создающий элементы. Если тег `Inspectable` помещен перед объявлением свойства, то изменение значения этого свойства не приводит к автоматическому выполнению требуемых действий. Если же разместить тег перед объявлением `get/set`-методов, то при изменении значения свойства будет вызываться соответствующий метод, в который можно поместить все необходимые инструкции.
- Непосредственное обращение к свойствам класса нарушает инкапсуляцию. Поместив тег `Inspectable` перед объявлением свойства, мы сделали бы это свойство доступным для изменения извне.

Следует иметь в виду, что тег `Inspectable` должен находиться в строке, непосредственно предшествующей строке, содержащей объявление свойства или `get/set`-методов.

Тег `Inspectable` имеет следующий синтаксис:

```
[Inspectable(атрибут1=значение, атрибут2=значение, ...,↳
атрибутN=значение)]
```

Атрибуты тега `Inspectable` являются необязательными. Поэтому их количество определяется только конкретными задачами. Порядок следования атрибутов не оказывает влияния на результат. Порядок следования параметров в панелях **Properties** и **Property Inspector** определяется на основании кодовой позиции первого символа их названий. Приведем перечень атрибутов тега `Inspectable` с описанием их назначения:

- `defaultValue` — задает значение свойства, используемое по умолчанию. Как правило, этот атрибут должен быть задан. Присваиваемое данному атрибуту значение необходимо оформить в соответствии с типом представляемого им элемента данных (число, строка и т. д.). Соответствующие правила будут приведены в пункте с описанием атрибута `type`;
- `enumeration` — этот атрибут применяется в случае, когда требуется, чтобы пользователь не вводил значение вручную, а выбирал его из выпадающего списка, который автоматически появляется при щелчке по соответствующей строке в панели **Properties** или **Property Inspector**. Значение данного атрибута заключается в кавычки и представляет собой набор допустимых значений, перечисленных через запятую, например:

  ```
  [Inspectable(defaultValue="Средние",enumeration= "Мелкие, Средние,
  Крупные")]
  ```

- `name` — данный атрибут принимает строковое значение и позволяет присвоить параметру имя, которое будет отображаться в панелях **Properties** или **Property**

Inspector, вместо реального имени свойства. Это позволяет сделать интерфейс компонента более понятным для пользователя. Так, например, вместо вывода малоинформативного названия `dustNum` можно назвать параметр `"Количество пылинок"`;

- `type` — этот атрибут позволяет указать тип параметра и, таким образом, влияет на форму, в которой он будет задаваться пользователем. Значение данного атрибута задается строкой. Типы параметра компонента не полностью совпадают с типами данных ActionScript. Это связано с тем, что задаваемый тип определяет способ задания значения параметра компонента пользователем, а не тип представляемых им данных. Приведем перечень допустимых типов.
  - `Array` — массив, позволяет задавать значение параметра в виде массива, состоящего из строковых и/или числовых значений. Значение параметра этого типа выводится в строке панели **Properties** или **Property Inspector** в виде заключенного в квадратные скобки набора значений, перечисленных через запятую. В правой части строки имеется кнопка с пиктограммой линзы . Для задания значений элементам массива (или для изменения порядка их следования) нужно дважды щелкнуть по соответствующей строке или по пиктограмме с изображением линзы. В результате этого действия появится диалоговое окно **Values** (рис. 27.3). Кнопки с пиктограммами (+) и (−) служат, соответственно, для добавления нового элемента (его значение нужно будет ввести в новое поле) и удаления элемента из выделенной строки. С помощью стрелок можно перемещать выделенный элемент в начало или в конец массива. Если задаваемое значение элемента соответствует числовому литералу, элемент массива будет иметь числовой тип, в противном случае элемент конвертируется в строку. Свойство, соответствующее параметру типа `Array`, должно иметь тип `Array`. Значение по умолчанию задается строкой, содержащей набор элементов, перечисленных через запятую. Если элементом является строка, наличие внутренних кавычек не требуется.

```
[Inspectable(type="Array", defaultValue="Привет, Пока, Hello, Bye")]
```

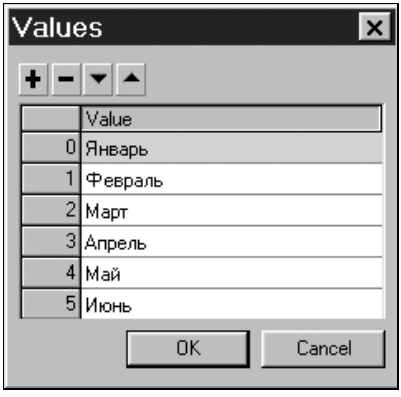


Рис. 27.3. Диалоговое окно **Values**

- `Boolean` — булев тип. Значение параметра этого типа задается с помощью выпадающего списка, содержащего два значения `true` и `false`. Свойство, соответствующее параметру типа `Boolean`, должно иметь булев тип. Значение по умолчанию для этого типа задается в виде строки, содержащей значение `true` или `false`.

  ```
  [Inspectable(type="Boolean", defaultValue="true")]
  ```

- `Color` — цвет, позволяет вручную устанавливать значение цвета, соответствующее параметру. Значение параметра этого типа выводится в строке панели **Properties** или **Property Inspector** в виде шестнадцатеричного числа с префиксом `#`, рядом с которым находится цветовой образец, соответствующий текущему значению цвета. Для задания требуемого цветового значения необходимо щелкнуть по цветовому образцу (или дважды щелкнуть по строке данного параметра), в результате чего будет выведен стандартный каталог цветов с текущей палитрой, а курсор приобретет вид инструмента **Eyedropper**. Свойство, соответствующее параметру типа `Color`, должно иметь числовой тип. Значение по умолчанию задается строкой, содержащей шестнадцатеричное представление цвета с префиксом `#`.

  ```
  [Inspectable(type="Color", defaultValue="#FFFFFF")]
  ```

- `Font Name` — шрифт, позволяет выбирать шрифт из списка установленных шрифтов. Этот тип применяется в случаях, когда параметр представляет собой название шрифта, используемого компонентом. Для указания требуемого шрифта пользователю достаточно щелкнуть по строке параметра и выбрать его из стандартного перечня шрифтов, установленных в системе. Свойство, соответствующее параметру типа `Font Name`, должно иметь строковый тип. Значение по умолчанию задается строкой, содержащей название шрифта:

  ```
  [Inspectable(type="Font Name", defaultValue="Times New Roman")]
  ```

- `List` — список. Этот параметр использовался в предыдущей версии Flash для задания параметров компонента в виде списка. При работе в Flash MX 2004 и применении ActionScript 2.0 вместо использования данного типа следует применять атрибут `enumeration`. Поддержка типа `List` сохраняется исключительно для обеспечения совместимости с компонентами, созданными в Flash MX.

- `Number` — число, используется для задания числовых значений. Для задания этого параметра пользователь должен вручную ввести требуемое число в соответствующее поле. Если заданное пользователем значение не является числом, оно автоматически заменяется значением, заданным по умолчанию с помощью атрибута `defaultValue`. Свойство, соответствующее параметру типа `Number`, должно иметь числовой тип. Значение по умолчанию задается числом:

  ```
  [Inspectable(type = "Number", defaultValue=100)]
  ```

- `Object` — объект, позволяет задать значение параметра в виде объекта с заданным набором свойств, которые интерпретируются как числа или строки. Значение параметра этого типа выводится в строке панели **Properties** или **Property Inspector** в виде заключенного в фигурные скобки набора пар *имя*:*значение*, перечисленных через запятую (похоже на литерал объекта). Значения параметра данного типа задаются с помощью диалогового окна **Values**,

аналогично тому, как это делается при использовании типа `Array`. При использовании типа `Object` пользователь не может изменять существующие имена свойств, а также добавлять или удалять свойства объекта. Допускается только изменение значений свойств. Если задаваемое значение свойства соответствует числовому литералу, свойство объекта будет иметь числовой тип, в противном случае свойство конвертируется в строку. Свойство класса компонента, соответствующее параметру типа `Object`, должно иметь тип `Object`. Значение по умолчанию задается строкой, содержащей набор пар *имя:значение*, разделенных запятыми.

```
[Inspectable(type = "Object", defaultValue="num:3.14, str:Flash")]
```

- `String` — строка, позволяет задать значение параметра в виде строки. Для задания этого параметра пользователь должен вручную ввести требуемое значение в соответствующее поле панели **Properties** или **Property Inspector**. Свойство, соответствующее параметру типа `String`, должно иметь тип `String`. Значение по умолчанию задается строкой:

```
[Inspectable(type = "String", defaultValue="default")]
```

- `verbose` — этот атрибут позволяет определить, будет ли параметр выводиться в панели **Properties**. По умолчанию все свойства, помеченные тегами `Inspectable`, выводятся и в панели **Properties**, и в панели **Component Inspector**. Поскольку панель **Properties** имеет небольшую высоту, одновременно в ней могут быть видны только 6 параметров. При большем числе параметров появляется вертикальная полоса прокрутки, что несколько затрудняет настройку компонента. Как правило, если компонент имеет много параметров, в панель **Properties** выносятся только основные. Для настройки всех параметров используется панель **Component Inspector**. Чтобы запретить вывод параметра в панели **Properties**, необходимо присвоить атрибуту `verbose` значение 1. Задать значение такому параметру можно будет только с помощью панели **Component Inspector**.

```
[Inspectable(name="Оболочка", defaultValue="default", verbose=1)]
```

Для нашего компонента параметры, позволяющие пользователю задать идентификаторы связи для присоединения пользовательских графических оболочек, будут отображаться только в панели **Component Inspector**.

- `category` — этот атрибут позволяет объединить параметры, выводимые в панелях **Properties** и **Component Inspector**, в тематические категории. По умолчанию все параметры выводятся в алфавитном порядке. Для того чтобы близкие по назначению параметры следовали друг за другом (вне зависимости от названий), их нужно объединить в тематические группы или категории. Для того чтобы создать такую группу, необходимо задать одно и то же строковое значение атрибутам `category` для каждого свойства, соответствующего параметру, относящемуся к определенной категории. Значение атрибута `category` задается в виде произвольной строки, которая отражает общее назначение параметров, относящихся к одной категории. Внутри каждой тематической группы параметры располагаются в алфавитном порядке. Параметры, не относящиеся ни к одной из категорий (то есть не снабженные атрибутом `category`), всегда предшествуют сгруппированным параметрам. При наличии нескольких разных тематических групп порядок

следования входящих в их состав параметров определяется на основании значений атрибута `category` в алфавитном порядке. Следующий тег помещает параметр `Макс. непрозрачность` в категорию `other`.

```
[Inspectable(name = "Макс. непрозрачность", type = "Number",
defaultValue = 100, category = "other")]
```

На рис. 27.4 представлен внешний вид панели **Component Inspector** с выведенными на нее параметрами нашего компонента `DustAndScratches`.

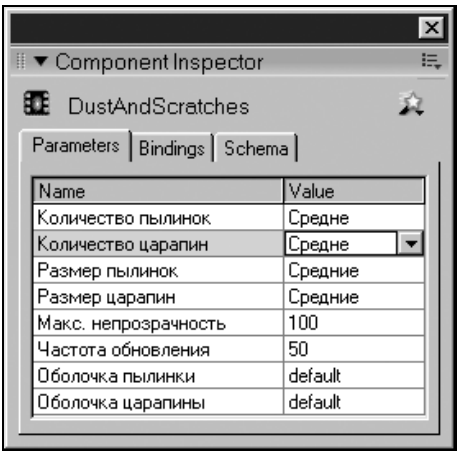


Рис. 27.4. Параметры компонента `DustAndScratches`

Примечание

Замечено, что иногда применение некоторых атрибутов тега `Inspectable` (например, `verbose` или `category`) не оказывает должного эффекта. Причиной этого является то, что Flash некорректно интерпретирует данные параметры и, соответственно, некорректно записывает их в XML-документ catalog.xml, автоматически создаваемый при экспортировании SWC-архива компонента. Для решения этой проблемы можно вручную отредактировать файл catalog.xml, являющийся частью SWC-архива.

Тег *InspectableList*. Список внешних параметров

Если управляющий компонентом класс является подклассом класса, реализующего более общую функциональность, то по умолчанию в панелях **Properties** и **Component Inspector** будут выводиться параметры, соответствующие свойствам, снабженным тегом `Inspectable`, не только непосредственно управляющего класса, но и его суперкласса.

В случаях, когда создание параметров на основе свойств суперкласса компонента нежелательно, можно воспользоваться тегом `InspectableList`. Данный тег должен обязательно размещаться непосредственно перед объявлением класса, так как он

применяется не к конкретному свойству, а ко всему классу целиком. Назначение этого тега состоит в том, что он указывает, какие именно свойства, снабженные тегом `Inspectable`, будут использоваться в качестве параметров компонента. Это относится как к свойствам, объявленным в самом классе, так и к наследуемым свойствам. Таким образом, тег `InspectableList` фактически создает список свойств, на основании которых будут созданы параметры компонента. Остальные свойства при этом не буду выводиться в качестве параметров в панелях **Properties** и **Component Inspector**, даже несмотря на то, что они снабжены тегом `Inspectable`.

Тег `InspectableList` имеет следующий синтаксис:

```
[InspectableList("свойство1", "свойство2", ..., " свойствоN")]
```

В следующем примере параметрами компонента, которые будут отображаться в панелях **Properties** и **Component Inspector**, будут выступать только свойства `width`, `height` и `area`, вне зависимости от того, объявлены они в классе `Rectangle` или `Shape` (предполагается, что эти свойства снабжены тегом `Inspectable`).

```
[InspectableList("width", "height", "area")]
class Rectangle extends Shape {
...
}
```

Тег *Event*. Описание событий компонента

Если компонент обладает способностью генерировать и транслировать события, информацию об этом можно включить в файл управляющего класса с помощью тега метаданных, который называется `Event`. Этот тег необходимо поместить перед объявлением класса, поскольку он предоставляет информацию, связанную со всем классом. Тег `Event` имеет следующий синтаксис:

```
[Event("событие")]
```

Если компонент генерирует несколько разных событий, то для каждого из них необходимо создать собственный тег.

В принципе, наличие или отсутствие тега `Event` никак не сказывается на работоспособности компонента. Однако, если данный тег присутствует в управляющем классе, в рабочей среде при прикреплении сценария непосредственно к экземпляру компонента с использованием обработчика `on()` будет выводиться всплывающая подсказка, содержащая перечень всех событий, указанных в классе с помощью тега `Event`.

Наш компонент будет генерировать только одно событие `onUpdate`, в связи с чем перед объявлением класса мы поместили соответствующий тег `Event`.

Тег *IconFile*. Создание пиктограммы компонента

В панели **Components** (**Window>Development Panels>Components**) и в библиотеке Flash-документа рядом с названием документа отображается соответствующая ему пиктограмма. Всем пользовательским компонентам по умолчанию соответствует стандартная пиктограмма . Тем не менее у каждого из встроенных компонентов

имеется своя индивидуальная пиктограмма. При желании можно создать пользовательскую пиктограмму и для собственного компонента. Чтобы это сделать, необходимо выполнить следующие действия.

1. В любом растровом редакторе (например, Macromedia Fireworks или Adobe Photoshop) создать изображение и сохранить его в формате PNG. Это можно также сделать прямо в Flash, а затем экспортировать изображение в формат PNG с помощью команды **File>Export>Export Image...**. Изображение должно отвечать следующим требованиям:
   - иметь размер 18×18 пикселов;
   - иметь глубину цвета 8 бит.
2. Сохранить изображение в том же самом каталоге, где сохранен исходный FLA-файл компонента.
3. Перед объявлением класса компонента поместить тег `IconFile`, указывающий имя файла, содержащего пиктограмму компонента.

Тег всегда помещается перед объявлением класса. При наличии этого тега в управляющем классе компонента соответствующее изображение будет использоваться в качестве иконки компонента вместо стандартной иконки. Тег `IconFile` имеет следующий синтаксис:

```
[IconFile("имя файла.png")]
```

Объявление *get/set*-методов

Для реализации внешнего программного интерфейса компонента очень удобно использовать `get/set`-методы. Эти методы должны предоставлять пользователю возможность считывания и установки значений свойств, используемых для настройки работы компонента. Причем вызов данных методов для пользователя выглядит как простое считывание свойства или присваивание свойству значения, в то время как при этом могут выполняться еще и дополнительные действия (например, приведение заданного значения к допустимому значению). Часть `get/set`-методов (или все) можно задействовать в качестве параметров компонента, которые будут выведены в панели **Properties** и/или **Component Inspector**. Для этого перед объявлением соответствующих методов необходимо поместить тег `Inspectable`, снабдив его необходимыми атрибутами. Когда пользователь в рабочей среде задает значение параметру, созданному на основе `get/set`-метода, Flash автоматически осуществляет вызов соответствующего `set`-метода. Для вывода значения параметра, используемого по умолчанию, выполняется вызов `get`-метода. Таким образом, если экземпляр компонента был помещен на сцену в рабочей среде вручную, то пользователь, задавая значения его параметров, выполняет инициализацию свойств данного экземпляра. После публикации документа свойства, соответствующие параметрам, выведенным в панели **Properties** и **Component Inspector**, принимают установленные в рабочей среде значения. Все открытые `get/set`-методы, используемые в качестве параметров компонента, также являются доступными и в процессе воспроизведения фильма. Их можно использовать для программного управления экземпляром компонента.

Следует иметь в виду, что если экземпляр компонента создается не вручную, а динамически (например, с помощью метода `attachMovie()`), то его создание, естественно, не сопровождается автоматическим вызовом `get/set`-методов. Автоматически при этом вызывается только конструктор управляющего компонентом класса. Поэтому, если компонент предполагает возможность динамического присоединения, разработчик должен позаботиться о том, чтобы во время создания экземпляра произошла инициализация его свойств. Это можно обеспечить различными способами:

- инициализировать собственные свойства экземпляра прямо в конструкторе. Для того чтобы это происходило только при динамическом создании экземпляра, необходимо осуществлять проверку того, что свойства еще не определены (то есть имеют значение `undefined`). В этом случае свойству можно присвоить значение по умолчанию. Если же свойство уже определено (то есть экземпляр был создан вручную, и его параметры были заданы в рабочей среде), то изменять значение свойств нельзя;
- при объявлении наследуемых свойств класса сразу присваивать им значения, используемые по умолчанию. В этом случае, если для экземпляра не определено собственное свойство, он будет использовать свойство, наследуемое из класса;
- можно возложить обязанность по инициализации свойств экземпляра на пользователя, который должен будет при вызове метода `attachMovie()` передавать ему в качестве параметра объект, содержащий полный набор свойств со значениями, необходимыми для функционирования компонента. Этот подход является ненадежным, поскольку ставит работу компонента в зависимость от действий пользователя.

В нашем случае некоторым свойствам значение присваивается непосредственно при их объявлении. Другие инициализируются в конструкторе, поскольку присвоить им значения при объявлении нельзя, так как эти значения не являются константами на момент компиляции. Тем не менее при динамическом создании экземпляра компонента пользователь при желании сможет передать методу `attachMovie()` инициализирующий объект, для того чтобы переопределить значения требуемых свойств, используемые по умолчанию. Если он этого не сделает, все свойства будут автоматически установлены в значения по умолчанию. Переопределить эти значения также можно будет и после создания экземпляра, используя соответствующие `get/set`-свойства программного интерфейса компонента.

Далее мы приведем программный код, содержащий объявление `get/set`-методов для нашего компонента `DustAndScratches`. Все пояснения будут даваться непосредственно по мере необходимости.

Поскольку значения параметров, отвечающих за количество и размер царапин и пылинок, задаются строковыми значениями, характеризующими качественные понятия ("много" — "мало" или "большие" — "маленькие"), нам потребуется преобразовать их в числовой вид. Как мы помним, качественные значения количества и размеров элементов у нас хранятся в массивах `_quantityValues` и `_magnitudeValues`, а соответствующие им количественные значения — в массивах `_quantityFigures` и `_magnitudeFigures`. Таким образом, после получения от пользователя качественного значения параметра нам потребуется определить его местоположение в массиве

качественных значений и на этом основании найти соответствующий элемент в массиве количественных значений. Для извлечения значения параметра необходимо выполнить обратные действия. Поскольку эти процедуры нам придется проделывать многократно для разных характеристик и разных элементов, целесообразно вынести их в отдельные скрытые методы. Первый метод мы назовем `setValue()` и будем использовать его для установки значения параметра компонента. Второй метод мы назовем `getValue()` и будем использовать его для извлечения значения параметра компонента. Данные методы будут принимать следующие параметры:

- `value` — значение, которое необходимо присвоить параметру (только для метода `setValue()`);
- `prop` — строка, содержащая название внутреннего свойства, соответствующего параметру компонента;
- `sourceFigures` — массив, в котором хранятся количественные значения свойства `prop`, соответствующие качественным значениям параметра компонента;
- `sourceValues` — массив, в котором хранятся качественные значения параметра компонента, соответствующего свойству `prop`.

```
private function setValue(value:String, prop:String, sourceFigures:Array,↳
sourceValues:Array):Void {
        for (var i = 0; i < sourceValues.length; i++) {
                if (value == sourceValues[i]) {
                        this[prop] = sourceFigures[i];
                        break;
                }
        }
}
```

Наш компонент будет снабжен возможностью программного задания произвольного количественного значения размера пылинок и царапин. Это значение, очевидно, может не совпадать с набором значений, содержащихся в массиве `_magnitudeFigures`, и для него нельзя будет найти соответствующее качественное значение в массиве `_magnitudeValues`. Для корректной работы `get`-методов желательно предусмотреть возможность появления такой ситуации. Если текущее значение внутреннего свойства не совпадает ни с одним из предустановленных количественных значений, `get`-метод будет возвращать строку "`Пользовательское значение`" (это может произойти только при программном задании размеров элементов пользователем).

```
private function getValue(prop:String, sourceFigures:Array,
sourceValues:Array):String {
        for (var i = 0; i < sourceFigures.length; i++) {
                if (this[prop] == sourceFigures[i]) {
                        return sourceValues[i];
                }
```

```
        }
        return "Пользовательское значение";
}
```

Теперь можно объявить открытые get/set-методы для установки количества и размеров элементов. На основании этих методов должны быть созданы параметры компонента, поэтому перед объявлением каждой пары методов поместим тег Inspectable. Обратите внимание, что при задании количества элементов выполняется вызов методов createChildren() и arrange(), создающих заданное количество элементов и позиционирующих их в области компонента.

```
[Inspectable(name="Количество пылинок",↳
enumeration="Отсутствуют,Мало,Средне,Много,Очень много",↳
defaultValue="Средне")]
public function set dustNum(value:String):Void {
        setValue(value, "_dustNum", _quantityFigures, _quantityValues);
        createChildren();
        arrange();
}
public function get dustNum():String {
        return getValue("_dustNum", _quantityFigures, _quantityValues);
}
[Inspectable(name="Количество царапин",↳
enumeration="Отсутствуют,Мало,Средне,Много,Очень много",↳
defaultValue="Средне")]
public function set scratchNum(value:String):Void {
        setValue(value, "_scratchNum", _quantityFigures,
        _quantityValues);
        createChildren();
        arrange();
}
public function get scratchNum():String {
        return getValue("_scratchNum", _quantityFigures,
        _quantityValues);
}
[Inspectable(name="Размер пылинок",↳
enumeration="Мелкие,Средние,Крупные", defaultValue="Средние")]
public function set dustSize(value:String):Void {
        setValue(value, "_dustSize", _magnitudeFigures,
        _magnitudeValues);
}
public function get dustSize():String {
```

```
        return getValue("_dustSize", _magnitudeFigures,
        _magnitudeValues);
}
[Inspectable(name="Размер
царапин",enumeration="Мелкие,Средние,Крупные",↳
defaultValue="Средние")]
public function set scratchSize(value:String):Void {
        setValue(value, "_scratchSize", _magnitudeFigures,↳
        _magnitudeValues);
}
public function get scratchSize():String {
        return getValue("_scratchSIze", _magnitudeFigures,↳
        _magnitudeValues);
}
```

Наряду с заданием размеров пылинок и царапин на основании предустановленных качественных значений имеет смысл предусмотреть возможность задания количественного коэффициента размера, на тот случай, если не одно из качественных значений не дает требуемого результата. Поскольку это более специфическая возможность, параметр для нее создаваться не будет. Задать количественное значение размера пользователь сможет только программным образом, используя следующие `get/set`-методы.

```
public function set dustSizeValue(value:Number) {
        _dustSize = value;
}
public function get dustSizeValue():Number {
        return _dustSize;
}
public function set scratchSizeValue(value:Number) {
        _scratchSize = value;
}
public function get scratchSizeValue():Number {
        return _scratchSize;
}
```

Наш компонент не будет предусматривать возможность задания произвольного количества элементов, поскольку предлагаемые варианты позволяют добиться нужного эффекта. Тем не менее при желании читатель может самостоятельно добавить соответствующие `get/set`-методы.

Следующие `get/set`-методы обеспечивают возможность задания максимальной непрозрачности элементов. На их основании будет создан параметр, позволяющий задать данную характеристику с помощью панелей **Properties** или **Component Inspector**. Поскольку значение прозрачности, за которую отвечает встроенное свойство `_alpha`, изменяется в диапазоне от 0 до 100, если заданное пользователем значение

будет выходить за допустимые границы, мы скорректируем его, присвоив граничное значение (правда, это не приведет к автоматическому изменению значения, введенного пользователем в строку параметра).

```
 [Inspectable(name="Макс. непрозрачность", type = "Number",↳
defaultValue=100)]
public function get alpha():Number {
       return __alpha;
}
public function set alpha(value:Number):Void {
       __alpha = Math.max(0, Math.min(value, 100));
}
```

Следующие `get/set`-методы обеспечивают возможность установки/считывания значения частоты обновления в миллисекундах. На их основании будет создан соответствующий параметр компонента. Частота обновления определяет частоту генерирования компонентом события `onUpdate` и, соответственно, скорость мерцания пыли и перемещения царапин. Для того чтобы избежать установки слишком высоких значений частоты, мы устанавливаем нижний предел на значение этого свойства, равным 20 миллисекундам.

```
[Inspectable(name="Частота обновления", type = "Number",↳
defaultValue=50)]
public function set frequency(value:Number) {
       _frequency = Math.max(value, 20);
}
public function get frequency():Number {
       return _frequency;
}
```

Наш компонент предусматривает возможность включения и отключения действия эффекта. В отключенном состоянии анимация пылинок и царапин прекращается, и они становятся не видны. Для того чтобы отключить работу эффекта, достаточно просто удалить из экземпляра компонента клип-контейнер `holder` (при этом все его содержимое также будет удалено) и маскирующий его клип `mask`. Для повторного включения эффекта необходимо снова вызвать методы `createChildren()` и `arrange()`. Отметим, что присваивание значения внутреннему свойству `_active` должно выполняться перед вызовом `createChildren()`, поскольку присоединение клипов с помощью этого метода выполняется только при условии, что свойство `_active` принимает значение `true`. При отключении работы компонента имеет смысл прекратить транслирование события `onUpdate`. Для этого при установке свойства `_active` в значение `false` мы вызываем метод `removeUpdate()`, прекращающий транслирование события, путем удаления интервала (объявление данного метода будет приведено далее). При повторном включении компонента необходимо возобновить транслирование события, для этого мы вызываем метод `setUpdate()`.

```
public function set active(value:Boolean) {
if (value) { // Включение
```

```
        _active = true;
        setUpdate();
        createChildren(); // Присоединяем клипы
        arrange();// Позиционируем клипы
        }
else {// Отключение
        _active = false;
        removeUpdate();
        holder.removeMovieClip();// Удаляем контейнер с клипами
        mask.removeMovieClip();// Удаляем его маску
        }
}
public function get active():Boolean {
        return _active;
}
```

Теперь создадим открытые свойства, позволяющие программным образом задавать размер компонента.

```
public function set width(value:Number):Void {
        setSize(value, __height);
}
public function get width():Number {
        return __width;
}
public function set height(value:Number):Void {
        setSize(__width, value);
}
public function get height():Number {
        return __height;
}
```

Последними элементами программного интерфейса нашего компонента будут `get/set`-свойства, позволяющие пользователю задать идентификаторы связи символов, которые он хочет использовать в качестве графических оболочек пылинки и царапины. На основании этих свойств будут созданы параметры, позволяющие задать требуемое значение с помощью панели **Component Inspector** (для того чтобы данные параметры выводились только в панели **Component Inspector**, мы присвоим атрибуту `verbose` тега `Inspectable` значение 1). Значение параметра `"default"` соответствует стандартной оболочке и выводится по умолчанию. Для того чтобы применить пользовательскую графическую оболочку для элемента, достаточно в качестве значения соответствующего параметра в панели **Component Inspector** ввести строку, содержащую идентификатор связи клипа. Если пользователь захочет отка-

заться от своей графической оболочки и использовать стандартную, в качестве значения параметра ему нужно будет ввести значение "default". Следует помнить, что в клипе, исполняющем роль пользовательской графической оболочки, точка регистрации должна находиться в левом верхнем углу. Несоблюдение этого требования приведет к некорректной работе компонента. Пользовательский клип может состоять из нескольких кадров, содержащих разные изображения. Это позволит добиться большего разнообразия.

```
[Inspectable(name="Оболочка царапины", type = "String",↳
defaultValue="default", verbose = 1)]
public function set scratchSkin(value:String) {
       if(value == "default"){
       _customScratchSkin = "_scratch_mc";
       }
       else _customScratchSkin = value;
       createChildren();
       arrange();
}
public function get scratchSkin():String{
       if(_customScratchSkin == "_scratch_mc"){
       return "default";
       }
       else return _customScratchSkin;
}
[Inspectable(name="Оболочка пылинки", type = "String",↳
defaultValue="default", verbose = 1)]
public function set dustSkin(value:String) {
       if(value == "default"){
       _customDustSkin = "_dust_mc";
       }
       else _customDustSkin = value;
       createChildren();
       arrange();
}
public function get dustSkin():String{
       if(_customDustSkin == "_dust_mc"){
       return "default";
       }
       else return _customDustSkin;
}
```

Трансляция и обработка событий компонентов

Подобно многим встроенным объектам ActionScript компоненты могут генерировать события. Событие компонента может быть связано с протеканием внутренних процессов или с действиями пользователя. Поскольку компонент, по сути, является клипом, он автоматически получает возможность обрабатывать стандартные события клипов и кнопок точно так же, как и обычные клипы. При этом разработчик может наделить компонент возможностью генерировать пользовательские события. Это очень мощная и широко используемая на практике возможность. В этом разделе мы рассмотрим, каким образом выполняется обработка событий компонентов и как можно "заставить" компонент генерировать событие.

Генерирование и трансляция событий встроенных компонентов Flash реализуется с помощью встроенного класса `EventDispatcher`. Этот класс расположен в пакете mx.events, который расположен в стандартном каталоге Classes, содержащем все классы, поставляемые вместе с Flash MX 2004. Абсолютный путь к файлу с классом `EventDispatcher` имеет примерно следующий вид: **E:\Program files\Macromedia\Flash MX 2004\en\First Run\Classes\mx\events\EventDispatcher.as**.

В принципе, при создании пользовательских событий компонентов можно было бы обойтись без применения класса `EventDispatcher`. Генерирование и трансляцию событий можно реализовать, например, с помощью глобального объекта `AsBroadcaster`, описанного в *главе 11*, или создать собственный класс, реализующий эти функции. Тем не менее на практике, как правило, имеет смысл использовать именно встроенный класс `EventDispatcher`, поскольку он уже создан и, с одной стороны, предоставляет широкую функциональность, а с другой — имеет сравнительно небольшой объем. Применение данного класса при создании пользовательских компонентов также дает возможность прикреплять сценарий непосредственно к экземпляру компонента, используя для этого обработчик `on()`.

Событийная модель компонентов основана на использовании источников (broadcasters или dispatchers) и приемников (listeners) событий. Источником событий выступает экземпляр компонента, в то время как обязанности приемника может выполнять произвольный объект или функция. Вначале приведем некоторые теоретические сведения, относящиеся к вопросам генерирования, трансляции и обработки событий компонентов.

Инициализация источника событий

Для того чтобы превратить компонент в источник событий, достаточно выполнить несколько очень простых действий.

1. Перед объявлением управляющего класса компонента следует импортировать класс `EventDispatcher`, чтобы далее при обращении к нему из кода можно было использовать только имя.

   ```
   import mx.events.EventDispatcher
   ```

2. В классе компонента следует объявить три свойства типа `Function`, соответствующие методам класса `EventDispatcher`, непосредственно участвующим в генерировании и трансляции событий.

```
public var addEventListener:Function;
public var removeEventListener:Function;
private var dispatchEvent:Function;
```

Применение методов `addEventListener()` и `removeEventListener()` зачастую оказывается необходимо за пределами класса для реализации возможности обработки событий компонента внешними объектами и функциями подобно тому, как мы ранее использовали в аналогичных целях методы встроенных классов `addListener()` и `removeListener()`. В связи с этим данные методы объявлены с атрибутом `public`. Метод `dispatchEvent()` применяется для непосредственной трансляции события компонента приемникам. Поскольку транслирование события выполняется из внутреннего кода класса компонента, данный метод объявляется как защищенный с использованием атрибута `private`. Реализация всех этих методов содержится в классе `EventDispatcher`, однако они связываются с классом компонента уже после компиляции. Тем не менее обращение к этим свойствам содержится в методах, связанных с трансляцией событий управляющего компонентом класса. Для того чтобы на этапе компиляции Flash не выдавал сообщение об ошибке, возникающей в связи с обращением к необъявленным в классе методам, необходимо в коде класса объявить одноименные свойства, как и было сделано в нашем примере. В принципе, вместо создания свойств можно было бы объявить соответствующие методы, но не реализовывать их в классе (то есть оставить фигурные скобки пустыми).

```
public function addEventListener (){}
public function removeEventListener(){}
private function dispatchEvent(){}
```

Принципиальной разницы между этими подходами нет, однако объявление методов дает возможность отразить их в интерфейсе, который реализуется классом, в то время как свойства в интерфейсе присутствовать не могут.

3. В коде класса компонента необходимо осуществить вызов статического метода `EventDispatcher.initialize()`. Именно этот метод выполняет копирование методов класса `EventDispatcher`, связывая их с классом компонента и, таким образом, превращая компонент в источник событий.

Метод `initialize()` имеет следующий синтаксис:

```
mx.events.EventDispatcher.initialize(объект)
```

Если класс `EventDispatcher` был предварительно импортирован, то использовать полный путь к нему при вызове метода не требуется.

Параметр *объект* — это ссылка на объект, который должен выступать в качестве источника событий. В результате вызова метода `EventDispatcher.initialize()` в этот объект будут скопированы методы `addEventListener()`, `removeEventListener()`, `dispatchEvent()`, а также служебный метод `dispatchQueue()`, который применяет-

ся во внутренней реализации транслирования событий. Метод `EventDispatcher.initialize()` должен вызываться только один раз, обычно это выполняется в конструкторе или в методе `init()`, который вызывается из конструктора. Для того чтобы сделать экземпляр класса (то есть каждый компонент) источником события, достаточно поместить в его конструкторе следующее предложение:

```
mx.events.EventDispatcher.initialize(this)
```

Это приведет к тому, что методы класса `EventDispatcher` будут непосредственно скопированы в каждый экземпляр класса, то есть каждый экземпляр компонента будет обладать собственными копиями методов. Такой подход достаточно часто используется на практике, когда количество создаваемых экземпляров компонента невелико. Тем не менее при создании каждого нового экземпляра компонента в него копируются методы класса `EventDispatcher`, что, очевидно, создает некоторую избыточность.

В качестве одной из альтернатив можно выполнить инициализацию так, чтобы методы класса `EventDispatcher` были скопированы не непосредственно в экземпляры класса компонента, а в его прототип (объект, свойства которого наследуются всеми экземплярами класса). В этом случае не происходит избыточного дублирования методов, вместо этого экземпляры используют методы, хранящиеся в прототипе класса. Для реализации данного подхода нужно объявить в управляющем классе компонента статическое свойство и присвоить ему выражение, представляющее собой вызов статического метода `EventDispatcher.initialize()`, которому в качестве параметра следует передать ссылку на прототип конструктора:

```
private static var _event_mixin = EventDis-
patcher.initialize(DustAndScratches.prototype);
```

Трансляция события. Событийный объект

Реализация событийной модели класса `EventDispatcher` предполагает наличие объекта-источника, транслирующего событие, и объекта-приемника, выполняющего обработку этого события. Объект-источник транслирует свое событие только тем приемникам, которые зарегистрированы на его получение. В этом модель событий компонентов отличается от событийной модели, используемой объектом `ASBroadcaster`. При работе с объектом `ASBroadcaster` зарегистрированный приемник может обрабатывать все события, транслируемые источником. При использовании класса `EventDispatcher` каждый приемник регистрируется на получение какого-то определенного события. В нашем случае в качестве источника события выступает компонент.

Для того чтобы выполнить трансляцию события всем зарегистрированным на его получение приемникам, необходимо в момент наступления этого события вызвать метод `dispatchEvent()` подобно тому, как мы вызываем метод `broadcastMessage()` объекта `ASBroadcaster`. Метод `dispatchEvent()` имеет следующий синтаксис:

источник`.dispatchEvent(`*событийный объект*`)`

Здесь *источник* — это объект, выступающий источником события.

Параметр *событийный объект* — это объект `Object`, содержащий информацию о транслируемом событии и его источнике.

Событийный объект (event object) должен обладать, как минимум, двумя свойствами:

- `type` — данное свойство представляет собой строку, содержащую название транслируемого события;
- `target` — данное свойство содержит ссылку на объект, транслирующий данное событие.

Событийный объект всегда передается в качестве параметра обработчику события, позволяя таким образом получить информацию об обнаруженном событии и его источнике. Наряду с указанными свойствами событийный объект может включать и другие свойства, несущие информацию, которая может быть использована при обработке события объектом-приемником.

Событийный объект может быть передан методу `dispatchEvent()` по ссылке или в виде литерала. Следующий код демонстрирует модель класса `TestEvent`, экземпляры которого являются источниками пользовательского события `onCustomEvent`. Данное событие будет транслироваться экземплярами класса при вызове метода `broadcast()` всем зарегистрированным на его получение объектам.

```
import mx.events.EventDispatcher;
class TestEvent {
        public var addEventListener:Function;
        private var dispatchEvent:Function;
        public function TestEvent() {
                // Инициализируем экземпляр в качестве источника событий
                EventDispatcher.initialize(this);
        }
        public function broadcast(){
        // Транслируем событие при вызове метода broadcast()
        dispatchEvent({type:"onCustomEvent", target:this});
        }
}
```

Обработка событий компонентов

Для обработки событий компонентов можно использовать несколько различных способов. Рассмотрим каждый из этих способов.

Использование обработчика *on()*

Данный подход предполагает прикрепление программного кода, выполняющего обработку события, непосредственно к экземпляру компонента. Для этого используется стандартный обработчик событий `on()`, которому в качестве параметра следует передать имя события. Если в классе компонента имеются теги `Event`, Flash будет

выводить всплывающую подсказку, содержащую перечень всех событий, транслируемых данным компонентом.

Идентификатор `this`, используемый в теле обработчика `on()` компонента, возвращает ссылку на экземпляр данного компонента.

Использование объекта-приемника

Применение объекта-приемника похоже на использование приемников для обработки событий встроенных объектов ActionScript (таких как `Mouse`, `Stage` и др.).

В качестве объекта-приемника может быть использован имеющийся объект, специально созданный объект или текущий объект, ссылку на который возвращает идентификатор `this`.

Для обработки события компонента в объекте-приемнике необходимо создать свойство, имя которого должно совпадать с именем обрабатываемого события. Данному свойству нужно присвоить функцию, содержащую инструкции, которые будут выполнены при обнаружении события.

Для того чтобы объект-приемник мог реагировать на события источника, его необходимо зарегистрировать, используя для этого метод `addEventListener()`. Как мы помним, данный метод копируется в экземпляр компонента (или наследуется им из прототипа класса) при вызове статического метода `EventDispatcher.initialize()`. Метод `addEventListener()` имеет следующий синтаксис:

компонент`.addEventListener("`*событие*`", `*приемник*`)`

Здесь *компонент* — это экземпляр компонента, транслирующий событие.

Параметр "*событие*" — это строка, содержащая имя транслируемого события.

Параметр *приемник* — это ссылка на объект, выполняющий обработку события *событие* компонента.

Обратите внимание на то, что при регистрации приемника необходимо указывать, какое именно событие он должен обрабатывать.

Для того чтобы прекратить обработку события объектом-приемником, нужно отменить его регистрацию с помощью метода `removeEventListener()`, который принимает те же параметры, что и метод `addEventListener()`.

При вызове метода обработчика объекта-приемника ему в качестве параметра передается событийный объект, значение свойств которого можно использовать в теле метода.

Идентификатор `this`, используемый в теле метода обработчика объекта-приемника, возвращает ссылку на сам объект-приемник.

Продемонстрируем принцип применения данного способа на примере обработки события `onUpdate`, транслируемого компонентом с именем `ds`.

```
var obj:Object = new Object();
obj.onUpdate = function(evtObj){
trace(evtObj.type); // Выводит onUpdate
```

```
trace(evtObj.target); // Выводит ds
trace(this == obj);// Выводит true
}
ds.addEventListener("onUpdate", obj);
```

Вместо создания нового объекта-приемника мы могли использовать текущий объект (то есть объект, на монтажной линейке которого расположены код обработчика и экземпляр компонента). Это дало бы нам возможность обращаться из кода обработчика к экземпляру компонента непосредственно по имени:

```
this.onUpdate = function(evtObj){
trace(evtObj.type); // Выводит onUpdate
trace(evtObj.target); // Выводит ds
trace(this == obj);// Выводит _level0 (код на основной монтажной линейке)
trace(ds); // Выводит _level0.ds
}
ds.addEventListener("onUpdate", this);
```

Использование метода *handleEvent()*

Наряду с созданием у объекта-приемника метода обработчика, имя которого должно соответствовать имени обрабатываемого им события, можно использовать универсальный метод с именем `handleEvent()`. Всякий раз, когда компонент транслирует событие посредством вызова метода `dispatchEvent()`, при этом выполняется проверка того, имеется ли у зарегистрированного в качестве приемника объекта метод с именем `handleEvent()`. Если данный метод существует, осуществляется его вызов с передачей в качестве параметра событийного объекта. Таким образом, метод объекта-приемника, зарегистрированного на получение событий (в том числе и от разных компонентов), вызывается в результате обнаружения любого из них. Для определения того, какое событие произошло и какой объект является источником данного события, можно воспользоваться событийным объектом, который передается методу `handleEvent()` в качестве параметра. В коде метода `handleEvent()` обычно выполняется проверка с применением условных предложений или предложения `switch`, и в зависимости от того, какое событие инициировало вызов данного метода, осуществляется выполнение тех или иных инструкций.

Следующий условный пример иллюстрирует применение данного подхода. Объект `obj` здесь зарегистрирован на получение двух разных событий от двух разных объектов:

```
var obj:Object = new Object();
obj.handleEvent = function(evtObj) {
        switch (evtObj.type) {
        case "Событие1" :
                trace("Событие" + evtObj.type + "объекта" + ↳
                evtObj.target);
                break;
```

```
        case "Событие2" :
                trace("Событие" + evtObj.type + "объекта" +
                evtObj.target);
                break;
        }
};
ds1.addEventListener("Событие1", obj);
ds2.addEventListener("Событие2", obj);
```

Использование функции-приемника

Вместо применения объекта-приемника и создания у него метода обработчика для обработки событий компонента можно использовать функции-приемники. Для этого необходимо вначале объявить функцию и поместить в нее инструкции, которые будут выполняться при обнаружении события. После этого данную функцию необходимо зарегистрировать в качестве приемника событий компонента, используя метод `addEventListener()`. Когда компонент транслирует событие, выполняется вызов зарегистрированной функции, которой в качестве параметра передается событийный объект.

Важная особенность использования функции-приемника заключается в том, что областью видимости такой функции является источник события. Это, в частности, означает, что используемый в ее теле идентификатор `this` возвращает ссылку на экземпляр компонента, транслирующий событие, а не на монтажную линейку или класс, содержащие ее объявление. Продемонстрируем это на примере:

```
function handler (evtObj){
trace(evtObj.type); // Выводит onUpdate
trace(evtObj.target); // Выводит ds
trace(this); // Выводит ds
}
ds.addEventListener("onUpdate", handler);
```

Для изменения области видимости функции-приемника можно воспользоваться классом `mx.utils.Delegate`[1] *(см. в разд. "Изменение области видимости функции обработчика" данной главы).*

Использование зарезервированного метода компонента

При транслировании события у экземпляра компонента всегда вызывается особый метод, имя которого формируется из названия транслируемого события и суффикса `handler`. Так, при транслировании события `onUpdate` у экземпляра компонента будет вызываться метод `onUpdateHandler()`. Данное обстоятельство можно использовать в ситуации, когда компонент должен сам осуществлять обработку своих событий. Для этого достаточно просто объявить в управляющем классе компонента или непосредственно в экземпляре метод, имя которого отвечает описанным выше тре-

[1] Данный класс доступен только в обновленной версии Flash 7.2. — *Прим. автора.*

бованиям. При вызове этого метода ему в качестве параметра передается событийный объект. Областью видимости метода является содержащий его экземпляр компонента. Следующий пример демонстрирует применение метода `onUpdateHandler()`, который (как следует из его названия) вызывается при обнаружении события `onUpdate`, транслируемого экземпляром компонента с именем `ds`.

```
ds.onUpdateHandler = function(evtObj){
trace(evtObj.type); // Выводит onUpdate
trace(evtObj.target); // Выводит _level0.ds
trace(this); // Выводит _level0.ds
}
```

Изменение области видимости функции обработчика

Как уже было отмечено ранее, при использовании функции в качестве приемника областью ее видимости является не объект, содержащий объявление данной функции, а объект, транслирующий событие (то есть экземпляр компонента, вызвавший метод `addEventListener()`). Это может стать причиной проблем, связанных с невозможностью обратиться к членам класса из метода, объявленного в этом классе, который используется для обработки события компонента. Продемонстрируем это на примере. В FLA-документе создадим экземпляр компонента `DustAndScratches`, присвоим ему имя `ds` и конвертируем его в клип (**Modify>Convert to Symbol**), которому присвоим идентификатор `mc`. Свяжем данный символ с классом `TestEvent`, приведенным ниже.

```
import mx.events.EventDispatcher;
class TestEvent extends MovieClip {
        var addEventListener:Function;
        var ds:DustAndScratches; // Экземпляр компонента
        var foo = "foo";
        // Функция-приемник события onUpdate
        function myTrace() {
                trace(foo + " : " + this);
        }
        // При инициализации компонента регистрируем приемник
        function onLoad() {
                ds.addEventListener("onUpdate", myTrace);
        }
}
```

Если мы протестируем фильм, то в панель **Output** будет периодически выводиться следующее:

```
undefined : _level0.mc.ds
```

Полученный результат объясняется тем, что областью видимости метода myTrace() при вызове его в качестве приемника является не класс, содержащий его объявление, а экземпляр ds, транслирующий событие onUpdate. По этой причине мы не можем обратиться к члену класса foo из определения метода myTrace().

Для того чтобы исправить данную ситуацию, можно воспользоваться встроенным классом Delegate, который расположен в пакете mx.utils. Данный класс имеется только в обновленной версии Flash 7.2. Обновление можно свободно скачать с сайта компании Macromedia по следующему адресу: **http://www.macromedia.com/support/flash/downloads.html**.

Класс Delegate располагает единственным статическим методом, который называется Delegate.create(). Этот метод позволяет задать, объект, который будет находиться в области видимости функции-приемника, вызываемой при обнаружении события компонента. Вызов метода Delegate.create() непосредственно помещается в качестве второго параметра метода addEventListener(), регистрирующего требуемую функцию-приемник, поскольку возвращаемым значением данного метода является функция. При этом используется следующий синтаксис:

```
import mx.utils.Delegate;
компонент.addEventListener("событие", Delegate.create(объект, функция))
```

Здесь *компонент* — это экземпляр компонента, транслирующий событие *событие*.

Параметр *объект* — это объект, который должен представлять собой область видимости функции-приемника. На этот объект будет указывать идентификатор this, используемый в теле функции.

Параметр *функция* — это ссылка на функцию-приемник, выполняющую обработку события.

Метод Delegate.create() создает на основании переданной ему в качестве параметра функции новую функцию, которая будет вызываться при обнаружении события и инициировать вызов исходной функции-приемника так, чтобы ее областью видимости являлся не экземпляр компонента (как это происходит по умолчанию), а объект объект. Создаваемая данным методом функция и является его возвращаемым значением.

В качестве примера использования метода Delegate.create() скорректируем рассмотренный в начале данного раздела пример, сделав областью видимости метода myTrace() экземпляр класса TestEvent:

```
import mx.utils.Delegate;
import mx.events.EventDispatcher;
class TestEvent extends MovieClip {
        var addEventListener:Function;
        var ds:DustAndScratches; // Экземпляр компонента
        var foo = "foo";
        // Функция-приемник события onUpdate
        function myTrace() {
```

```
            trace(foo + " : " + this);
        }
        // При инициализации компонента регистрируем приемник
        function onLoad() {
        // В качестве области видимости myTrace() будем использовать
        // экземпляр класса, содержащий его объявление
        ds.addEventListener("onUpdate", Delegate.create(this, myTrace));
        }
}
```

Теперь при тестировании в панель **Output** будет периодически выводиться следующее:

```
foo : _level0.mc.
```

Трансляция пользовательского события *onUpdate*

В обязанности нашего компонента `DustAndScratches` входит транслирование события, которое мы назвали `onUpdate`. Это событие должно транслироваться периодически с частотой, значение которой хранится в свойстве `_frequency`. Для того чтобы экземпляры компонента могли служить источниками события, необходимо наделить их методами класса `EventDispatcher`, отвечающими за транслирование пользовательских событий. Чтобы не копировать эти методы в каждый экземпляр компонента, мы поместим их в прототип конструктора класса `DustAndScratches`, объявив в нем статическое свойство:

```
private static var _event_mixin =↳
EventDispatcher.initialize(DustAndScratches.prototype);
```

Для периодического транслирования события мы будем использовать функцию `setInterval()`. В связи с этим создадим еще одно защищенное свойство, предназначенное для хранения идентификационного номера интервала:

```
private var _interval:Number;
```

Для того чтобы событие `onUpdate` транслировалось периодически, очевидно необходимо с заданной частотой вызывать метод `dispatchEvent()`, передавая ему в качестве параметра событийный объект. Можно было бы выполнить это непосредственно, инициировав периодический вызов метода `dispatchEvent()` с помощью глобальной функции `setInterval()`. Однако в этом случае частота транслирования нашего события будет зависеть от скорости воспроизведения, используемой в документе. Для того чтобы свести эту зависимость к минимуму, следует после каждого вызова метода `dispatchEvent()` выполнять обновление экрана. Поэтому мы создали вспомогательный метод `update()`, который в качестве параметра принимает событийный объект. В обязанности данного метода входит вызов метода `dispatchEvent()` с передачей ему соответствующего событийного объекта и последующее обновление экрана с помощью функции `updateAfterEvent()`.

```
private function update(eventObj:Object):Void {
        dispatchEvent(eventObj);
```

```
        updateAfterEvent();
}
```

Теперь, для того чтобы инициировать периодическую трансляцию события `onUpdate`, достаточно выполнять периодический вызов метода `update()`. Для этого мы создадим метод `setUpdate()`, который будет устанавливать интервал для вызова метода `update()` и передавать ему в качестве параметра событийный объект, свойство `target` которого будет хранить ссылку на текущий экземпляр, а свойство `type` — строку `"onUpdate"`. Идентификационный номер интервала мы сохраним в свойстве `_interval` с тем, чтобы его в дальнейшем можно было при необходимости очистить, прекратив таким образом трансляцию события.

```
private function setUpdate(Void):Void {
        var eventObj = {target:this, type:"onUpdate"};
        _interval = setInterval(this, "update", _frequency, eventObj);
}
```

Для того чтобы трансляция события началась сразу же после создания экземпляра компонента, метод `setUpdate()` следует вызвать в конструкторе класса. Мы поместим вызов метода `setUpdate()` в метод `init()`, который будет вызван из конструктора.

Поскольку наш компонент предполагает возможность отключения посредством внешнего `get/set`-свойства `active`, желательно прекратить трансляцию событий, когда компонент находится в отключенном состоянии. Для обеспечения этой возможности создадим метод `removeUpdate()`.

```
private function removeUpdate(Void):Void{
        clearInterval(_interval);
}
```

Теперь при отключении компонента с помощью свойства `active` в соответствующем `set`-методе выполняется вызов метода `removeUpdate()`, прекращающего транслирование события, а при повторном включении компонента вызывается метод `setUpdate()`, возобновляющий транслирование.

Приемником нашего события `onUpdate` может являться любой объект. В частности, все экземпляры клипов, представляющих пылинки и царапины, как это будет показано далее, сразу же после своего создания регистрируются в качестве приемников данного события.

Инициализация компонента

При создании экземпляра компонента нужно выполнить ряд действий, необходимых для его корректного отображения и функционирования. Обязанности, связанные с инициализацией компонента, мы возложим на метод `init()`, который будет вызываться только один раз при создании экземпляра компонента.

Для этого вызов метода `init()` будет помещен в конструктор класса `DustAndScratches`:

```
public function DustAndScratches(Void){
        init();
}
```

Рассмотрим, какие действия выполняет метод `init()`:

```
private function init(Void):Void {
```

Во-первых, нам необходимо корректно отобразить экземпляр компонента, учитывая размеры, которые пользователь установил для него в рабочей среде (с помощью инструмента **Free Transform**, панели **Transform** или иным способом) Дело в том, что при масштабировании экземпляра компонента в рабочей среде, как и при масштабировании обычного клипа, масштабируется и его содержимое. При этом также изменяются значения свойств `_xscale` и `_yscale`. В случае компонента масштабирование его содержимого является абсолютно недопустимым, поскольку при этом нарушаются пропорции графических элементов, что приводит к искажению его внешнего вида. Размер элементов, составляющих графическую оболочку компонента, должен устанавливаться программным образом и не зависеть от "ручного" масштабирования, выполняемого пользователем. "Ручное" масштабирование должно определять только размер области, занимаемой экземпляром компонента, а масштабирование и позиционирование входящей в его состав графики с учетом заданных пользователем размеров области входит в обязанности управляющего компонентом кода. В связи с этим для хранения истинных размеров компонента (то есть размеров, заданных пользователем), мы ввели свойства `__width` и `__height`. При инициализации компонента нам необходимо сохранить в этих внутренних свойствах его истинные размеры (то есть абсолютные значения ширины и высоты, которые пользователь установил в рабочей среде), хранящиеся в наследуемых от класса `MovieClip` свойствах `_width` и `_height`. В дальнейшем при необходимости определения размеров компонента программный код, расположенный внутри класса, будет обращаться именно к свойствам `__width` и `__height`.

```
        __width = _width;
        __height = _height;
```

Теперь мы располагаем сведениями о размерах экземпляра компонента, однако если пользователь изменял размеры компонента в рабочей среде, то, очевидно, содержимое компонента подверглось масштабированию. Для того чтобы восстановить исходные пропорции содержимого компонента, нужно присвоить его свойствам `_xscale` и `_yscale` (наследуемым от класса `MovieClip`) значение 100. В результате этого действия содержимое компонента примет исходные пропорции. Для того чтобы это масштабирование не привело к повторному изменению размеров компонента, необходимо после динамического присоединения элементов вызвать метод `arrange()`, который установит размеры графических элементов компонента и позиционирует их на основании значений размеров компонента, хранящихся в свойствах `__width` и `__height`. Таким образом, после инициализации компонента его размер будет соответствовать размеру, установленному пользователем, в то время как свойства `_xscale` и `_yscale` будут принимать значение 100% и пропорции графических

объектов составляющих компонент не будут нарушены. Аналогичный подход применяется и в архитектуре встроенных компонентов.

```
_xscale = 100;
_yscale = 100;
```

Если экземпляр компонента будет создаваться не вручную, а программным образом с помощью метода `attachMovie()`, то экземпляр компонента будет иметь размер, соответствующий размеру ограничивающей рамки. Передав методу `attachMovie()` объект инициализации, можно сразу же задать размеры создаваемого экземпляра.

После того как мы инициировали свойства, отвечающие за размеры, необходимо скрыть ограничивающую рамку компонента, роль которой выполняет клип `_boundingBox`. Для этого мы просто отключаем ее отображение и уменьшаем размеры до нуля.

```
_boundingBox._visible = false;
_boundingBox._width = 0;
_boundingBox._height = 0;
```

Если экземпляр компонента создается вручную, то все свойства, влияющие на его работу, устанавливаются пользователем с помощью параметров, выведенных в панель **Properties** (или используются значения по умолчанию). Однако, если экземпляр создается динамически с помощью `attachMovie()`, необходимо инициировать все нужные свойства, присвоив им значения, используемые по умолчанию.

```
if(_dustNum == undefined) _dustNum=_quantityFigures[2];
if (_scratchNum == undefined) _scratchNum=_quantityFigures[2];
if(_dustSize == undefined) _dustSize=_magnitudeFigures[1];
if(_scratchSize == undefined) _scratchSize=_magnitudeFigures[1];
```

Для того чтобы новый экземпляр сразу начинал транслировать событие `onUpdate`, вызываем метод `setUpdate()`.

```
setUpdate();
```

Теперь необходимо присоединить из библиотеки экземпляры клипов, представляющих пылинки и царапины.

```
createChildren();
```

И, наконец, вызываем метод `arrange()`, позиционирующий клипы и придающий им исходные размеры.

```
arrange();
}
```

Изменение размеров компонента

При использовании режима предварительного просмотра (Live Preview) в рабочей среде изменение размеров компонента должно сопровождаться выполнением пози-

ционирования (и при необходимости масштабирования) составляющих его графических элементов, подобно тому, как это делается при инициализации экземпляра. Для реализации данной задачи в управляющем классе компонента необходимо объявить метод `setSize()`, который в качестве параметров должен принимать два числа, задающие новую ширину и высоту экземпляра. Поставляемый вместе с Flash программный код, который обеспечивает функционирование режима предварительного просмотра, всегда вызывает метод компонента `setSize()`, когда пользователь изменяет его размер посредством любых инструментов рабочей среды. Метод `setSize()` может использоваться не только в целях обеспечения корректной работы режима предварительного просмотра, его также можно использовать как элемент программного интерфейса компонента для динамического изменения его размеров. В случае с нашим компонентом программное изменение размеров может выполняться как непосредственно с помощью данного метода, так и с применением `get/set`-свойств `width` и `height`, в основе работы которых, впрочем, лежит все тот же метод `setSize()`. В архитектуре встроенных компонентов свойства `width` и `height` предназначены только для чтения, а размер компонента задается с помощью метода `setSize()`.

Механизм работы метода `setSize()` заключается в масштабировании компонента до значения 100% и записи новых значений ширины и высоты в свойства `__width` и `__height`. После этого выполняется вызов метода `arrange()`, позиционирующего элементы в области, соответствующей новым размерам компонента. Таким образом, его фактический размер будет соответствовать заданным значениям, в то время как свойства `_xscale` и `_yscale` будут принимать значение 100, что обеспечит сохранение пропорций.

```
public function setSize(w:Number, h:Number):Void {
        _xscale = 100;
        _yscale = 100;
        __width = w;
        __height = h;
        arrange();
}
```

Динамическое присоединение графических элементов компонента

Динамическое присоединение клипов, представляющих пылинку и царапину, реализуется с помощью метода `createChildren()`. Присоединение должно выполняться только в том случае, если компонент включен, то есть его свойство `_active` принимает значение `true`. Поскольку пользователь должен иметь возможность независимо включать и отключать отображение пылинок или царапин, присоединение соответствующих элементов необходимо выполнять по отдельности (в разные контейнеры) и только в том случае, если их количество, заданное пользователем, отлично от нуля.

Для размещения всех клипов мы создадим пустой клип с именем `holder`, исполняющий роль внешнего контейнера. В свою очередь, в клипе мы создадим два клипа-контейнера: `dustHolder` — для клипов пылинок и `scratchHolder` — для клипов царапин.

Для того чтобы наделить динамически создаваемые клипы соответствующим поведением, мы свяжем их родительские символы с классами `AnimateDust` и `AnimateScratch` с помощью метода `Object.registerClass()`. Поскольку наш компонент предполагает возможность применения пользовательских графических оболочек, методу `Object.registerClass()` в качестве идентификатора связи символа, с которым связывается класс, передается свойство `_customDustSkin`, указывающее на требуемый символ. После создания каждого экземпляра мы сразу регистрируем его в качестве приемника события `onUpdate`.

Ниже приведен программный код метода `createChildren()`, снабженный подробными комментариями.

```
private function createChildren(Void):Void {
// Если компонент включен...
if(_active){
        // Создаем клип-контейнер
        createEmptyMovieClip("holder", 1);
        // Если количество пылинок отлично от нуля...
        if (_dustNum) {
        // В контейнере holder создаем внутренний контейнер только для
        // клипов, представляющих пылинки
                holder.createEmptyMovieClip("dustHolder", 1);
                // Связываем символ пылинки с классом AnimateDust,
                // обеспечивающим функциональность пылинок (их анимацию)
                Object.registerClass(_customDustSkin, AnimateDust);
                // Помещаем в компонент экземпляры "пылинок"
                for (var i = 0; i < _dustNum; i++) {
                holder.dustHolder.attachMovie(_customDustSkin,"dust"+i, i);
                // Регистрируем клипы "пылинок" в качестве приемников
                // события onUpdate
                addEventListener("onUpdate", holder.dustHolder["dust"+ i]);
                }
        }
        // С царапинами - аналогично
        if (_scratchNum) {
                holder.createEmptyMovieClip("scratchHolder", 2);
                Object.registerClass(_customScratchSkin, AnimateScratch);
```

```
            for (var k = 0; k < _scratchNum; k++) {
            holder.scratchHolder.attachMovie(_customScratchSkin,↵
            "scratch" + k, k);
            addEventListener("onUpdate",holder.scratchHolder["scratch"↵
            + k]);
}}}}
```

Компоновка составляющих элементов

После выполнения динамического присоединения графических элементов, входящих в состав компонента, необходимо позиционировать их и придать им требуемые размеры, исходя из фактических размеров компонента. Для этих целей мы создали метод `arrange()`, который случайным образом распределяет клипы, представляющие пылинки и царапины в пределах области компонента. Кроме того, в методе `arrange()` содержится вызов метода `drawMask()`, который осуществляет маскирование области компонента для того, чтобы пылинки и царапины не выходили за его границы.

Размеры пылинок устанавливаются с учетом размеров области компонента, заданной величины пылинок и элемента случайности. Высота пылинки устанавливается равной ширине. Для царапин устанавливается только высота. Их ширина остается неизменной. Кроме случайных координат и размеров элементам сообщается случайная прозрачность.

```
private function arrange(Void):Void{
        // Установка размеров и позиционирование пылинок
        for (var i = 0; i < _dustNum; i++) {
                // Размеры пылинок
                holder.dustHolder["dust" + i]._width = Math.random() *↵
                (__width + __height) / 10 * _dustSize;
                holder.dustHolder["dust"+i]._height=↵
                holder.dustHolder["dust"+i]._width;
                // Координаты пылинок
                holder.dustHolder["dust"+i]._x = Math.random() * __width;
                holder.dustHolder["dust"+i]._y = Math.random()* __height;
                // Прозрачность пылинок
                holder.dustHolder["dust"+i]._alpha = Math.random()*__alpha;
        }
        // Установка размеров и позиционирование царапин
        for (var k = 0; k < _scratchNum; k++) {
                holder.scratchHolder["scratch" + k]._x = Math.random()*↵
                __width;
                holder.scratchHolder["scratch" + k]._y = Math.random()*↵
                __height;
```

```
                holder.scratchHolder["scratch" + k]._height =__height *↳
                _scratchSize * (1+Math.random());
                holder.scratchHolder["scratch" + k]._alpha =↳
                Math.random()*__alpha;
        }
        drawMask();
}
```

Принцип работы метода `drawMask()` заключается в том, что он создает пустой клип с именем `mask`, в котором динамически рисуется прямоугольник, размеры которого совпадают с размерами компонента (поскольку центр регистрации компонента находится в его левом верхнем углу, дополнительно позиционировать клип `mask` не требуется). Затем клип `mask` используется в качестве маски для клипа `holder` (в котором находятся все графические элементы компонента). Таким образом, элементы, выходящие за границы компонента, становятся не видны. Обратите внимание на то, что клип `mask` мы помещаем на второй уровень стека компонента, поскольку первый уровень занят клипом `holder`.

```
private function drawMask(Void):Void{
        if(active){
                createEmptyMovieClip("mask", 2);
                mask.moveTo(0, 0);
                mask.beginFill(0xFFFF);
                mask.lineTo(__width, 0);
                mask.lineTo(__width, __height);
                mask.lineTo(0, __height);
                mask.lineTo(0, 0);
                mask.endFill();
                holder.setMask(mask);
        }
}
```

Обеспечение функциональности графических элементов компонента

Анимация хаотического мерцания пылинок и движения царапин реализуется с помощью классов `AnimateDust` и `AnimateScratch`, которые мы связали с экземплярами соответствующих клипов непосредственно перед их созданием.

При создании экземпляров, представляющих пылинки и царапины, метод `createChildren()` сразу же регистрирует их в качестве приемников события `onUpdate`. Для реализации анимации этих экземпляров достаточно объявить в управляющих ими классах метод `onUpdate()`, исполняющий роль обработчика события `onUpdate`, которое генерируется компонентом.

Поскольку при позиционировании элементов нам необходимо учитывать фактические размеры компонента, следует объявить в классе свойство `parent`, хранящее ссылку на родительский экземпляр компонента.

В обязанности класса `AnimateDust`, реализующего анимацию пылинок, входит:

- переход на кадр, номер которого определяется случайным образом. Если клип, изображающий пылинку, состоит из нескольких кадров, содержащих разные изображения, то случайный выбор кадра сделает анимацию более разнообразной;
- позиционирование пылинок. Координаты точки, в которую помещается экземпляр клипа, представляющего пылинку, определяются случайным образом внутри области, занимаемой компонентом;
- задание размеров пылинок. Размер пылинок мы определяем случайным образом в диапазоне от нуля до числа, которое зависит от размеров области компонента и заданного пользователем значения параметра, определяющего размер пылинки. Для того чтобы добиться большей реалистичности изменение размеров и местоположения пылинок выполняется не при каждом вызове метода `onUpdate()`, а с вероятностью 60%. Это означает, что некоторые пылинки могут присутствовать на экране более длительное время, чем их соседи (причем их внешний вид может меняться, поскольку при каждом вызове метода `onUpdate()` выполняется переход на произвольный кадр);
- задание случайной прозрачности (в пределах значения, заданного пользователем с помощью соответствующего параметра) и случайного угла поворота.

В следующем листинге приведен исходный код класса `AnimateDust`. Исходный код читатель сможет найти в файле AnimateDust.as, расположенном в папке Глава 27 на сопровождающем книгу компакт-диске.

Класс `AnimateDust`. Реализация поведения пылинок

```
class AnimateDust extends MovieClip {
        private static var parent:Object;
        function AnimateDust() {
                // Сохраняем ссылку на родительский экземпляр компонента
                parent = _parent._parent._parent;
        }
        function onUpdate() {
                var rnd = Math.random();
                this.gotoAndStop(Math.ceil(rnd * _totalframes));
                if (rnd < .6) {// В 60% случаев...
                        _x = Math.random() * parent.width;
                        _y = Math.random() * parent.height;
                        _width = Math.random() * (parent.width +↵
                        parent.height) / 10 * parent.dustSizeValue;
                        _height = _width;
                }
```

```
            if (rnd > 0.9) {// В 10% случаев...
                _width = Math.random() * (parent.width +
                parent.height) / 5 * parent.dustSizeValue;
                _height = _width;
            }
            _alpha = Math.random() * parent.alpha;
            _rotation = Math.random() * 360;
}}
```

Класс `AnimateScratch` реализует функциональность, связанную с анимацией царапин. По структуре он очень похож на класс `AnimateDust`. Основное отличие поведения царапин состоит в том, что представляющие их клипы должны перемещаться сверху вниз. Скорость перемещения для всех царапин должна быть одинаковой (поскольку они находятся на единой пленке), поэтому в классе `AnimateScratch` мы создадим статическое свойство `_increment`, в котором будет храниться значение расстояния, на которое царапины будут перемещаться при каждом вызове метода `onUpdate()`. Значение свойства `_increment` должно быть прямо пропорционально высоте компонента и обратно пропорционально частоте обновления, заданной пользователем. Это позволит снизить зависимость скорости перемещения царапин от размеров компонента и повысить плавность их движения при низкой частоте обновления.

В следующем листинге приведен исходный код класса `AnimateScratch`. Исходный код читатель сможет найти в файле AnimateScratch.as, расположенном в папке Глава 27 на сопровождающем книгу компакт-диске.

Класс `AnimateScratch`. Реализация поведения царапин

```
class AnimateScratch extends MovieClip {
        private static var _increment:Number;
        private static var parent:Object;
        function AnimateScratch() {
                // Сохраняем ссылку на родительский экземпляр компонента
                parent = _parent._parent._parent;
                // Шаг смещения
                _increment = 20 * parent.height / parent.frequency;
        }
        function onUpdate() {
                var rnd = Math.random();
                _y += _increment;
                // Если царапина достигла нижней границы компонента...
                if (_y >= parent.height) {
                        this.gotoAndStop(Math.ceil(rnd * _totalframes));
                        // Изменяем высоту царапины
```

```
                    _height = parent.height * parent.scratchSizeValue*↳
                    (rnd + 1);
                    // Позиционируем царапину над компонентом
                    _x = Math.random() * parent.width;
                    _y = -2 * _height;
                    if (Math.random() < .1) {// В 10% случаев...
                          _height = parent.height *↳
                          parent.scratchSizeValue * (rnd + 5);
}}}}
```

Использование режима предварительного просмотра (Live Preview)

Характерной особенностью применения компонентов является возможность использования режима предварительного просмотра (Live Preview), позволяющего непосредственно в рабочей среде отображать компонент так же, как он будет выглядеть в конечном фильме. Для того чтобы включить или отключить режим предварительного просмотра, пользователь должен выполнить команду главного меню **Control> Enable Live Preview**. Режим предварительного просмотра функционирует только для компилированного клипа (то есть уже готового компонента).

Активизация режима предварительного просмотра приводит к тому, что компилированный клип, помещенный на сцену, непосредственно в рабочей среде Flash воспроизводится в окне проигрывателя, размер которого совпадает с его собственным размером. Воспроизведение компилированного клипа в рабочей среде в режиме предварительного просмотра имеет ряд характерных особенностей:

- в компилированном клипе не работают обработчики событий, контролирующие мышь и клавиатуру (`onPress`, `onRollOver`, `onKeyDown` и т. д.);
- в компилированном клипе не работает анимация монтажной линейки;
- в компилированном клипе не работает обработчик события `onEnterFrame`;
- в компилированном клипе не работает глобальная функция `setInterval()`.

В режиме предварительного просмотра отсутствует возможность непосредственного взаимодействия с клипом и не выполняется анимация. Таким образом, пользователь видит только статичное изображение компонента, что, впрочем, и требуется на этапе создания исходного документа.

Главной особенностью режима предварительного просмотра является то, что Flash автоматически обновляет изображение компонента при каждом изменении его параметров в панели **Properties** или **Component Inspector** и при каждом изменении его размеров. Такая возможность позволяет пользователю сразу же видеть результат выполняемых им действий, что значительно облегчает процедуру настройки компонента. Обновление изображения компонента также выполняется при каждом щелчке мыши в пределах сцены. Это особенно хорошо заметно на примере нашего компонента. Поскольку в режиме предварительного просмотра функция `setInterval()` не действует, вместо мелькания пылинок и царапин мы будем видеть просто статичную

картинку. Тем не менее при щелчке мышью в области сцены Flash будет выполнять обновление изображения нашего компонента, в результате чего положение пылинок и царапин изменится. Функциональность режима предварительного просмотра обеспечивается частично низкоуровневым механизмом рабочей среды Flash, а частично — программным кодом ActionScript, который автоматически включается в код первого кадра компилированного клипа. Данный программный код расположен в каталоге Include, полный путь к нему имеет примерно следующий вид: **E:\Program files\Macromedia\Flash MX 2004\en\First Run\Include\ mm_livepreview.as**.

Режим предварительного просмотра обладает еще одной любопытной особенностью. Когда пользователь создает экземпляр компонента или изменяет значение его параметра, программный код, автоматически включаемый в состав компонента, вызывает метод компонента `onUpdate()`. Если в управляющем классе компонента реализовать метод с таким названием, то это даст нам возможность выполнить определенные действия, специфичные только для режима предварительного просмотра. Так, например, с помощью данного метода можно в рабочей среде выводить в компоненте информацию об авторских правах. Поскольку метод компонента `onUpdate()` вызывается в режиме предварительного просмотра, данное сообщение будет присутствовать только в рабочей среде. В конечном фильме пользователь его не увидит.

Приведем код метода `onUpdate()`, объявленного в нашем классе `DustAndScratches`.

```
private var _txt:TextField;
private function onUpdate(Void):Void{
        // Выводим информацию об авторских правах
        createTextField("_txt", 3, 0, 0, 0, 0);
        _txt.autoSize = "left";
        _txt.background = true;
        _txt.backgroundColor = 0x888888;
        _txt.textColor = 0xFFFFFF;
        _txt.text = "© 2004 D.Albert, E.Albert, book_flash@narod.ru";
}
```

Распространение компонента

После окончания работы по созданию клипа компонента и написанию управляющих классов (а также на протяжении работы в целях тестирования) необходимо превратить исходные составляющие элементы компонента в единый конечный продукт, предназначенный для дальнейшего распространения. Эта процедура может включать в себя несколько этапов.

- Компиляция клипа компонента.
- Экспортирование SWC-файла.
- Создание MXP-пакета расширения.

В заключение этой главы мы приведем описание выполнения данных этапов.

Компиляция клипа компонента

Для того чтобы превратить исходный клип компонента, созданный в рабочей среде, в "полноценный" компонент, необходимо преобразовать его в компилированный клип. При создании компилированного клипа Flash создает SWF-документ, включающий в свой состав все элементы компонента, в том числе графику и программный код управляющих классов. Для дальнейшего функционирования компилированный клип уже не нуждается в наличии внешних файлов классов, поскольку их код включен в состав клипа. Характерной особенностью компилированного клипа является невозможность изменения его идентификатора связи и переназначения связанного с ним класса.

Для того чтобы создать компилированный клип, необходимо правой кнопкой мыши щелкнуть на имени исходного клипа в библиотеке и из контекстного меню выбрать команду **Convert to Compiled Clip**.

Экспортирование SWC-файла

Теоретически компилированный клип можно непосредственно распространять как готовый компонент в качестве элемента библиотеки FLA-документа, поскольку он обладает полной функциональностью. Однако такой подход неудобен, поскольку требует переноса компонента из библиотеки одного документа в библиотеку другого документа вручную.

С выходом Flash MX 2004 компоненты стали распространяться в особом формате SWC. SWC-файл представляет собой обычный zip-архив (его можно открыть любым zip-архиватором), в состав которого входят следующие файлы.

- Компилированный клип (в формате SWF), то есть собственно компонент.
- Вспомогательные файлы с расширением asi, которые соответствуют используемым компонентом классам и содержат перечень свойств и методов этих классов. Данные файлы включают только информацию о типах значений свойств, принимаемых методами, параметрах и их типе, а также о наличии и типе возвращаемых методами значений. Кроме того, ASI-файлы включают теги метаданных, используемые в управляющих классах компонента. Назначение данных файлов состоит в предоставлении среде Flash информации, необходимой для осуществления контроля типов, выполняемого при компиляции. На функционирование компонента наличие данных файлов не влияет.
- XML-документ с именем catalog.xml. Данный документ содержит служебную информацию, используемую рабочей средой Flash. Здесь содержатся данные, описывающие характеристики компонента, используемые в управляющем классе теги метаданных, параметры компонента, последовательность инициализации классов и др. Структура этого документа достаточно проста и очевидна. Читатель сможет убедиться в этом, открыв файл catalog.xml, который входит в состав SWC-архива DustAndScratches.swc, находящегося в папке Глава 27 на сопровождающем книгу компакт-диске. Просмотреть и при необходимости вручную отредактировать этот документ можно, воспользовавшись, любым zip-архиватором. В случае внесения изменений необходимо убедиться в том, что SWC-архив содержит новую версию этого файла.

- Графический файл в формате PNG, содержащий пиктограмму компонента. Если для компонента не создавалась пользовательская пиктограмма, сюда будет помещена пиктограмма, используемая по умолчанию.

SWC-архив также может включать в свой состав SWF-документы, используемые для создания пользовательского ролика предварительного просмотра и пользовательского ролика для настройки интерфейса компонента.

Для того чтобы создать SWC-архив, необходимо правой кнопкой мыши щелкнуть на имени исходного клипа компонента в библиотеке и из контекстного меню выбрать команду **Export SWC File...**. После этого необходимо указать каталог, в котором требуется сохранить SWC-архив.

Доступ ко всем встроенным компонентам Flash в рабочей среде обеспечивает специальная панель **Components** (**Window>Development Panels>Components**). Пользовательский компонент также может быть помещен в эту панель. Для того чтобы пользовательский компонент отображался в панели **Components**, необходимо скопировать содержащий его SWC-архив в стандартный каталог, используемый Flash для хранения компонентов. Путь к этому каталогу имеет примерно следующий вид: **C:\WINDOWS\Application Data\Macromedia\Flash MX 2004\en\Configuration\Components**. В панели **Components** все компоненты разбиты на группы. Каждой группе соответствует папка в стандартном каталоге для хранения компонентов Components. SWC-архив можно поместить в одну из имеющихся папок. В этом случае пользовательский компонент будет входить в одну из стандартных категорий. При необходимости можно создать собственную папку. В этом случае в панели компонентов будет добавлена новая категория, название которой будет совпадать с названием созданной папки. Если SWC-архив будет помещен непосредственно в корневой каталог, то в панели **Components** будет создана категория с названием **Standard Components**.

Для того чтобы пользовательский компонент после экспортирования в формат SWC и сохранения в стандартном каталоге был выведен в панель **Components**, необходимо либо перезапустить Flash, либо в контекстном меню панели **Components** выполнить команду **Reload**.

Создание MXP-пакета расширения

Наряду с применением формата SWC для распространения компонентов также может быть использован формат MXP, применяемый для хранения расширений для различных продуктов Macromedia. Расширение (extension) — это программный модуль, который можно установить с помощью специальной утилиты Extensions Manager. Новейшую версию Extensions Manager можно загрузить с сайта компании Macromedia по адресу: **http://www.macromedia.com/go/flash_exchange**.

Для того чтобы пользователь смог инсталлировать MXP-расширение, у него должна быть установлена современная версия Extensions Manager. Преимущество использования MXP-расширения вместо SWC-архива заключается в том, что пользователь не должен вручную копировать файл в стандартный каталог, это будет выполнено автоматически. Кроме того, Extensions Manager дает пользователю возможность легко манипулировать расширениями, включать, отключать, а при необходимости удалить

их. Процедура установки расширения напоминает стандартную процедуру установки программного обеспечения.

Для того чтобы установить расширение, пользователю достаточно дважды щелкнуть на соответствующем файле. В результате этого будет выполнен запуск Extensions Manager, окно которого представлено на рис. 27.5. Для того чтобы установленным расширением можно было воспользоваться, необходимо перезапустить Flash.

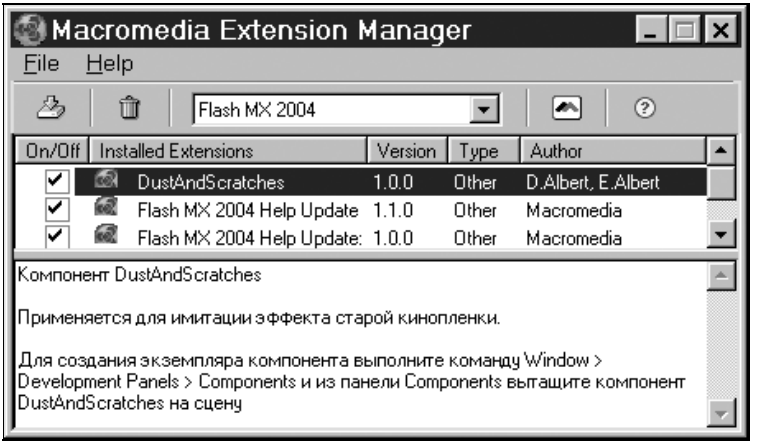


Рис. 27.5. Окно утилиты Extensions Manager

Для управления установленными расширениями необходимо запустить Extensions Manager. Это можно сделать либо непосредственно из файловой системы, либо из рабочей среды Flash MX 2004 с помощью команды **Help>Manage Extensions**. В выпадающем списке в верхней части окна Macromedia **Extensions Manager** необходимо выбрать Flash MX 2004. В результате этого в нижней части окна будет выведен перечень всех установленных Flash-расширений. При выделении расширения в нижней части окна выводится информация о нем. Для того чтобы отключить установленное расширение, необходимо снять установленный слева от его имени флажок. В результате этого действия расширение отключается, а соответствующий MXP-файл перемещается в специальную папку Disabled, расположенную по адресу: **C:\WINDOWS\Application Data\Macromedia\Flash MX 2004\en\Configuration\Extensions\Disabled**. Для того чтобы вновь подключить расширение, необходимо установить соответствующий флажок. Для того чтобы удалить расширение, нужно выделить его название и либо щелкнуть по кнопке с изображением мусорной корзины, либо в меню **File** выполнить команду **Remove Extension** (если расширение используется рабочей средой Flash, то удалить его можно будет, только закрыв Flash и запустив утилиту Extensions Manager из файловой системы).

Каким же образом создается MXP-расширение, предназначенное для распространения компонента? Для его создания необходимо наличие современной версии Extensions Manager.

Сам процесс достаточно прост, он включает три этапа:

1. Создание SWC-архива с полным набором составляющих элементов компонента.
2. Создание установочного MXI-файла.
3. Упаковка расширения.

Для того чтобы создать MXP-расширение, нам потребуется создать установочный файл. Этот файл имеет расширение mxi и представляет собой обычный XML-документ, содержащий информацию, необходимую для выполнения упаковки расширения. Для его создания можно воспользоваться любым XML- или HTML-редактором (например, Macromedia Dreamweaver). В приведенном далее листинге содержится образец такого файла для нашего компонента `DustAndScratches` с комментариями, отражающими назначение используемых элементов. Исходный файл DustAndScratches.mxi читатель сможет найти на сопровождающем книгу компакт-диске в папке Глава 27. Более подробную информацию о формате установочного файла читатель сможет найти на сайте компании Macromedia по адресу **http://www.macromedia.com/go/em_file_format**.

MXI-файл, используемый для упаковки расширения

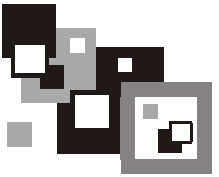
```
<!-- Название, версия и тип расширения -->
<macromedia-extension
       name="DustAndScratches"
       version="1.0"
       type="Component">
       <!-- Имя автора -->
       <author name="D.Albert, E.Albert" />
       <!-- Приложение, для которого предназначено расширение -->
       <products>
              <product name="Flash" version="7" primary="true" />
       </products>
       <!-- Информация о расширении, которая будет выводиться в окне Ex-
       tension Manager -->
       <description>
       <![CDATA[
       Компонент DustAndScratches<br><br>
       Применяется для имитации эффекта старой кинопленки.
       ]]>
       </description>
       <!-- Информация о том, как получить доступ к расширению из рабочей
       среды. Эти данные также будут выводиться в окне -->
       <ui-access>
       <![CDATA[
       Для создания экземпляра компонента выполните команду Window >
       Development Panels > Components и вытащите компонент на сцену
```

```
        ]]>
        </ui-access>
        <!-- Имя SWC-файла и каталог, в который он должен быть установлен.
        При создании расширения SWC-файл должен находиться в одном
        каталоге с MXI-файлом. Для указания стандартного каталога
        Configuration, в которой, в частности, находится стандартная папка
        Components для хранения компонентов, используется идентификатор
        $flash. Если указанной в пути папки не существует, она будет
        автоматически создана (в данном случае папка Effects) -->
        <files>
               <file name="DustAndScratches.swc"
               destination="$flash/Components/Effects" />
        </files>
</macromedia-extension>
```

Создав установочный файл, его необходимо сохранить с расширением mxi в том же каталоге, куда был помещен SWC-архив компонента. Теперь остается только упаковать расширение. Для этого нужно вызвать Extensions Manager и в главном меню **File** выполнить команду **Package Extension**. В появившемся вслед за этим окне нужно указать путь к установочному MXI-файлу. Затем нужно задать имя файла расширения и каталог, в котором оно должно быть сохранено. Следует иметь в виду, что имя MXP-файла не должно содержать пробелов. Полученный в результате произведенных действий MXP-файл может использоваться для распространения компонента.

В результате установки созданного нами расширения DustAndScratches.mxp в панели **Components** будет создана новая категория с названием Effects, в которую будет помещен компонент `DustAndScratches`. При этом соответствующий SWC-архив будет сохранен в стандартном каталоге Flash, предназначенном для хранения компонентов.

Глава 28



Обработка исключительных ситуаций. Отладка сценариев. Тестирование проекта

Никогда не выявляйте в программе ошибки, если вы не знаете, что с ними дальше делать.

Руководство по системному программированию Штейнбаха

Обработка исключительных ситуаций

Исключительной ситуацией или просто *исключением* (exception) в программировании называется ошибка или нештатная ситуация, возникающая в ходе выполнения программы. Появление такой ошибки называется возбуждением исключительной ситуации. Исключительная ситуация возбуждается в результате наступления связанного с ней события. Таким событием, например, может являться передача функции неверного количества параметров, получение недопустимого для использования в дальнейших вычислениях значения и т. д. Для повышения устойчивости программы и обеспечения ее бесперебойной работы необходимо предусмотреть механизм обнаружения и обработки типовых исключительных ситуаций. Это становится особенно важно при создании больших и сложных систем. Обработка исключительной ситуации осуществляется с помощью специального программного модуля, называемого *обработчиком исключительной ситуации*. Возбуждение исключительной ситуации и ее обработка, как правило, выполняются разными программными модулями.

В наиболее простых случаях (и, в частности, в примерах, приведенных ранее на страницах этой книги) для обнаружения и устранения возможных ошибок можно использовать стандартную проверку определенных критериев с помощью условного предложения `if` и в зависимости от результата инициировать выполнение тех или иных действий. Такой подход демонстрируется в следующем примере. Здесь приводится функция `divide()`, вычисляющая частное от деления двух чисел. При работе этой функции могут возникнуть три исключительные ситуации. Первая связана с передачей функции неверного количества параметров. Вторая связана с передачей параметров нечислового типа или значения `NaN`. И, наконец, третья связана с передачей функции в качестве второго параметра значения 0. В теле функции выполняется проверка, позволяющая обнаружить появление ис-

ключительной ситуации. В случае ее обнаружения инициируется вызов функции, осуществляющей обработку данного исключения, а сама функция возвращает значение `null` и прекращает работу.

```
function divide(numerator, denominator):Number {
        // Проверка числа параметров (должно быть два)
        if (arguments.length != 2){
                paramNumError();
                return null;
        }
        // Проверка типа параметров (должны быть числом)
        var s = arguments[0] + arguments[1];
        if (!(typeof s == "number" && !isNaN(s))){
                typeMismatchError();
                return null;
        }
        // Проверка деления на ноль
        if (denominator == 0) {
                divideError();
                return null;
        }
        // Если ошибки не обнаружено, возвращаем результат
        return numerator / denominator;
}
// Функции для обработки обнаруженных ошибок
function paramNumError(){
        trace("**Ошибка!** Неверное число параметров");
}
function typeMismatchError(){
        trace("**Ошибка!** Неверный тип параметра.");
}
function divideError() {
        trace("**Ошибка!** Деление на ноль.");
}
```

Рассмотренный вариант обработки исключений часто применяется на практике при создании небольших проектов. Однако при создании более сложных и комплексных программ он оказывается недостаточно удобным и эффективным. В ActionScript имеются специализированные программные средства, предназначенные для возбуждения и обработки исключительных ситуаций. К их числу относятся: конструкция `try...catch...finally`, предложение `throw` и встроенный класс `Error`. Эти инструменты в сочетании позволяют реализовать гибкую систему контроля исключительных

ситуаций. Вначале рассмотрим общий вид конструкции `try...catch...finally` и опишем принцип ее работы.

```
try {
блок кода, который может возбуждать исключительную ситуацию
}
catch (формальный параметр : тип){
блок кода, выполняющий обработку исключительной ситуации
}
...
catch(формальный параметр : тип){
блок кода, выполняющий обработку исключительной ситуации
}
finally{
блок кода, который выполняется всегда
}
```

Конструкция состоит из одного предложения `try`, произвольного количества предложений `catch` и одного предложения `finally`. В этой конструкции обязательно должно присутствовать предложение `try` и либо хотя бы одно предложение `catch`, либо предложение `finally`, или, как это показано ранее, и то и другое. Приведем обзорное описание каждого их этих предложений.

Предложение `try` содержит блок кода, при выполнении которого возможно возбуждение исключительной ситуации. В блок этого предложения необходимо поместить программный код, выполнение которого может привести к возникновению ошибки. В обязанности этого кода входит проверка наличия факта ошибки (нештатной ситуации) и возбуждение соответствующей исключительной ситуации при его обнаружении. Для того чтобы возбудить исключительную ситуацию, используется предложение `throw`. Этому предложению передается параметр, чаще всего представляющий собой объект, соответствующий возбуждаемой исключительной ситуации. При возбуждении исключительной ситуации все инструкции, расположенные в блоке предложения `try` после возбудившего данную исключительную ситуацию предложения `throw`, будут проигнорированы, а управление будет передано соответствующему обработчику `catch`.

Предложение `catch` исполняет роль обработчика исключительной ситуации. В блок этого предложения необходимо поместить инструкции, которые должны быть выполнены при обнаружении соответствующей исключительной ситуации. Предложение `catch` имеет формальный параметр, который может быть строго типизирован. При возбуждении исключительной ситуации фактический параметр предложения `throw` передается тому обработчику `catch`, тип формального параметра которого соответствует типу параметра предложения `throw`, возбудившего исключительную ситуацию. Это позволяет различным образом реагировать на разные исключения.

Инструкции, помещенные в блок предложения `finally`, будут выполнены независимо от того, возбудило ли предложение `try` исключительную ситуацию или нет, и

была ли возбужденная ситуация обработана предложением catch. В связи с этим в блок предложения finally следует помещать инструкции, которые должны выполняться в любом случае.

Семантика конструкции try...catch...finally заключается в следующем: если выполнение кода предложения try не привело к возбуждению исключительной ситуации, то после выполнения всех инструкций этого предложения управление переходит к предложению finally. В этом случае вызова обработчика исключительной ситуации не происходит, и все инструкции предложений catch игнорируются. Если при выполнении кода предложения try происходит выполнение предложения throw, то есть возбуждается исключительная ситуация, она передается обработчику catch, тип формального параметра которого совпадает с типом параметра предложения throw, возбудившего эту исключительную ситуацию. При возбуждении исключительной ситуации происходит выход из предложения try и все инструкции, расположенные в блоке этого предложения, после возбудившего данную исключительную ситуацию предложения throw, игнорируются. Если тип формального параметра одного из обработчиков catch совпадает с типом объекта, передаваемого предложением throw, выполняется код данного обработчика. При наличии нескольких обработчиков с одинаковыми типами формальных параметров исключительная ситуация перехватывается первым (ближайшим к предложению try) из них. После перехвата исключительной ситуации и выполнения кода соответствующего обработчика все остальные обработчики *catch* игнорируются, и управление переходит к предложению finally. Если в конструкции отсутствует предложение *catch* или если тип формального параметра обработчика *catch* не соответствует типу объекта, передаваемого предложением throw, исключительная ситуация не перехватывается, то есть ее возбуждение не приводит к выполнению каких-либо специфических действий. При этом сразу же после выполнения предложения try управление передается предложению finally.

Механизм работы конструкции try...catch...finally представлен на рис. 28.1.

Перед тем как перейти к рассмотрению практических примеров, необходимо более подробно остановиться на механизме возбуждения исключительной ситуации. Как уже говорилось, исключительная ситуация возбуждается с помощью предложения throw, имеющего следующий синтаксис:

```
throw исключение
```

Здесь *исключение* — это выражение или объект, который передается в качестве параметра обработчику исключительной ситуации catch. Если в конструкции try...catch...finally имеется только один обработчик catch и его формальный параметр не типизирован, то при возбуждении исключительной ситуации этому обработчику передается параметр предложения throw и выполняется блок его кода. Если формальный параметр обработчика строго типизирован, то перехват исключительной ситуации, возбужденной предложением throw, происходит, только если тип формального параметра обработчика совпадает с типом параметра предложения throw. Характерной особенностью в работе предложения throw является то, что если возбужденная им исключительная ситуация не перехватывается ни одним из обработчиков catch, при тестировании документа в панель **Output** будет выведено строковое представление значения *исключение*.

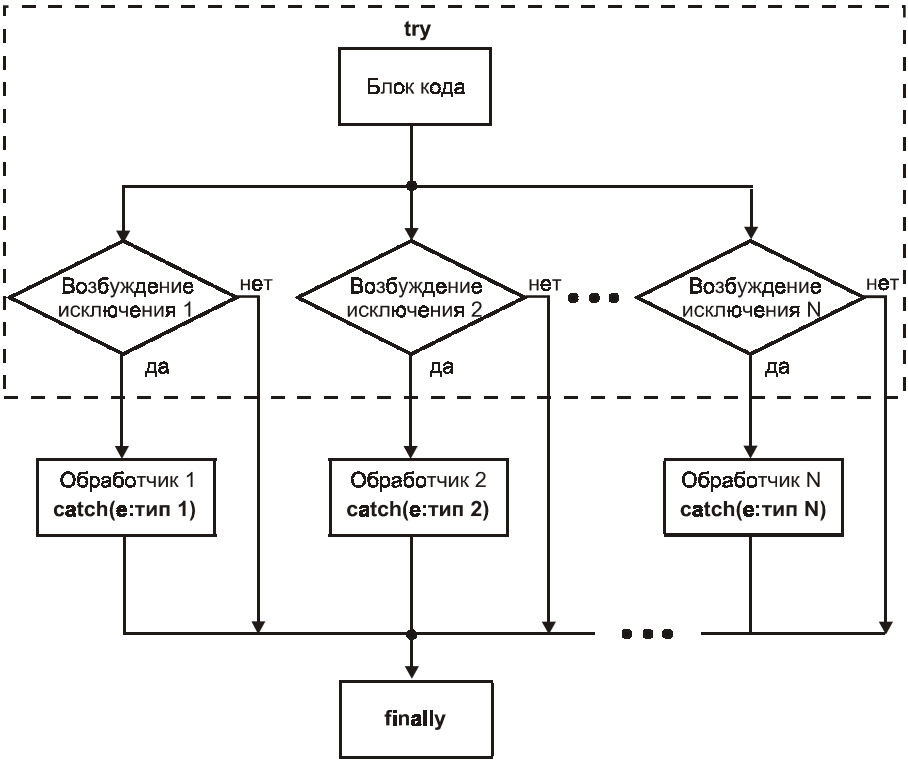


Рис. 28.1. Механизм обработки исключительных ситуаций с помощью конструкции `try...catch...finally`

Как правило, в качестве выражения предложения `throw` выступает экземпляр встроенного класса `Error` или его подкласса. Класс `Error` применяется для создания объектов, хранящих информацию о возбужденной исключительной ситуации. Объекты класса `Error` обладают двумя свойствами:

- `message` — значением данного свойства является строка, содержащая описание исключительной ситуации. По умолчанию значением этого свойства для объектов класса `Error` является строка `"Error"`. При создании объекта `Error` можно задать произвольное значение свойства `message`, которое будет лучше отражать специфику исключения, соответствующего данному объекту;
- `name` — значением данного свойства является строка, содержащая имя конструктора класса, породившего объект. По умолчанию значением этого свойства для объектов класса `Error` является строка `"Error"`.

Создание объекта `Error` выполняется с применением следующего синтаксиса:

```
var e:Error = new Error(message)
```

Параметр *message*, передаваемый конструктору `Error()`, представляет собой строку, которая присваивается свойству *message* создаваемого объекта. Этот параметр не

является обязательным. В случае его отсутствия свойство *message* созданного объекта Error будет принимать значение по умолчанию. Значение свойства *message* также можно задать уже после создания экземпляра.

Чаще всего на практике исключительная ситуация возбуждается следующим образом:

```
throw new Error ("Содержание обнаруженной ошибки");
```

При возбуждении исключительной ситуации объект Error передается обработчику catch, формальный параметр которого имеет тип Error. Информация, отражающая характер обнаруженного исключения, хранящаяся в свойствах message и name передаваемого объекта, может быть использована инструкциями, расположенными в теле обработчика.

В классе Error имеется единственный метод toString(). Этот метод возвращает строку, содержащую значение свойства message объекта Error. Вызов этого метода дает тот же результат, что и обращение к свойству message:

```
var e:Error = new Error("Исключение");
trace(e.toString() == e.message); // Выводит true
```

Рассмотрим простой пример обработки исключительной ситуации. Допустим, нам требуется вывести в панель **Output** случайное четное число в диапазоне от 2 до 100. Для получения случайного числа мы воспользуемся стандартным генератором случайных чисел Math.random(). Учитывая требование четности, появление нечетного числа должно привести к возбуждению исключительной ситуации. В этом случае нам потребуется заменить полученное нечетное число на ближайшее к нему четное. В листинге 28.1 приводится программный код, реализующий поставленную задачу.

Листинг 28.1. Возбуждение исключительной ситуации из блока предложения try

```
// Выполнение блока кода, являющегося потенциальным источником ошибки
try {
        // Сохраняем случайное целое число от 2 до 100 в переменной.
        var rnd:Number = Math.round(Math.random()*98) + 2;
        // Если это число оказалось нечетным, возбуждаем исключение
        if (rnd % 2 != 0) {
                throw new Error("**Ошибка!** Нечетное число.");
        }
}
// Обработка исключительной ситуации
catch (e:Error) {
        trace(e.message); // Выводим сообщение об обнаруженном исключении
        rnd++; // Увеличиваем rnd на единицу (чтобы получить четное число)
}
// Действия, выполняемые независимо от обнаружения и обработки исключения
finally {
```

```
        trace(rnd); // Выводим требуемое число
        delete e; // Удаляем исключение, так как больше этот объект
                  // не нужен
        delete rnd; // Удаляем переменную rnd, так как она больше не нужна
}
```

Обратите внимание, что поскольку инструкции предложения `finally` выполняются независимо от того, была ли возбуждена и обработана исключительная ситуация или нет, сюда помещается предложение, осуществляющее вывод результата в панель **Output**. Вообще, предложение `finally`, как правило, используется для обработки результатов действий, реализованных в предложениях `try` и `catch` (например, удаление вспомогательных переменных и объектов).

Предложение `throw` не должно обязательно размещаться непосредственно в блоке предложения `try`. Исключительная ситуация может быть возбуждена из тела функции или метода, вызов которого помещен в блок предложения `try`. В качестве примера приведем уже рассмотренную в начале этой главы функцию для вычисления частного двух чисел. Теперь обработка исключительных ситуаций будет выполнена с применением конструкции `try...catch...finally`. Обратите внимание на то, что в этом примере (листинг 28.2) возбуждение исключительной ситуации выполняется в теле функции, а ее обработка — за пределами этой функции.

Листинг 28.2. Возбуждение исключительной ситуации из тела функции

```
function divide(numerator, denominator):Number {
        // Проверка числа параметров (должно быть два)
        if (arguments.length != 2){
                throw new Error("**Ошибка!** Неверное число параметров.");
        }
        // Проверка деления на ноль
        if (denominator == 0) {
                throw new Error("**Ошибка!** Деление на ноль.");
        }
        // Проверка типа параметров (должны быть числом)
        // Для того чтобы не проверять каждый параметр по отдельности,
        // сложим их и будем проверять сумму.
        var s = arguments[0] + arguments[1];
        if (!(typeof s == "number" && !isNaN(s))){
                throw new Error ("**Ошибка!** Неверный тип параметра.");
        }
        return numerator / denominator;
}
```

```
try {
        result = divide(2, 0); // попытка деления на 0
}
catch (e:Error) {
        trace(e);
        result = null;
}
finally {
        trace(result);
        delete e;
}
```

В результате выполнения данного кода в панель **Output** будет выведено следующее:

```
**Ошибка!** Деление на ноль.
null
```

В этом примере могут иметь место три различных исключительных ситуации. Отметим, что после выполнения предложения `throw` в теле функции `divide()` управление сразу же передается предложению `catch`, а все остальные инструкции в теле функции (включая последующие предложения `throw` и предложение `return`) игнорируются. В связи с этим одновременное возбуждение нескольких исключительных ситуаций оказывается невозможным. При обнаружении первой ошибки сразу же выполняется выход из предложения `try`. В нашем случае различные исключительные ситуации обрабатываются одним и тем же обработчиком `catch`. Такой подход, впрочем, оказывается приемлем далеко не всегда. Чаще необходимо различным образом реагировать на разные исключительные ситуации. Для этого можно применить один из следующих подходов.

- Можно создать единственный обработчик `catch` и в нем организовать проверку значения свойства `message` объекта, переданного обработчику предложением `throw`, возбудившим исключительную ситуацию. В зависимости от значения этого свойства следует выполнить те или иные действия в качестве реакции на обнаруженную ошибку.
- Можно создать несколько обработчиков `catch`, каждый из которых должен будет отвечать за перехват соответствующей исключительной ситуации. В этом случае при возбуждении исключительной ситуации выражение предложения `throw` должно иметь уникальный тип, соответствующий данному исключению, а перехватывающий это исключение обработчик `catch` должен быть снабжен формальным параметром того же типа.

В качестве иллюстрации применения первого подхода приведем пример, представляющий собой модификацию рассмотренного ранее примера. Допустим, нам требуется вывести в панель **Output** случайное четное число, входящее в полуинтервал [2; 40) или в полуинтервал (80; 100]. Исключительная ситуация может возникнуть в одном из трех случаев: если полученное число находится на отрезке [40, 80], если

полученное число равно нулю и если полученное число — нечетное. В приведенном далее примере (листинг 28.3) используется единственный обработчик `catch`, в котором на основании проверки значения свойства `message` выполняется обработка соответствующего исключения.

Листинг 28.3. Обработка нескольких исключений единственным обработчиком

```
try {
        var rnd:Number = Math.round(Math.random() * 100);
        if (rnd >= 40 && rnd <= 80) { // число лежит на отрезке [40, 80]
                throw new Error("Недопустимое значение");
        }
        if(rnd == 0 ){ // число равно нулю
                throw new Error ("Нулевое значение");
        }
        if (rnd % 2 != 0) { // число нечетное
                throw new Error("Нечетное число");
        }
} catch (e:Error) {
        switch (e.message) {
        case "Недопустимое значение" :
                // Присваиваем числу ближайшее допустимое значение
                rnd = (rnd-40 < 80-rnd) ? 38 : 82;
                break;
        case "Нулевое значение":
                rnd = 2;
        case "Нечетное число" :
                // Присваиваем числу ближайшее допустимое четное значение
                rnd = (rnd==39) ? 38 : ++rnd;
                break;
        }
} finally {
        trace(rnd);
        delete rnd;
        delete e;
}
```

При использовании индивидуального обработчика для каждой исключительной ситуации возбуждающее эту ситуацию предложение `throw` должно передавать обработчику значение, тип которого соответствует обнаруженному исключению. При ис-

пользовании данного подхода для каждой возможной исключительной ситуации создается класс, являющийся подклассом встроенного класса `Error`. Для возбуждения исключительной ситуации предложение `throw` передает обработчику экземпляр пользовательского класса, соответствующего обнаруженной исключительной ситуации. Таким образом, для обработки этой исключительной ситуации достаточно создать предложение `catch` и типизировать его формальный параметр, используя класс данной исключительной ситуации.

Рассмотрим пример, демонстрирующий применение пользовательских классов исключительных ситуаций. Допустим, нам требуется создать функцию, вычисляющую площадь прямоугольника, размеры которого передаются этой функции в качестве параметров. С работой такой функции могут быть связаны следующие исключительные ситуации: задано неверное число параметров, один или оба параметра равны нулю, один или оба параметра отрицательны. Чтобы не загромождать пример, мы не станем здесь рассматривать ситуацию нечисловых параметров, вместо этого строго типизируем их, чтобы обеспечить контроль соответствия типов на этапе компиляции.

Для того чтобы обеспечить бесперебойную работу нашей функции, мы должны реализовать механизм, позволяющий перехватывать и исправлять обнаруженные ошибки. Поскольку каждая из возможных ошибок должна обрабатываться особым образом, мы, в первую очередь, создадим классы для каждой из перечисленных исключительных ситуаций. Пользовательский класс исключительной ситуации должен являться подклассом встроенного класса `Error`. В этом классе следует, как минимум, переопределить два стандартных свойства: `message` и `name`. Ниже приводится определение класса `ZeroValueError`, представляющего исключительную ситуацию, связанную с нулевым значением параметра функции.

```
class ZeroValueError extends Error {
        public var message:String = "**Ошибка!** Значения параметров не↳
        могут быть равны нулю.";
        public var name:String = "ZeroValueError";
}
```

Пользовательский класс исключительной ситуации помимо объявления констант может также включать определения методов. Кроме того, в конструктор этого класса можно поместить инструкции, которые должны быть выполнены при создании экземпляра класса. Так, следующий класс `ParamNumError` соответствует исключительной ситуации, связанной с передачей функции неверного числа параметров. Конструктор этого класса принимает числовой параметр, указывающий правильное количество параметров, которое должно быть передано функции. Инициализация свойства `message` здесь выполняется в конструкторе, и его значение формируется на основании значения параметра, передаваемого конструктору при создании экземпляра класса.

```
class ParamNumError extends Error {
        public var name:String = "ParamNumError";
        function ParamNumError(n:Number) {
```

```
            message = "**Ошибка!** Неверное количество параметров.↵
            Требуется ровно " + n;
        }
}
```

Класс `ParamValueError` соответствует исключению, связанному с передачей функции отрицательных значений параметров.

```
class ParamValueError extends Error {
        public var message:String = "**Ошибка!** Значения параметров↵
        должны быть положительными.";
        public var name:String = "ParamValueError";
}
```

Ниже приведен упрощенный вариант функции `rectangleArea()`, предназначенной для вычисления площади прямоугольника. В этом примере конструкция `try...catch...finally` помещена непосредственно в тело функции. Для того чтобы сделать программный код более прозрачным, мы намеренно поместили здесь в обработчики `catch` одинаковый набор инструкций.

```
function rectangleArea(w:Number, h:Number):Number {
        try {
                if (arguments.length != 2) { //неверное число параметров
                        throw new ParamNumError(2);
                }
                if(w * h == 0){ // один или оба параметра равны нулю
                        throw new ZeroValueError();
                }
                if (w < 0 || h <0) { // один или оба параметра
                                     // отрицательны
                        throw new ParamValueError();
                }
                return w * h;
        }
        catch (e:ParamNumError) { //неверное число параметров
                trace(e.message);
                return null;
        }
        catch (e:ZeroValueError){// один или оба параметра равны нулю
                trace(e.message);
                return null;
        }
```

```
        catch (e:ParamValueError) {// один или оба параметра отрицательны
                trace(e.message);
                return null
        }
        finally {
                trace("Окончание работы функции");
        }
}
```

Отметим, что инструкции предложения `finally` выполняются даже после выполнения предложения `return`. Это происходит независимо от того, была ли возбуждена и перехвачена исключительная ситуация или нет.

В приведенном варианте функции `rectangleArea()` наступление исключительной ситуации сопровождается прекращением работы функции и возвратом значения `null`. Такое поведение может оказаться неприемлемым на практике, поскольку появление ошибки приведет к невозможности выполнять дальнейшие вычисления, используя значение, возвращенное функцией `rectangleArea()`. Иногда обработчик исключительной ситуации должен не просто констатировать обнаружение ошибки, но и по возможности устранять ее. Попробуем модифицировать приведенную выше функцию `rectangleArea()`, так чтобы при обнаружении исключительной ситуации она автоматически ликвидировалась. Очевидно, что для исправления возможных исключительных ситуаций, свойственных данной функции, не существует готовых способов. Устранение той или иной исключительной ситуации осуществляется исходя из требований, диктуемых постановкой общей задачи. Так, например, при передаче функции нулевого значения можно либо прекратить ее работу, возвратив значение `null` или `undefined`, либо скорректировать значение переданного параметра, присвоив ему значение по умолчанию и повторно вызвать функцию. В нашем учебном случае для исправления обнаруженных ошибок примем следующие соглашения.

- Если функции передано неверное количество параметров, то:
  - при отсутствии параметров повторно вызываем функцию, передавая ей в качестве параметров единицы;
  - при одном параметре повторно вызываем функцию, передавая ей в качестве отсутствующего параметра единицу;
  - при трех и более параметрах повторно вызываем функцию, передавая ей первые два из исходных параметров.
- Если один или оба параметра равны нулю, повторно вызываем функцию, передавая ей вместо нулевого параметра значение 1 (ближайшее целое положительное число).
- Если один или оба параметра отрицательны, повторно вызываем функцию, передавая ей абсолютные значения исходных параметров.

Для вызова функции из самой себя мы воспользуемся свойством `callee` объекта `arguments` *(см. главу 8)*. Поскольку рассматриваемый способ обработки исключи-

тельных ситуаций предполагает рекурсивный вызов функции, это дает возможность на протяжении работы функции последовательно обнаружить и ликвидировать несколько исключительных ситуаций. Если при вызове функции в предложении `try` возбуждается исключительная ситуация, она передается соответствующему обработчику, который, исправив ошибку, рекурсивно вызывает функцию. Если при этом происходит возбуждение новой исключительной ситуации, она передается другому обработчику. Таким образом, функция будет рекурсивно вызываться до тех пор, пока все обнаруженные ошибки не будут исправлены. В соответствии с ограничением, действующим в проигрывателе Flash Player, количество уровней рекурсии не может превышать 256. Этого количества оказывается вполне достаточно для решения рассматриваемой задачи (в нашем случае достаточно всего четырех уровней). Для наглядности мы ввели в функцию переменную `count`, выводящую в панель **Output** количество ее вызовов.

Окончательный вид функции `rectangleArea()` представлен в листинге 28.4.

Листинг 28.4. Рекурсивная обработка нескольких исключительных ситуаций разными обработчиками

```
function rectangleArea(w:Number, h:Number):Number {
        count = 0; // Счетчик вызовов функции
        try {
                if (arguments.length != 2) { //неверное число параметров
                        throw new ParamNumError(2);
                }
                if(w * h == 0){ // один или оба параметра равны нулю
                        throw new ZeroValueError();
                }
                if (w < 0 || h <0) { // один или оба параметра отрицательны
                        throw new ParamValueError();
                }
                return w * h;
        }
        catch (e:ParamNumError) { //неверное число параметров
                trace(e.message);
                if(arguments.length == 0){// Если параметры отсутствуют
                //Вызываем функцию, передав ей две единицы
                return arguments.callee(1, 1);
                }
                // Если указан один параметр
                else if (arguments.length == 1){
```

```
            // вызываем функцию, передав ей единицу в качестве второго
            // параметра
            return arguments.callee(arguments[0], 1);
            }
            else { // Если передано больше двух параметров
            // Вызываем функцию только с двумя первыми параметрами
            return arguments.callee(arguments[0], arguments[1]);
            }
        }
        catch (e:ZeroValueError){// один или оба параметра равны нулю
            trace(e.message);
            if(arguments[0] == 0) arguments[0] = 1;
            if(arguments[1] == 0) arguments[1] = 1;
            // Рекурсивно вызываем функцию, передавая ей исправленные
            // параметры в виде массива
            return arguments.callee.apply(null, arguments);
        }
        catch (e:ParamValueError) {// один или оба параметра отрицательны
            trace(e.message);
            // Вызываем функцию, передавая ей значения параметров по
            // модулю
            return arguments.callee(Math.abs(arguments[0]),↵
            Math.abs(arguments[1]));
        }
        finally {
            count++ // Модифицируем счетчик при каждом вызове функции
            trace(count+"-й вызов функции");
        }
}
```

Попробуем вывести в панель **Output** результат работы функции `rectangleArea()`, намеренно вызвав ее с тремя ошибками:

```
trace("Площадь = " + rectangleArea(-3, 0, 7)); // Три ошибки
```

В панель **Output** будет выведено следующее:

```
**Ошибка!** Неверное количество параметров. Требуется ровно 2
**Ошибка!** Значения параметров не могут быть равны нулю.
**Ошибка!** Значения параметров должны быть положительными.
1-й вызов функции
2-й вызов функции
```

```
3-й вызов функции
4-й вызов функции
Площадь = 3
```

Тестирование проекта

В процессе разработки проекта постоянно возникает необходимость тестировать его на предмет соответствия поставленным задачам. Среда тестирования, являющаяся частью среды разработки Flash, позволяет оценить функционирование конечного фильма и всех его элементов, а также получить представление о распределении данных по кадрам и о том, как будет выглядеть процесс работы данного фильма при его загрузке по сети. Тестирование позволяет "вживую" проанализировать работу сценариев, оценить эффективность программного кода, выявить ошибки и проблемные места. Перед публикацией проекта необходимо тщательно протестировать его и убедиться в полной работоспособности и отсутствии ошибок и неполадок.

При переходе в среду тестирования Flash автоматически генерирует файл SWF, используя текущие настройки публикации, заданные в окне **Publish Settings**. Для перехода в среду тестирования можно выполнить одну из двух команд главного меню рабочей среды **Control**:

- **Test Movie** <Ctrl>+<Enter> — тестирование фильма. В результате выполнения данной команды весь фильм экспортируется в файл SWF, который открывается в среде тестирования;
- **Debug Movie** — данная команда используется для отладки сценариев. Ее выполнение приводит к переходу в среду тестирования и запуску отладчика (**Debugger**);
- **Test Scene** <Ctrl>+<Alt>+<Enter> — тестирование сцены. В результате выполнения данной команды текущая сцена экспортируется в файл SWF, который открывается в среде тестирования. Если эта команда выполняется из среды редактирования символа, то будет выполнено тестирование только данного символа, то есть в результирующий SWF-файл будет включено только содержимое монтажной линейки этого символа.

Окно среды тестирования в развернутом виде всегда несколько больше, чем рабочая область, размеры которой задаются в рабочей среде Flash. В связи с этим в среде тестирования может быть видна часть вспомогательной области с расположенными на ней объектами. Для того чтобы привести окно среды тестирования к размерам рабочей области, можно щелкнуть на стандартной кнопке **Restore**  в правом верхнем углу окна.

Управление отображением и воспроизведением фильма

Для изменения масштаба просмотра фильма в среде тестирования используются команды главного меню **View**:

- **Zoom In** — увеличение масштаба просмотра;

- **Zoom Out** — уменьшение масштаба просмотра;
- **Magnification** — выбор одного из стандартных значений масштаба.

Установить требуемый масштаб просмотра можно также с помощью контекстного меню, для вызова которого необходимо щелкнуть на окне фильма правой кнопкой мыши.

Поскольку Flash использует технологию сглаживания (antialiasing) по умолчанию, это может вызвать снижение скорости воспроизведения, если ресурсов процессора недостаточно для прорисовки кадров с требуемой скоростью. Такая ситуация может иметь место при одновременном выполнении нескольких сложных анимационных процессов с применением автоматической анимации. Чтобы задать соотношение между качеством прорисовки графики и скоростью воспроизведения, можно воспользоваться командой главного меню **View>Quality** или соответствующей командой контекстного меню среды тестирования. Данная команда содержит три значения, определяющие приоритет качества или быстродействия: **Low** — отключение сглаживания, максимальная скорость; **Medium** — компромисс между качеством и скоростью, **High** — сглаживание включено, скорость прорисовки может быть снижена.

Управление воспроизведением фильма в среде тестирования осуществляется с помощью команд меню **Control**, аналогичных соответствующим командам рабочей среды фильма:

- **Play/Stop** — включение/остановка воспроизведения, начиная с текущего кадра;
- **Rewind** — переход к первому кадру монтажной линейки;
- **Loop —** зациклить воспроизведение;
- **Step Forward One Frame** — переход на следующий кадр и остановка воспроизведения;
- **Step Backward One Frame** — переход на предыдущий кадр и остановка воспроизведения.

Данные команды также могут быть выполнены с помощью контекстного меню среды тестирования.

Как и в рабочей среде, в среде тестирования имеются свои горячие клавиши (клавиатурные эквиваленты команд). Если в проекте используется программный контроль клавиатуры, применение горячих клавиш в среде тестирования может повлиять на работу фильма, поскольку горячие клавиши имеют приоритет над программным кодом ActionScript. Для того чтобы исключить воздействие горячих клавиш среды тестирования, их можно отключить с помощью команды главного меню **Control>Disable Keyboard Shortcuts**.

Эмуляция загрузки

При загрузке по сети Flash-фильм представляет собой поток данных (data streaming), организованный таким образом, что первыми загружаются те данные, которые должны отображаться в первую очередь. Вне зависимости от скорости соединения проигрыватель Flash Player пытается поддержать заданную в рабочей среде скорость воспроизведения. Если скорость передачи данных недостаточно высока, то для того чтобы обеспечить своевременную загрузку содержимого кадров, на соответствующем

участке воспроизведение будет приостановлено до тех пор, пока не загрузится информация, необходимая для отображения следующего кадра. Это приводит к нарушению ритма фильма и синхронизации звукового сопровождения, разрушая впечатление целостности. Если размер кадра меньше, чем объем информации, которая может быть загружена при заданной скорости за время экспонирования кадра, то в процессе его воспроизведения будет происходить загрузка следующих кадров. Элементы, которые многократно используются в фильме, загружаются только в кадре их первого вхождения в фильм. В дальнейшем для их повторного отображения загрузка не должна выполняться повторно. Программный код классов, используемых в фильме, выполняется перед выводом остального содержимого содержащего их кадра. Таким образом, реальная картина загрузки проекта по сети будет существенно отличаться от запуска Flash-фильма, размещенного локально на компьютере пользователя.

Среда тестирования предоставляет возможность имитации процесса загрузки по сети с заданной скоростью. Используя эту возможность, можно смоделировать процесс загрузки с использованием различных модемов, по цифровым линиям или по локальной сети. Для того чтобы включить режим эмуляции загрузки, необходимо выполнить команду главного меню **View>Simulate Download** или воспользоваться ее клавиатурным эквивалентом <Ctrl>+<Enter>.

Скорость загрузки зависит от скорости соединения. Команда **View>Download Settings** позволяет задать одно из предустановленных значений скорости передачи данных, на основании которого будет реализована имитация загрузки фильма. При моделировании процесса загрузки учитывается не номинальная скорость модема, а более реальная величина, имеющая место на практике. Так, например, если скорость модема составляет 56 Кбит/с, то теоретически он может передавать информацию со скоростью около 7 Кбайт/с, однако на практике скорость будет иметь величину порядка 4,7 Кбайт/с или 4800 байт/с. При эмуляции загрузки также учитывается компрессия SWF-файла (если данная опция была включена при публикации), которая приводит к уменьшению размера конечного файла и, соответственно, к увеличению скорости его передачи. Выбор команды **Customize...** приводит к появлению диалогового окна настройки скорости передачи данных **Custom Download Settings**. Здесь в колонке **Menu text** можно указать обозначение требуемой скорости, причем эти данные не влияют на результат, а используются для вывода в меню **View>Download Settings**. Колонка **Bit rate** позволяет задать величину скорости соединения, на основании которой Flash будет эмулировать процесс загрузки.

Окно *Bandwidth Profiler*

Важнейшим инструментом среды тестирования является окно **Bandwidth Profiler**, позволяющее произвести анализ процесса загрузки фильма. Для того чтобы открыть данное окно, необходимо выполнить команду главного меню **View>Bandwidth Profiler** или воспользоваться ее клавиатурным эквивалентом <Ctrl>+<B>. Окно **Bandwidth Profiler** состоит из двух разделов: слева приводятся параметры загрузки фильма, справа находится один из двух видов графиков, иллюстрирующий процесс загрузки или размер кадров фильма (рис. 28.2).

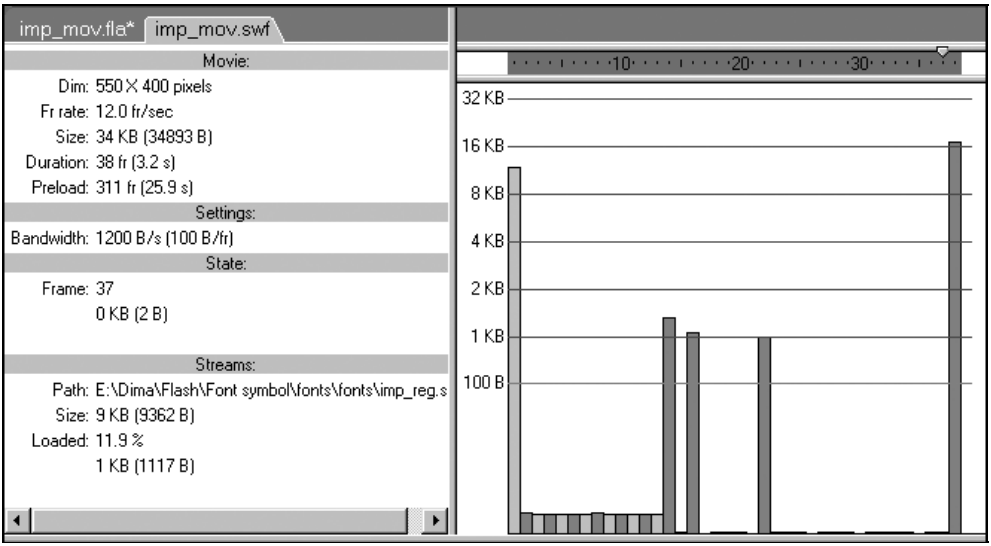


Рис. 28.2. Окно **Bandwidth Profiler**

Информационная панель

Информационная панель в левой части окна **Bandwidth Profiler** состоит из нескольких разделов и содержит следующие сведения.

- **Movie** — параметры фильма:
  - **Dim** — размер фильма, определяемый размерами рабочей области;
  - **Fr rate** — скорость воспроизведения монтажной линейки фильма;
  - **Size** — размер конечного файла SWF в килобайтах и байтах;
  - **Duration** — длительность фильма в кадрах и секундах;
  - **Preload** — суммарное время ожидания загрузки данных на протяжении всего фильма. Если скорость передачи данных при заданной скорости соединения ниже, чем требуется для поддержания требуемой скорости воспроизведения, то на соответствующих кадрах воспроизведение приостанавливается, ожидая загрузки. Данный параметр указывает суммарную длительность всех таких задержек и измеряется в секундах и в кадрах (исходя из используемого значения скорости воспроизведения монтажной линейки). Если скорость передачи данных достаточна для обеспечения требуемой скорости воспроизведения на протяжении всего фильма, то значение данного параметра равно нулю.
- **Settings** — заданные настройки скорости соединения, для которой выполняется тестирование. Первая цифра указывает скорость в байтах в секунду, вторая (в скобках) — максимальный размер каждого кадра фильма, при котором обеспечивается требуемая скорость воспроизведения и не возникают остановки при загрузке с заданной скоростью.

- **State** — информация о размере кадров и состоянии загрузки:
  - **Frame** — номер кадра и его размер;
  - **Loaded** — эти данные отображаются только в режиме эмуляции загрузки **Simulate Download** на протяжении загрузки фильма и указывают, какой объем фильма (в процентах и килобайтах) загрузился на текущий момент. После того как фильм загружен, эта информация не видна.
- **Streams** — этот раздел появляется только в режиме эмуляции загрузки **Simulate Download** в том случае, если в процессе воспроизведения в текущий фильм подгружаются внешние данные (например, растровое изображение, звуковой файл, внешний ролик SWF, элементы общей библиотеки и т. д.):
  - **Path** — путь к загружаемому файлу;
  - **Size** — размер загружаемого файла;
  - **Loaded** — состояние загрузки внешнего файла (в процентах и килобайтах).

Работа с графиками окна *Bandwidth Profiler*

В правой части окна **Bandwidth Profiler** расположены шкала с нумерацией кадров и один из графиков: **Streaming Graph** (потоковый график) или **Frame by Frame Graph** (покадровый график). В верхней части окна находится шкала с нумерацией кадров. Треугольный бегунок, находящийся на этой шкале, указывает на текущее положение воспроизводящей головки. Его можно перемещать так же, как и указатель текущего кадра в рабочей среде фильма, таким образом переходя из одного кадра в другой. По вертикальной оси (расположенной справа) откладывается размер кадров. Красная горизонтальная линия показывает максимальный размер кадра, обеспечивающий нормальную передачу потока данных без прерываний воспроизведения. Это значение рассчитывается как отношение скорости передачи данных к скорости воспроизведения монтажной линейки. Так, если заданная в настройках **Custom Download Settings** скорость соединения составляет 1200 байт/с, то при скорости воспроизведения монтажной линейки 12 кадр/с объем каждого кадра не должен превышать 100 байт. Здесь, однако, существует один нюанс, который необходимо иметь в виду. Дело в том, что если объем предыдущих кадров меньше критического объема и число таких кадров достаточно велико, то в процессе их воспроизведения следующий за ними кадр, превышающий критический размер, может успеть загрузиться и не вызовет паузы.

При активизации режима эмуляции загрузки **Simulate Download** поток данных графически отображается в виде растущей зеленой полосы загрузки на шкале в верхней части окна **Bandwidth Profiler**. Если полоса загрузки опережает указатель текущего кадра, это означает, что скорость передачи информации достаточна для обеспечения заданной скорости воспроизведения; в противном случае воспроизведение может быть приостановлено.

Потоковый график (Streaming Graph)

Потоковый график является визуальным представлением процесса загрузки кадров фильма и позволяет определить те участки фильма, которые могут вызвать задержку при загрузке. Для того чтобы отобразить этот график в правой части окна **Bandwidth**

Profiler, необходимо выполнить команду главного меню **View>Streaming Graph** (<Ctrl>+<G>). Потоковый график представлен на рис. 28.3. Вертикальные светлые и темные серые полосы (или блоки) представляют собой кадры фильма. Каждый из них может быть выделен по отдельности; при этом в разделе **State** информационной панели будет выведен номер данного кадра и его размер. Теоретически загрузка будет выполняться без остановок и прерываний воспроизведения в том случае, если время загрузки содержимого кадра равно (или меньше) времени или длительности его экспонирования, которая определяется исходя из заданной скорости воспроизведения (frame rate). Так, если скорость воспроизведения составляет 12 кадр/с, то на загрузку каждого кадра отводится 1/12 секунды. Максимальный размер кадра, который может быть загружен за время экспонирования при заданной скорости передачи данных, будем называть в дальнейшем *критическим размером*. Критический размер кадра на графике указывается красной горизонтальной полосой.

Если размер кадра меньше критического размера, отмеченного красной полосой, то это означает, что данный кадр загружается за более короткий интервал времени, чем длительность экспонирования (задаваемая скоростью воспроизведения). Таким образом, он не только не вызывает паузы при загрузке, но и предоставляет дополнительное время для загрузки следующих после него кадров. На графике блоки, соответствующие таким кадрам, располагаются друг над другом в одном или нескольких расположенных рядом столбцах (рис. 28.3, *а*).

Если размер кадра в точности равен критическому размеру, то такой кадр будет загружаться в течение всего отведенного ему времени, не вызывая паузы при загрузке, но и не предоставляя дополнительное время для загрузки других кадров. Такой кадр отображается столбцом, высота которого равна критическому размеру кадра (рис. 28.3, *б*).

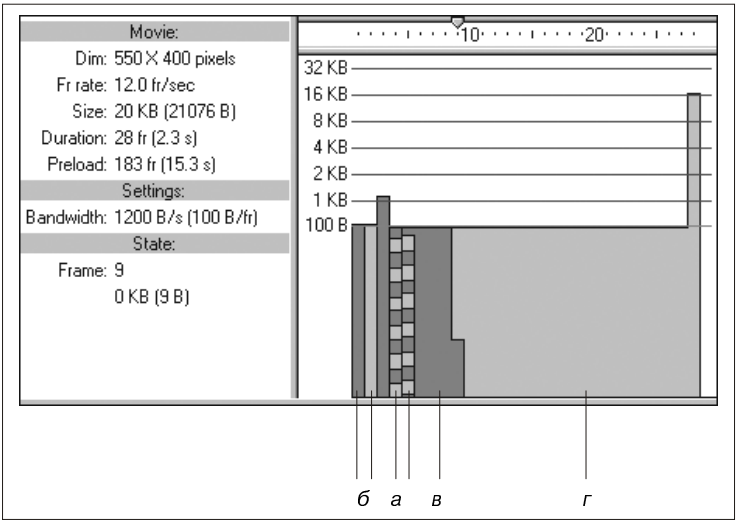


Рис. 28.3. Потоковый график

Если размер кадра превышает критический размер, то для его загрузки требуется больше времени, чем составляет длительность экспонирования кадра. Соответственно, такой кадр либо вызовет приостановку воспроизведения, либо использует дополнительное время, "сэкономленное" предшествующими кадрами небольшого размера (а если накопленный запас времени не сможет компенсировать размер кадра, то произойдет и одно, и другое одновременно). Такой кадр отображается в виде широкого блока. Если часть этого блока выходит за пределы красной линии, это означает, что в данном месте воспроизведение будет прерываться, ожидая загрузки данных (рис. 28.3, *в*). Если весь блок укладывается в допустимый диапазон, не пересекая красную полосу, то он не будет вызывать паузы при воспроизведении (рис. 28.3, *г*).

На практике, как правило, не удается добиться плавной загрузки фильма без приостановок воспроизведения. Для того чтобы избежать связанных с этим негативных последствий, создается предзагрузчик.

Покадровый график (Frame by Frame Graph)

Для того чтобы отобразить этот график в правой части окна **Bandwidth Profiler**, необходимо выполнить команду главного меню **View>Frame by Frame Graph** (<Ctrl>+<F>). Покадровый график представляет собой столбчатую диаграмму, высота каждого столбца которой соответствует размеру кадра, находящегося в текущей временной позиции (см. рис. 28.2). Участки покадрового графика с отсутствующими столбцами соответствуют кадрам, в которых не происходит никаких изменений. Для получения информации о размере каждого кадра можно щелкнуть на соответствующем столбце, при этом в разделе **State** информационной панели будет выведен номер данного кадра и его размер.

Отчет о размерах элементов фильма

Еще одна возможность, позволяющая проанализировать эффективность организации фильма, заключается в создании отчета о размерах всех используемых в фильме элементов. Данный отчет представляет собой текстовый документ (файл с расширением txt), который генерируется автоматически и дописывается в каталог рядом с результирующим SWF-документом. Отчет содержит информацию о размерах фильма по кадрам и по сценам, а также о размерах используемых символов, шрифтов, сценариев ActionScript и импортированных объектов. Анализ отчета позволяет увидеть, какой вклад в размер каждой сцены и символа вносят векторные объекты, текст и сценарии. Кроме того, здесь приводится информация обо всех используемых в фильме шрифтах и указываются внедряемые символы. Отчет также отображает информацию о местоположении всех элементов программного кода фильма и о параметрах компрессии импортированных объектов.

Для того чтобы создать отчет, необходимо в диалоговом окне настройки параметров публикации **File>Publish Settings...** на закладке **Flash** установить флажок **Generate Size Report**. В результате всякий раз при публикации или тестировании проекта отчет будет создаваться и автоматически записываться на диске. Кроме того, при тестировании проекта в среде тестирования отчет будет выводиться в диалоговом окне **Output**.

Анализ отчета зачастую позволяет выявить проблемные места, которые могут вызвать задержку загрузки фильма. Используя информацию, предоставленную отчетом, можно попытаться оптимизировать фильм за счет более рациональной организации его структуры (например, за счет включения кода классов в заданный кадр, разбиения исходного фильма на несколько самостоятельных файлов, применения общих библиотек и т. д.).

Средства отладки сценариев

Отладкой называется комплекс мероприятий, направленных на выявление и устранение ошибок в программе. Среда разработки Flash включает несколько стандартных средств для выполнения отладки. К их числу относятся:

- функция `trace();`
- панель **Output**;
- отладчик **Debugger**.

Функция `trace()` была описана в *главе 3*. Этот инструмент уже хорошо знаком читателю, поскольку примеры его практического применения многократно демонстрировались в рассмотренных на страницах этой книги листингах. Поэтому в данном разделе мы опишем два других инструмента: панель **Output** и панель **Debugger**.

Панель *Output*

Поскольку оценить работу сценария, созданного в рабочей среде, мы можем только после его компиляции, панель **Output,** в основном, используется в среде тестирования документа. Для того чтобы перейти в среду тестирования, можно выполнить команду главного меню **Control>Test Movie** (<Ctrl>+<Enter>).

Как и любая рабочая панель Flash, панель **Output** имеет собственное контекстное меню. В следующем списке содержится список входящих в его состав команд и описание их назначения:

- **Word Wrap** — режим переноса строк. Этот режим позволяет отображать все содержимое выводимой строки в окне панели без необходимости использования горизонтальной полосы прокрутки;
- **Copy** — при отсутствии выделения данная команда копирует в буфер обмена все текущее содержимое панели **Output**. Для того чтобы скопировать в буфер требуемый фрагмент, необходимо выделить его и выполнить данную команду;
- **Clear** — удаляет все текущее содержимое панели **Output**. Применение этой команды не предотвращает последующий вывод информации в панель;
- **Find** — позволяет задать сочетание символов, которое требуется отыскать в текущем содержимом панели **Output**;
- **Find Again** — позволяет найти следующее вхождение сочетания символов, заданного с помощью команды **Find**;
- **Save to File** — сохраняет текущее содержимое панели **Output** в файл с расширением txt;

- **Print** — позволяет вывести содержимое панели **Output** на печать;
- **Filter level** — данный пункт меню включает или отключает дальнейший вывод информации в панель **Output**. Установка значения **none** прекращает вывод информации. При этом текущее содержимое панели не удаляется, однако дальнейший вывод всякой информации (включая сообщения об ошибках и вызовы функции `trace()`) прекращается. Использование данного режима может оказаться полезным при необходимости прекратить вывод результатов периодического вызова функции `trace()`, осуществляемого, например, в обработчике `onEnterFrame()`. Для того чтобы возобновить вывод информации в панель **Output**, необходимо для параметра **Filter level** установить значение **verbose**.

Панель **Output** выполняет несколько функций, приведем их краткий перечень (все приведенные здесь возможности действуют только при тестировании документа в среде тестирования).

- В панель **Output** автоматически выводятся сообщения об ошибках, обнаруженных в процессе компиляции, а также сообщения об ошибках, возникших в процессе воспроизведения (например, превышение 256 уровней рекурсии).
- В панель **Output** выводятся результаты работы функции `trace()`.
- В панель **Output** выводится строковое представление возбужденной, но не обработанной исключительной ситуации.
- В панель **Output** выводится автоматически генерируемый отчет о размерах всех элементов документа. Эта возможность имеет место, только если в настройках параметров публикации был активизирован режим **Generate size report**.
- В панель **Output** может быть по требованию выведен перечень всех объектов, расположенных на сцене.
- В панель **Output** может быть по требованию выведен перечень всех переменных и объектов, используемых в фильме.
- Остановимся более подробно на двух последних опциях.

Вывод списка объектов

Для того чтобы в среде тестирования вывести в панель перечень всех объектов, расположенных на сцене, необходимо выполнить команду главного меню (среды тестирования) **Debug>List Objects**. Эта команда позволяет вывести в панель **Output** список клипов, кнопок, динамических (пользовательских) текстовых полей и векторных форм, включающий указание уровня стека, номера кадра, его метки и полного пути к объекту. Для текстовых полей наряду с идентификатором указываются: переменная (Variable), соответствующая текстовому блоку, режим отображения (Visible) и содержимое текстового блока (Text). Команда **List Objects** не перечисляет объекты, созданные программным образом (такие как `Object`, `Array` и т. д.). При выполнении данной команды текущее содержимое панели **Output** удаляется и в панель выводится перечень объектов, соответствующий текущему состоянию фильма.

Список объектов выводится в соответствии с их структурной иерархией и, в частности, с учетом их местоположения в стеке графических объектов. Объекты, расположенные в стеке ниже, указываются вначале. Если клип или кнопка содержат на сво-

ей монтажной линейке графический объект, указание на этот объект находится под названием родительского объекта и располагается с отступом по горизонтали. Все графические объекты, за исключением клипов, кнопок и текстовых полей (например, объекты рабочего уровня, группы, экземпляры символов типа `graphic`, растровые изображения) обозначаются словом `shape` и не дифференцируются по отдельности. Так, если клип содержит на своей монтажной линейке несколько различных групп, растровое изображение и заливку (форму), то при выводе списка объектов в панель **Output** все содержимое этого клипа будет обозначено словом `shape`.

Список объектов, выведенный с помощью команды **List Objects**, не обновляется автоматически. В панели **Output** отображаются только данные, соответствующие состоянию фильма на момент выполнения команды **List Objects**.

Анализ списка объектов, выведенного командой **List Objects**, позволяет быстро оценить состав документа и легко определить полный путь к этим объектам. Это может оказаться полезным при создании абсолютных или относительных ссылок, а также для проверки соответствия поставленным задачам количества объектов и их размещения в структуре документа.

На рис. 28.4 представлен список объектов, выведенных в панель **Output** в результате выполнения команды **List Objects**. В исходном документе, для которого выведен данный список, на уровне `_level0` находятся векторные объекты рабочего уровня, группы и экземпляр графического символа (все они обозначаются словом `shape`), клип с идентификатором `mc`, кнопка с идентификатором `but` и текстовое поле с идентификатором `txt`. Клип `mc` и кнопка `but` содержат графические объекты, которые обозначаются словом `shape`. На втором уровне стека документов находится клип с идентификатором `clip3`, на монтажной линейке которого находится клип `clip2`, содержащий клип `clip1`, который, в свою очередь, содержит графические объекты рабочего уровня (они обозначаются словом `shape`).



Рис. 28.4. Список объектов, выведенных в панель **Output** в результате выполнения команды **List Objects**

Вывод списка переменных

Для того чтобы в среде тестирования вывести в панель **Output** список всех переменных, используемых в фильме на текущий момент, нужно выполнить команду главного меню **Debug>List Variables**. Эта команда в отличие от команды **List Objects** выводит в панель **Output** не просто объекты, имеющие графическое представление, а все объекты и их свойства, которые присутствуют в проигрывателе во время выполнения данной команды. Такая возможность оказывается очень полезной, поскольку позволяет проанализировать пути и значения переменных, а также получить сведения о свойствах объектов.

Информация, которая приводится в списке, включает полный путь к объекту, породивший его класс и перечень всех его свойств вместе с принимаемыми ими значениями. Список, выводимый командой **List Variables**, также содержит перечень всех глобальных переменных. Глобальные переменные всегда помещаются в начало этого списка. При выводе объектов `Object` указываются их свойства, причем каждое свойство и его значение указывается на новой строке. Аналогично происходит и с массивами. После идентификатора массива выводится список, состоящий из его элементов. На рис. 28.5 приведен список переменных, созданных следующим сценарием:

```
_global.myClass = function(){};
var mc:MovieClip = clip;
var myFunction:Function = function(p1, p2){};
var myObj:Object = {p1:"a", p2:"b", p3:"c"};
var myArray:Array = ["А", "Б", "В"];
var myDate:Date = new Date();
```

Для динамических и пользовательских текстовых блоков кроме идентификатора и полного пути выводится весь набор свойств текстового блока. При этом в каждой строке списка указывается не более четырех свойств. Свойства, имеющие строковые значения, выводятся отдельной строкой.

Как и в случае команды **List Objects**, список, выводимый командой **List Variables**, автоматически не обновляется. Для того чтобы получить сведения о текущем состоянии переменных в фильме, необходимо заново выполнить данную команду.

Примечание

Панель **Output** также может быть вызвана непосредственно в рабочей среде. Для этого применяется команда главного меню **Window>Development Panels>Output**. В рабочей среде информация в панель **Output** выводится при работе с компилированными клипами, например, при отладке программного кода управляющего класса компонента. Кроме того, содержимое, выведенное в панель **Output** в среде тестирования, может оказаться полезным в рабочей среде, например, при внесении изменений в программный код. В этом случае, закрывая окно среды тестирования, следует оставить панель **Output** открытой. Все выведенные в нее в среде тестирования данные будут сохранены и в рабочей среде.

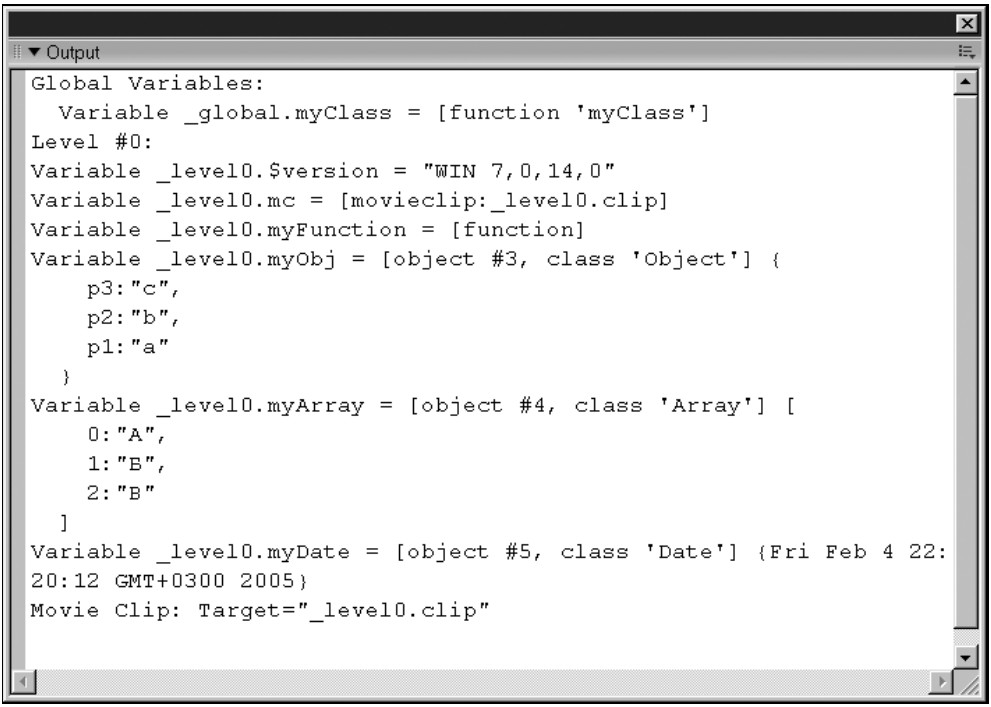


Рис. 28.5. Список переменных, выведенных в панель **Output** в результате выполнения команды **List Variables**

Применение отладчика (панель *Debugger*)

Среда разработки Flash предоставляет достаточно мощное инструментальное средство отладки программного кода, называемое *отладчиком* (панель **Debugger**). Отладчик позволяет осуществлять пошаговое выполнение программного кода, а также модификацию значений переменных и свойств объектов непосредственно в процессе воспроизведения документа. Отладка может выполняться как в среде тестирования для локально расположенных документов, так и для удаленных SWF-файлов, размещенных на сервере.

Для осуществления отладки с применением панели **Debugger** SWF-документ должен воспроизводиться в специальной версии проигрывателя Debug Flash Player, предоставляющей такую возможность. Проигрыватель, позволяющий выполнять отладку, поставляется и автоматически устанавливается вместе с Flash MX 2004. Автономные проигрыватели, поддерживающие возможности отладки, и проигрыватели для браузеров находятся в папке **Debug**, расположенной в стандартном каталоге Players, куда при установке Flash MX 2004 автоматически копируются все проигрыватели Flash.

Отладчик предполагает возможность вывода в панель **Debugger** исходного программного кода FLA-документа для того, чтобы разработчик мог установить точки

прерывания и осуществить пошаговое выполнение снабженных ими инструкций. Поскольку после компиляции программный код ActionScript преобразуется в байт-код, для полноценной работы отладчик необходимо снабдить информацией об исходном программном коде, используемом в FLA-документе. Для этого при публикации FLA-документа, предполагающего возможность отладки, Flash автоматически создает файл в формате SWD, который сохраняется вместе с результирующим SWF-документом. SWD-файл содержит информацию об установленных в рабочей среде точках прерывания, а также исходный программный код и сведения о его местоположении в структуре документа. Для того чтобы увидеть содержимое этого файла, его можно открыть с помощью любого текстового редактора.

Активация отладки в среде тестирования

Для запуска отладки сценариев текущего документа в среде тестирования можно выполнить одно из следующих действий.

- В главном меню рабочей среды выполнить команду **Control>Debug Movie** (<Ctrl>+<Shift>+<Enter>). В результате выполнения данной команды Flash автоматически выполнит публикацию в формат SWF, перейдет в среду тестирования и выведет готовую к работе панель **Debugger**.
- В диалоговом окне **Publish Settings** (**File>Publish Settings...**) на вкладке **Flash** установить флажок **Debugging permitted**. После этого выполнить команду главного меню **Control>Test Movie**. Затем в среде тестирования, щелкнув правой кнопкой на окне фильма, из контекстного меню проигрывателя выбрать команду **Debugger**. В результате этого действия будет выведена панель готового к работе отладчика **Debugger**.

Активация удаленной отладки

Для того чтобы иметь возможность проведения отладки удаленного SWF-документа, расположенного на сервере, необходимо на стадии его создания выполнить следующие подготовительные действия.

1. В диалоговом окне **Publish Settings** (**File>Publish Settings...**) на вкладке **Flash** установить флажок **Debugging permitted**. Если при публикации документа флажок **Debugging permitted** был снят, то удаленная отладка этого документа будет невозможна.
2. Для того чтобы отладка документа могла производиться только лицами, обладающими правами доступа к содержимому документа, его можно защитить паролем. При установке флажка **Debugging permitted** автоматически активизируется поле **Password**, в которое можно ввести пароль, позволяющий получить доступ к отладке документа. При попытке пользователя выполнить отладку документа будет выведено окно с полем для ввода пароля. Если введенный пользователем пароль не совпадет с паролем, установленным разработчиком, выполнение отладки окажется невозможным. Если защита паролем не является необходимой, можно просто оставить поле **Password** пустым. В этом случае пользователь все равно увидит окно с предложением ввести пароль, однако, оставив поле пустым и нажав кнопку **OK**, сможет запустить отладку.

3. Выполнить публикацию документа (**File>Publish**). При публикации Flash наряду с SWF-документом создаст SWD-документ, содержащий информацию, необходимую для работы отладчика. При размещении файлов проекта на удаленном сервере SWD-документ следует поместить в тот же каталог, в котором будет расположен соответствующий ему SWF-документ. В случае отсутствия SWD-файла возможности отладки будут существенно сужены из-за невозможности применения точек прерывания и пошагового выполнения программного кода.

Для того чтобы запустить отладку удаленного SWF-документа, необходимо выполнить следующие действия:

1. Открыть приложение Flash MX 2004.
2. Открыть панель **Debugger**. Для того чтобы это сделать, можно выполнить одно из следующих действий:
   - в главном меню рабочей среды выполнить команду **Window>Development Panels>Debugger**;
   - в главном меню среды тестирования выполнить команду **Window>Debugger**;
   - воспользоваться сочетанием клавиш <Shift>+<F4>.
3. В контекстном меню панели **Debugger** выполнить команду **Enable Remote Debugging**.
4. Открыть расположенный на сервере SWF-документ в браузере или в автономном проигрывателе, поддерживающем возможности отладки (Debug Flash Player). После загрузки документа должно появиться окно **Remote Debug**, представленное на рис. 28.6 Если проигрыватель Debug Flash Player и приложение Flash MX 2004 работают на текущей машине, необходимо установить переключатель в положение **Localhost**. Если Debug Flash Player и Flash MX 2004 работают на другом компьютере, необходимо установить переключатель в положение **Other Machine** и в поле **Enter IP address** указать IP-адрес этого компьютера.

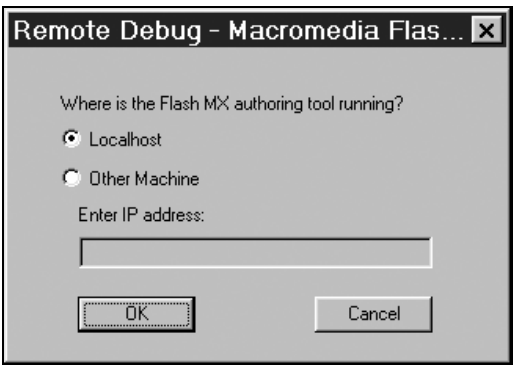


Рис. 28.6. Окно **Remote Debug**

Если после окончания загрузки SWF-файла окно **Remote Debug** не появляется, это означает, что Flash не может найти соответствующий данному файлу SWD-

документ. В этом случае следует щелкнуть правой кнопкой мыши внутри окна проигрывателя и из контекстного меню выбрать команду **Debugger**.

5. В появившемся диалоговом окне **Debugging Password Required** ввести пароль, установленный в настройках публикации. Если пароль не задавался, следует оставить поле пустым и нажать кнопку **OK**.

В результате выполненных действий в среде Flash будет открыта панель **Debugger**. В поле, расположенном в левом верхнем углу этой панели, указывается местоположение текущего документа. При запуске отладчика выполнение сценариев и воспроизведение монтажных линеек фильма приостанавливаются для того, чтобы разработчик мог установить точки прерывания и реализовать пошаговое выполнение кода. Для того чтобы запустить воспроизведение, необходимо выполнить команду **Continue** контекстного меню панели **Debugger** или воспользоваться кнопкой с изображением зеленой стрелки , расположенной в правой части панели.

Инструментарий панели *Debugger*

Панель **Debugger** состоит из двух разделов. В левой части этой панели приводится информация, отражающая структуру документа и текущее состояние программных элементов. В правой части выводится исходный программный код. Внешний вид панели **Debugger** представлен на рис. 28.7.

Рассмотрим назначение инструментов, расположенных в левой части панели.

- **Адресная строка** — предназначена для вывода местоположения файла, для которого выполняется отладка.
- **Окно структуры** — в этом окне отображается иерархическая структура всех заполненных уровней стека документов и расположенных на них клипов. Указание клипов выполняется с применением абсолютной адресации. Здесь также указывается объект `_global`. Для получения сведений о составе и значениях переменных и свойств элемента фильма (объекта `_global`, уровня или клипа) достаточно выделить его в иерархическом дереве.
- **Вкладка свойств (Properties)** — эта вкладка позволяет управлять встроенными свойствами клипов (рис. 28.8). Здесь в столбце **Name** приведен перечень всех встроенных свойств клипов. Свойства, используемые только для чтения, указаны светло-серым цветом. В столбце **Value** указаны значения всех встроенных свойств для объекта, который в текущий момент выделен в иерархическом дереве окна структуры. Если какие-то из перечисленных свойств для данного объекта не определены (то есть принимают значение `undefined`), их значение не выводится и соответствующая графа в столбце **Value** остается пустой.

  Значения свойств, доступных для редактирования, могут быть изменены непосредственно в процессе воспроизведения SWF-документа. Для того чтобы присвоить свойству клипа новое значение, его нужно просто ввести в соответствующую графу столбца **Value**. В качестве значений свойств могут выступать только данные элементарного типа (Number, String, Boolean). Изменение значений свойств клипов во вкладке **Properties** сразу же отражается в фильме, однако это происходит только при условии, что воспроизведение фильма и выполнение

кода не остановлено из-за применения точек прерывания. Изменение значений свойств клипов, реализуемое за счет работы программного кода, также сразу же отражается во вкладке **Properties**.

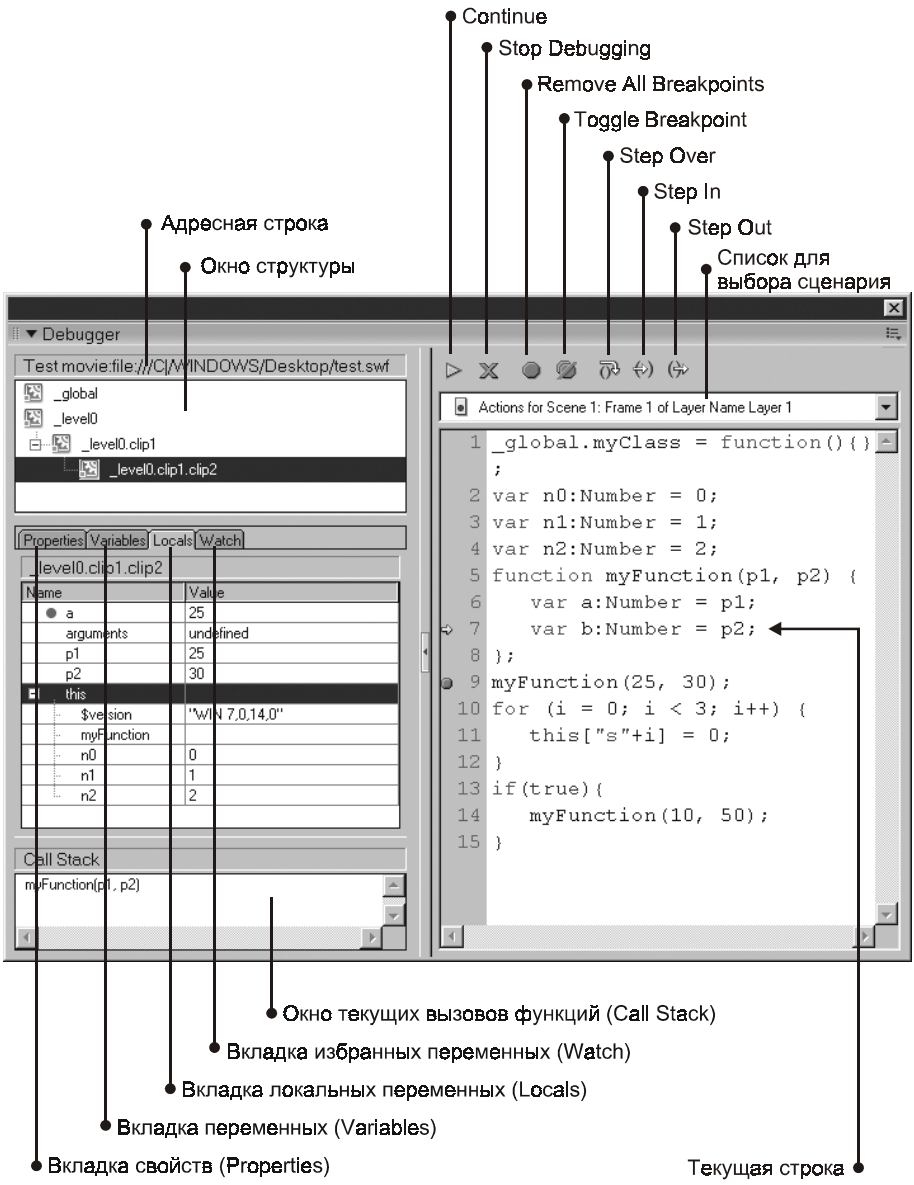


Рис. 28.7. Панель **Debugger**

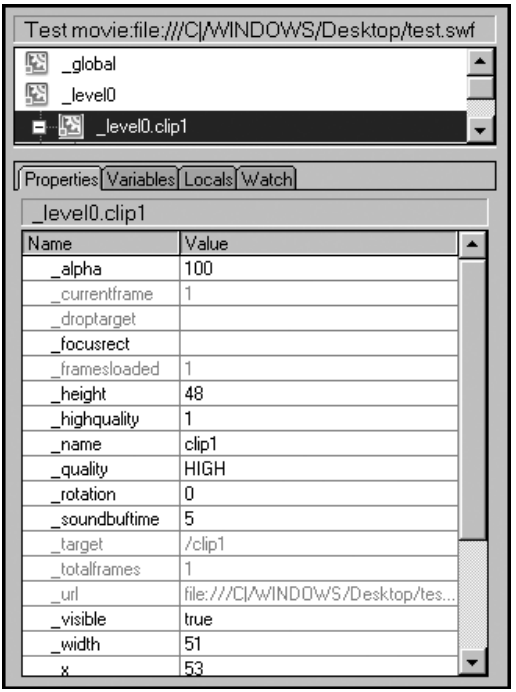


Рис. 28.8. Вкладка **Properties** панели **Debugger**

- **Вкладка переменных (Variables)** — эта вкладка содержит перечень всех переменных монтажной линейки клипа или объекта, выделенного в окне структуры, которые существуют в текущий момент. Значения и состав отображаемых переменных обновляются по мере воспроизведения фильма и выполнения программного кода. Значение переменной может быть задано вручную, подобно тому, как это делается со свойствами клипов во вкладке **Properties**. При задании значения переменной можно использовать только данные элементарных типов.

  Переменные, значениями которых являются ссылки на объекты `Object` или массивы, выводятся в виде элементов, содержащих подчиненные элементы. Это дает возможность контролировать и модифицировать в процессе отладки не только значения переменных элементарного типа, но и свойства объектов и значения элементов массивов. На рис. 28.9 представлена вкладка **Variables**, в которой выведена информация о переменных, созданных следующим кодом:

```
var k:Object = {a:1, b:2}
var ar:Array = ["A", "B", "C"]
```

- **Вкладка локальных переменных (Locals)** — эта вкладка содержит перечень всех текущих локальных переменных, существующих в фильме. Информация, выводимая в этой вкладе, не зависит от того, какой элемент выделен в окне структуры, а определяется только наличием локальных переменных в данный момент. Эта вкладка может содержать информацию только в том случае, если выполне-

ние программного кода остановлено с помощью точки прерывания внутри тела функции. Локальные переменные существуют только на протяжении времени работы функции, поэтому вкладка **Locals** остается заполненной только до тех пор, пока выполняются инструкции, расположенные в теле функции. Применение данной вкладки очень похоже на применение вкладки **Properties**. Значения переменных также могут задаваться вручную и автоматически обновляются в результате выполнения программного кода. При задании значения переменной можно использовать только данные элементарных типов.

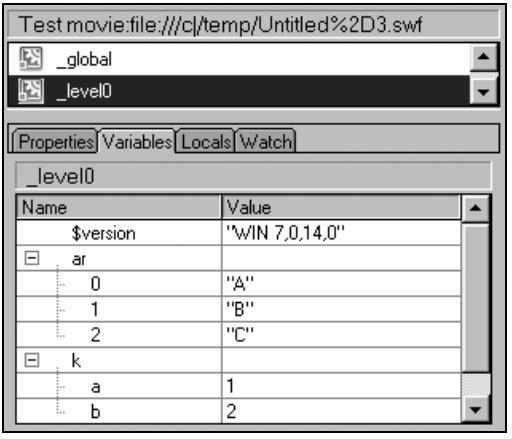


Рис. 28.9. Вкладка **Variables** панели **Debugger**

- **Вкладка избранных переменных (Watch)** — эта вкладка предназначена для вывода списка переменных, за которыми ведется наблюдение. Поскольку программный код может включать очень большое количество переменных, а при отладке, как правило, требуется контролировать только некоторые из них, удобно вынести все интересующие переменные в отдельный список, где их значения можно отслеживать и при необходимости изменять. Такой список позволяет сформировать вкладка **Watch**. В эту вкладку можно поместить как переменную монтажной линейки, так и локальную переменную. Однако сюда могут быть помещены только переменные, имеющие значение элементарного типа. Допускается размещение в списке **Watch** свойств объектов или элементов массивов. Список **Watch** является общим для всех клипов документа, то есть его содержимое не зависит от того, какой клип выделен в окне структуры в данный момент. Для того чтобы поместить переменную в список **Watch**, необходимо выполнить одно из следующих действий.
  - Во вкладке **Variables** или во вкладке **Locals** щелкнуть правой кнопкой мыши на графе, соответствующей требуемой переменной, и во всплывающем меню установить флажок рядом с командой **Watch**. При этом рядом с именем переменной будет отображена синяя точка.
  - Во вкладке **Watch** щелкнуть правой кнопкой мыши и из контекстного меню выбрать команду **Add**, в результате чего во вкладке появится новая графа. По-

сле этого, дважды щелкнув в столбец **Name** новой графы, ввести абсолютный путь к требуемой переменной. Если такая переменная уже существует, в столбце **Value** автоматически появится ее значение.

Для того чтобы удалить переменную из списка **Watch**, необходимо выполнить одно из следующих действий.

- Во вкладке **Variables** или во вкладке **Locals** щелкнуть правой кнопкой мыши на графе, соответствующей требуемой переменной (помеченной синей точкой), и во всплывающем меню снять флажок рядом с командой **Watch**.
- Во вкладке **Watch** щелкнуть правой кнопкой мыши на графе, соответствующей требуемой переменной, и из контекстного меню выбрать команду **Remove**.

- **Окно текущих вызовов функций (Call Stack)** — в это окно выводятся названия функций, активированных в текущий момент. Имя функции выводится в окно **Call Stack** на протяжении пошагового выполнения инструкций, расположенных в теле этой функции. Для того чтобы имя функции было выведено в этом окне, необходимо поместить точку прерывания на строку кода, содержащую вызов данной функции, и/или на строку, расположенную в теле вызванной функции. Так, например, при рекурсивном вызове функции на каждом уровне рекурсии в окне **Call Stack** будет выводиться имя этой функции. После выхода из блока инструкций, расположенных в теле функции, имя этой функции удаляется из окна **Call Stack**.

Установка точек прерывания

Точки прерывания (breakpoints) позволяют приостановить выполнение программного кода на требуемой строке и, таким образом, реализовать его пошаговое выполнение. Это оказывается чрезвычайно полезным для анализа работы функций, условных предложений, переключателей, циклов и т. д. Применение точек прерывания позволяет последовательно пройти все ключевые этапы выполнения программы и проследить, как себя ведут различные элементы программного кода на каждом из таких этапов.

Точки прерывания можно установить как в панели **Actions** в рабочей среде, так и непосредственно в панели **Debugger** в процессе отладки. Для того чтобы установить точку прерывания в панели **Actions**, достаточно просто щелкнуть на левом поле соответствующей строки программного кода. Точка прерывания обозначается красным кружком, который появляется слева от строки кода. Для того чтобы удалить точку прерывания, достаточно просто щелкнуть на ней. Кроме того, управлять установкой и удалением точек прерывания можно с помощью команд меню **Debug Options**, появляющегося при щелчке на кнопке , расположенной в правом верхнем углу панели **Actions**. Точки прерывания, установленные в панели **Actions**, сохраняются перманентно на протяжении всех сеансов отладки. Следует иметь в виду, что при выполнении автоматического форматирования кода расположение строк может измениться, и при этом точки прерывания окажутся установленными неверно. В связи с этим следует либо не использовать автоформатирование после установки точек прерывания, либо проверять их расположение после выполнения автоформатирования.

Для того чтобы установить точки прерывания в панели **Debugger**, необходимо вначале в выпадающем списке **Navigate to other scripts**, расположенном в верхней части правой половины, выбрать сценарий, для которого следует установить точки прерывания. Данный список содержит перечень всех сценариев, расположенных на монтажных линейках и прикрепленных непосредственно к клипам и кнопкам. После этого выбранный код будет выведен в правой части панели **Debugger**. Для установки и удаления точек прерывания можно щелкать по левому полю напротив требуемой строки так же, как это делается при установке и удалении точек прерывания в панели **Actions**. Кроме того, можно использовать кнопки, расположенные справа в верхней части панели. Кнопка **Toggle Breakpoint**  позволяет установить или удалить точку прерывания на строке, в которой в данный момент находится курсор. Кнопка **Remove All Breakpoints**  удаляет все установленные точки прерывания.

Точки прерывания можно устанавливать не на любой строке кода. Так, например, точки прерывания не устанавливаются на строках, содержащих объявление функции, оператор `if` условного предложения, скобку и т. д. Если точка прерывания была некорректно установлена в панели **Actions**, то при публикации она будет автоматически удалена, а в панель **Output** будет выведено соответствующее сообщение. В панели **Debugger** некорректно установить точки прерывания вообще не удастся, поскольку они будут устанавливаться только там, где это допустимо.

Применение точек прерывания. Пошаговое выполнение кода

При запуске процедуры отладки воспроизведение всех монтажных линеек документа и выполнение всех расположенных в нем сценариев приостанавливается. Это позволяет произвести расстановку точек прерывания. Для того чтобы запустить воспроизведение и выполнение кода, необходимо либо воспользоваться кнопкой **Continue** , либо выполнить команду **Continue** (<F10>) из контекстного меню панели **Debugger**. Для пошагового выполнения кода необходимо перед запуском воспроизведения снабдить его хотя бы одной точкой прерывания. Точки прерывания можно поместить в блоки кода, относящиеся к разным монтажным линейкам или объектам (клипам и кнопкам). Последовательность пошагового выполнения фрагментов кода в точности соответствует обычной последовательности выполнения сценариев, расположенных на разных монтажных линейках.

После выполнения команды **Continue** программный код будет выполнен до появления первой строки, снабженной точкой прерывания. После встречи точки прерывания выполнение кода будет приостановлено, причем снабженная ею строка выполнена не будет. Для индикации строки, на которой в данный момент приостановлено выполнение кода, слева от нее появляется желтая стрелка (7-я строка на рис. 28.7). Для управления пошаговым выполнением кода используются кнопки, расположенные справа в верхней части панели **Debugger**. Рассмотрим назначение этих кнопок, иллюстрируя их применение на примере следующего кода (для удобства приведем здесь номера строк). Приведенный код читатель сможет найти в файле test.fla, расположенном в папке Глава 28 на сопровождающем книгу компакт-диске для того, чтобы самостоятельно выполнить описанные далее действия.

```
1       _global.myClass = function(){};
2       var n0:Number = 0;
```

```
3       var n1:Number = 1;
4       var n2:Number = 2;
5       function myFunction(p1, p2) {
6               var a:Number = p1;
7               var b:Number = p2;
8       };
9       myFunction(25, 30);
10      for (i = 0; i < 3; i++) {
11              this["s"+i] = 0;
12      }
```

- Кнопка **Continue**  — данная кнопка возобновляет выполнение кода до появления ближайшей точки прерывания. Так, если в приведенном выше коде точки прерывания установлены на строках 2 и 9, то при первом нажатии кнопки **Continue** будет выполнена первая строка кода, и дальнейшее выполнение будет остановлено на второй строке. При повторном нажатии данной кнопки будут выполнены все строки со второй по восьмую включительно, и дальнейшее выполнение кода будет приостановлено на строке с номером 9.

- Кнопка **Step In**  — данная кнопка позволяет осуществлять построчное выполнение кода. Если выполнение кода остановлено на строке, содержащей точку прерывания и не являющейся вызовом функции или заголовком цикла, то нажатие кнопки **Step In** приведет к переходу на следующую строку кода. Так, если выполнение приведенного кода остановлено на строке с номером 2, нажатие данной кнопки приведет к выполнению второй строки и переходу с последующей остановкой на строку с номером 3.

  Если строка, на которой остановлено выполнение кода, содержит вызов пользовательской функции (метода) или заголовок цикла, то нажатие кнопки **Step In** инициирует переход к первой строке, расположенной в теле этой функции или цикла. При последующих нажатиях данной кнопки будет происходить построчное выполнение инструкций, расположенных в теле функции или цикла, причем в случае цикла будет иметь место последовательный проход всех инструкций тела цикла для каждой итерации. Приведем несколько примеров.

  Если выполнение рассматриваемого кода остановлено на строке с номером 9, содержащей вызов функции, то после нажатия кнопки **Step In** будет выполнен переход на строку с номером 6, соответствующую первому предложению тела вызванной функции. При этом на вкладке **Locals** будет выведена информация о фактических значениях параметров, переданных функции при ее вызове. Следующее нажатие кнопки **Step In** приведет к выполнению шестой строки и переходу к строке с номером 7. При этом на вкладке **Locals** появится локальная переменная `a`.

  Если выполнение рассматриваемого кода остановлено на строке с номером 10, соответствующей заголовку цикла, то будет выполнен переход на строку с номером 11, соответствующую первому предложению тела цикла. При этом на вклад-

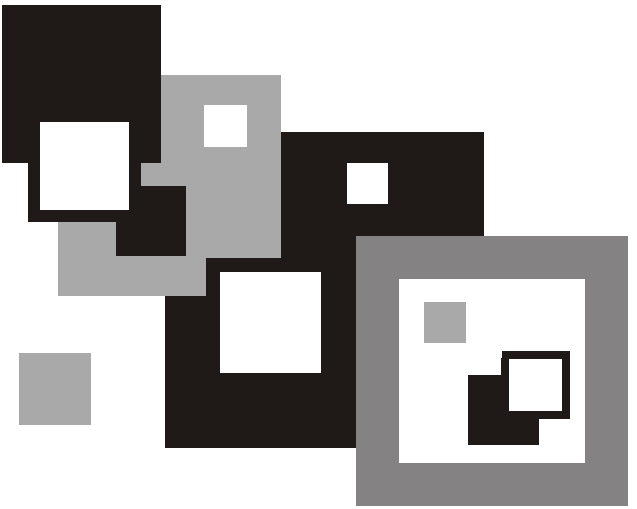
ке **Variables**, соответствующей объекту `_level0`, появится инициализированная в заголовке цикла переменная `i` (счетчик цикла), принимающая значение 0. Поскольку в приведенном примере тело цикла состоит из единственного предложения, при следующем нажатии кнопки желтая стрелка, указывающая текущую строку кода, останется на строке с номером 11, однако при этом переменная `i` на вкладке **Variables** уже будет принимать значение 1, а в списке переменных появится созданная на первой итерации цикла в результате выполнения одиннадцатой строки переменная `s0`, принимающая значение 0.

- Кнопка **Step Over**  — данная кнопка предназначена для выполнения кода текущей строки и перехода к следующей строке кода. Результат применения этой кнопки во многом похож на результат применения кнопки **Step In**. Принципиальное различие заключается в обработке вызовов пользовательских функций. Если выполнение кода остановлено на строке, содержащей вызов пользовательской функции, то нажатие кнопки **Step Over** инициирует выполнение этой функции и остановит выполнение программного кода на строке, следующей за строкой с ее вызовом. Таким образом, кнопка **Step In** позволяет пошагово пройти блок кода, расположенный в теле функции, в то время как кнопка **Step Over** "перескакивает" через тело функции, хотя ее код при этом выполняется. Так, если выполнение рассматриваемого кода остановлено на строке с номером 9, содержащей вызов функции, то после нажатия кнопки **Step Over** код этой функции будет выполнен, после чего произойдет переход на следующую строку с номером 10, и дальнейшее выполнение кода будет приостановлено.

- Кнопка **Step Out**  — данная кнопка применяется, когда выполнение кода остановлено на строке, расположенной в теле вызванной пользовательской функции. При ее нажатии происходит выполнение всех остальных инструкций, расположенных в теле функции, и последующий переход на строку, следующую за строкой с вызовом функции. Таким образом, назначение данной кнопки состоит в выходе из тела функции и остановке выполнения кода на строке, следующей за ее вызовом. Так, если выполнение кода остановлено на строке с номером 6, то при нажатии кнопки происходит выполнение инструкций, расположенных в теле функции и остановка выполнения кода на строке с номером 10.

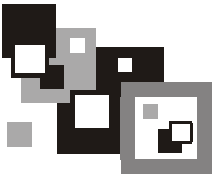
  Если выполнение кода остановлено не на строке, расположенной в теле пользовательской функции, то нажатие кнопки **Step Out** приводит точно к такому же результату, что и нажатие кнопки **Continue** — выполнение программного кода вплоть до следующей точки прерывания. Так, если точки прерывания установлены на строке 3 и строке 10, то при остановке выполнения кода на третьей строке нажатие кнопки **Step Out** приводит к выполнению всех инструкций до строки 9 включительно и выполнение кода останавливается на строке 10.

- Кнопка **Stop Debugging**  — данная кнопка предназначена для прекращения сеанса отладки. Ее нажатие приводит к удалению всех точек прерывания, продолжению воспроизведения фильма и выполнению программного кода в обычном режиме. При этом все инструментальные средства панели **Debugger** становятся неактивными, что исключает дальнейшую возможность ее применения.



Приложения

Приложение 1



Справочник: методы, свойства и обработчики событий основных встроенных классов ActionScript

Класс *Array*. Создание и обработка массивов

Метод	Описание
Добавление элементов в массив	
`Array.concat()`	Добавляет произвольное количество элементов в конец массива, а также объединяет несколько массивов, создавая новый массив
`Array.push()`	Добавляет новые элементы в конец массива и возвращает обновленное значение его свойства `length`
`Array.splice()`	Удаляет элементы из произвольного участка массива; добавляет новые элементы в произвольный участок массива; одновременно удаляет произвольное количество элементов, начиная с заданного индекса, и добавляет в этот участок массива произвольное количество новых элементов
`Array.unshift()`	Добавляет новые элементы в начало массива и возвращает обновленное значение его свойства `length`
Объединение элементов массива	
`Array.join()`	Объединяет все элементы массива в строку, позволяя задать произвольный разделитель элементов массива. Полученная строка возвращается в качестве результата
`Array.toString()`	Создает строковые представления для каждого элемента массива и объединяет их в единую строку, где строковые представления разных элементов разделяются запятыми. Полученная таким образом строка возвращается в качестве результата
Сортировка массива	
`Array.reverse()`	Позволяет инвертировать порядок следования элементов в исходном массиве

(окончание)

Метод	Описание
Добавление элементов в массив	
`Array.sort()`	Сортирует элементы массива на основании критериев, зависящих от числа, типа и значения передаваемых параметров
`Array.sortOn()`	Сортирует элементы массива, состоящего из объектов по определенному свойству или полю (field)
Извлечение части массива	
`Array.slice()`	Извлекает набор последовательно расположенных элементов массива и возвращает их как новый массив
Удаление элементов массива	
`Array.pop()`	Удаляет последний элемент из массива и возвращает его значение
`Array.shift()`	Удаляет первый элемент из массива и возвращает его значение
Свойство	**Описание**
`Array.length`	Содержит значение, отражающее количество элементов в массиве

Класс *Button*. Работа с кнопками

Метод	Описание
`Button.getDepth()`	Возвращает номер уровня стека (глубину), на котором находится экземпляр кнопки
Свойство	**Описание**
`Button._alpha`	Возвращает значение прозрачности экземпляра кнопки
`Button.enabled`	Определяет, является ли кнопка активной
`Button._focusrect`	Определяет, включен ли режим, при котором кнопка, находящаяся в фокусе, подсвечивается желтой рамкой
`Button._height`	Возвращает высоту экземпляра кнопки в пикселах
`Button._highquality`	Определяет степень сглаживания графики в документе (глобальное свойство)
`Button.menu`	Связывает объект `ContextMenu` с экземпляром кнопки
`Button._name`	Содержит имя экземпляра кнопки
`Button._parent`	Возвращает ссылку на объект, содержащий экземпляр кнопки

(продолжение)

Свойство	Описание
`Button._quality`	Определяет качество отображения графики в документе (глобальное свойство)
`Button._rotation`	Возвращает угол поворота экземпляра кнопки в градусах
`Button._soundbuftime`	Возвращает количество секунд перед началом воспроизведения потокового звука (глобальное свойство)
`Button.tabEnabled`	Определяет, включен ли данный экземпляр кнопки в автоматический порядок переноса фокуса
`Button.tabIndex`	Указывает индекс экземпляра кнопки в используемом порядке переноса фокуса
`Button._target`	Возвращает полный путь к экземпляру кнопки в нотации с наклонной чертой (устаревший способ адресации)
`Button.trackAsMenu`	Определяет, используется ли данным экземпляром кнопки режим `trackAsMenu`
`Button._url`	Возвращает URL документа SWF, в котором создана кнопка
`Button.useHandCursor`	Определяет тип указателя курсора, используемого при взаимодействии с данным экземпляром кнопки
`Button._visible`	Определяет, является ли кнопка видимой
`Button._width`	Возвращает ширину кнопки в пикселах
`Button._x`	X-координата точки регистрации кнопки
`Button._xmouse`	X-координата курсора мыши относительно начала координат объекта, содержащего кнопку
`Button._xscale`	Масштаб кнопки по горизонтали
`Button._y`	Y-координата точки регистрации кнопки
`Button._ymouse`	Y-координата курсора мыши относительно начала координат объекта, содержащего кнопку
`Button._yscale`	Масштаб кнопки по вертикали
Обработчик события	**Описание**
`Button.onDragOut`	Вызывается, когда курсор выходит за пределы области реагирования кнопки при нажатой кнопке мыши
`Button.onDragOver`	Вызывается, когда курсор выходит за пределы области реагирования кнопки при нажатой кнопке мыши, после чего вновь возвращается в область реагирования, кнопка мыши остается нажатой

(окончание)

Обработчик события	Описание
`Button.onPress`	Вызывается, когда кнопка мыши нажата при нахождении курсора в области реагирования кнопки
`Button.onRelease`	Вызывается, когда кнопка мыши нажата и затем отпущена при нахождении курсора в области реагирования кнопки
`Button.onReleaseOutside`	Вызывается, когда курсор выходит за пределы области реагирования кнопки при нажатой кнопке мыши, после чего кнопка мыши отпускается вне области реагирования
`Button.onRollOut`	Вызывается, когда курсор выходит за пределы области реагирования кнопки при ненажатой кнопке мыши
`Button.onRollOver`	Вызывается, когда курсор входит в область реагирования кнопки. Если в момент вхождения кнопка мыши оказывается нажата, данное событие генерируется в момент отпускания кнопки мыши при нахождении ее курсора в активной области кнопки
`Button.onKeyUp`	Вызывается при отпускании клавиатурной кнопки
`Button.onSetFocus`	Вызывается, когда происходит передача фокуса кнопке
`Button.onKillFocus`	Вызывается, когда фокус переходит с кнопки на другой объект

Класс *Color*. Работа с цветом

Метод	Описание
`Color.getRGB()`	Возвращает значение цвета, установленного последним вызовом метода `setRGB()`
`Color.getTransform()`	Возвращает объект, содержащий настройки цветовой трансформации, установленные последним вызовом метода `setTransform()`
`Color.setRGB()`	Задает цвет окрашивания для объекта `Color` на основе его шестнадцатеричного RGB-значения
`Color.setTransform()`	Задает параметры цветовой трансформации для объекта `Color`

Класс *ContextMenu*. Управление контекстным меню проигрывателя

Метод	Описание
`ContextMenu.copy()`	Создает копию объекта `ContextMenu`
`ContextMenu.hideBuiltInItems()`	Скрывает все стандартные элементы меню
Свойство	**Описание**
`ContextMenu.builtInItems`	Избирательно выводит стандартные команды контекстного меню
`ContextMenu.customItems`	Массив, содержащий объекты `ContextMenu` (по умолчанию неопределен), представляющие пользовательские элементы контекстного меню
Обработчик события	**Описание**
`ContextMenu.onSelect`	Вызывается в промежутке между нажатием правой кнопки мыши на объекте, связанном с контекстным меню, и выводом этого меню

Класс *ContextMenuItem*. Пользовательские элементы контекстного меню

Свойство	Описание
`ContextMenuItem.caption`	Принимает строковое значение, которое будет выводиться в контекстном меню в качестве названия пункта меню
`ContextMenuItem.enabled`	Принимает булево значение, определяющее, является ли элемент меню активным
`ContextMenuItem.separatorBefore`	Принимает булево значение, определяющее, будет ли в контекстном меню над данным элементом находиться разделительная линия
`ContextMenuItem.visible`	Принимает булево значение, определяющее, будет ли элемент отображаться в меню
Обработчик события	**Описание**
`ContextMenuItem.onSelect`	Вызывается при выборе элемента меню

Класс *Date*. Контроль времени и даты

Метод	Описание
`Date.toString()`	Возвращает дату и время в виде строкового представления объекта `Date`
`Date.getTimezoneOffset()`	Возвращает сдвиг местного времени по отношению к гринвичскому времени (UTC)
Определение местного времени и даты	
`Date.getFullYear()`	Возвращает год как четырехзначное число
`Date.getYear()`	Возвращает значение, представляющее разность `getFullYear()` — 1900
`Date.getMonth()`	Возвращает номер месяца (от 0 до 11)
`Date.getDate()`	Возвращает число (от 1 до 31)
`Date.getDay()`	Возвращает номер дня недели (от 0 до 6, причем 0 соответствует воскресенью, 1 — понедельнику и т. д.)
`Date.getHours()`	Возвращает часы (от 0 до 23)
`Date.getMinutes()`	Возвращает минуты (от 0 до 59)
`Date.getSeconds()`	Возвращает секунды (от 0 до 59)
`Date.getMilliseconds()`	Возвращает миллисекунды (от 0 до 999)
Определение UTC-времени и даты	
`Date.getUTCFullYear()`	Возвращает год как четырехзначное число
`Date.getUTCMonth()`	Возвращает номер месяца (от 0 до 11)
`Date.getUTCDate()`	Возвращает число (от 1 до 31)
`Date.getUTCDay()`	Возвращает номер дня недели (от 0 до 6, причем 0 соответствует воскресенью, 1 — понедельнику и т. д.)
`Date.getUTCHours()`	Возвращает часы (от 0 до 23)
`Date.getUTCMinutes()`	Возвращает минуты (от 0 до 59)
`Date.getUTCSeconds()`	Возвращает секунды (от 0 до 59)
`Date.getUTCMilliseconds()`	Возвращает миллисекунды (от 0 до 999)
Установка местного времени и даты	
`Date.setFullYear()`	Задает год как четырехзначное число. Возвращает новое время в миллисекундах. Наряду с первым обязательным параметром, задающим год, данный метод может принимать еще два необязательных параметра: месяц и число

(окончание)

Метод	Описание
Установка местного времени и даты	
`Date.setYear()`	Задает число. Возвращает новое время в миллисекундах
`Date.setMonth()`	Задает номер месяца. Возвращает новое время в миллисекундах. Наряду с первым обязательным параметром, задающим номер месяца, данный метод может принимать еще один необязательный параметр: число
`Date.setDate()`	Задает номер дня недели (0 соответствует воскресенью, 1 — понедельнику и т. д.). Возвращает новое время в миллисекундах
`Date.setHours()`	Задает часы. Возвращает новое время в миллисекундах
`Date.setMinutes()`	Задает минуты. Возвращает новое время в миллисекундах
`Date.setSeconds()`	Задает секунды. Возвращает новое время в миллисекундах.
`Date.setMilliseconds()`	Задает миллисекунды. Возвращает новое время в миллисекундах
Установка UTC-времени и даты	
`Date.setUTCFullYear()`	Задает год как четырехзначное число. Возвращает новое время в миллисекундах. Наряду с первым обязательным параметром, задающим год, данный метод может принимать еще два необязательных параметра: месяц и число
`Date.setUTCMonth()`	Задает номер месяца. Возвращает новое время в миллисекундах. Наряду с первым обязательным параметром, задающим номер месяца, данный метод может принимать еще один необязательный параметр: число
`Date.setUTCDate()`	Задает номер дня недели (0 соответствует воскресенью, 1 — понедельнику и т. д.). Возвращает новое время в миллисекундах
`Date.setUTCHours()`	Задает часы. Возвращает новое время в миллисекундах
`Date.setUTCMinutes()`	Задает минуты. Возвращает новое время в миллисекундах
`Date.setUTCSeconds()`	Задает секунды. Возвращает новое время в миллисекундах
`Date.setUTCMilliseconds()`	Задает миллисекунды. Возвращает новое время в миллисекундах

Класс *Error*. Исключительные ситуации

Метод	Описание
`Error.toString()`	Возвращает строку, содержащую значение свойства `message` объекта `Error`
Свойство	**Описание**
`Error.message`	Содержит строку с описанием исключительной ситуации
`Error.name`	Содержит строку с именем конструктора класса, породившего объект `Error`

Класс *Key*. Контроль клавиатуры

Метод	Описание
`Key.addListener()`	Регистрирует объект-приемник на принятие сообщений о событиях от объекта `Key`
`Key.getAscii()`	Возвращает ASCII-код последней нажатой клавиши
`Key.getCode()`	Возвращает виртуальный код последней нажатой клавиши
`Key.isDown()`	Проверяет факт нажатия указанной клавиши. Возвращает булево значение `true`, если клавиша нажата, и `false` — если клавиша не нажата
`Key.isToggled()`	Возвращает булево значение `true`, если клавиши <Num Lock> или <Caps Lock> активны
`Key.removeListener()`	Удаляет объект-приемник
Событие	**Описание**
`Key.onKeyDown`	Событие генерируется при нажатии любой клавиши
`Key.onKeyUp`	Событие генерируется при отпускании любой нажатой клавиши

Класс *LocalConnection*. Взаимодействие между фильмами

Метод	Описание
`LocalConnection.close()`	Закрывает соединение, открытое принимающим объектом `LocalConnection`

(окончание)

Метод	Описание
`LocalConnection.connect()`	Открывает соединение с заданным именем, подготавливая принимающий объект `LocalConnection` к принятию инструкции от метода `send()` отправляющего объекта `LocalConnection`
`LocalConnection.domain()`	Возвращает строку с именем домена, в котором расположен SWF-файл
`LocalConnection.send()`	Инициирует вызов метода принимающего объекта `LocalConnection`, подключенного к соединению, имя которого передается методу `send()` в качестве параметра
Обработчик события	**Описание**
`LocalConnection.allowDomain`	Событие генерируется при получении запроса на вызов методов принимающего объекта `LocalConnection` от файла-отправителя. Код в методе обработчика должен возвращать булево значение
`LocalConnection.allowInsecureDomain`	Событие генерируется при попытке осуществить передачу инструкций объекту `LocalConnection` файла-получателя, использующему протокол HTTPS, из файла, использующего протокол HTTP
`LocalConnection.onStatus`	Событие генерируется объектом `LocalConnection` файла-отправителя, если метод `send()` возвращает значение `true` (то есть если его вызов является синтаксически корректным)

Класс *Math*. Выполнение математических вычислений

Метод	Описание
Математические функции	
`Math.abs()`	Вычисляет модуль числа
`Math.atan2()`	Вычисляет угол поворота заданной точки по отношению к оси X
`Math.ceil()`	Округляет число до ближайшего целого сверху
`Math.exp()`	Возводит *e* в заданную степень
`Math.floor()`	Возвращает ближайшее целое число, меньшее или равное аргументу

(окончание)

Метод	Описание
Математические функции	
`Math.log()`	Вычисляет натуральный логарифм числа
`Math.max()`	Возвращает наибольшее из двух чисел
`Math.min()`	Возвращает наименьшее из двух чисел
`Math.pow()`	Возводит число в заданную степень
`Math.random()`	Возвращает псевдослучайное число с плавающей точкой в полуинтервале от 0 до 1 (не включая 1)
`Math.round()`	Вычисляет ближайшее целое
`Math.sqrt()`	Вычисляет корень квадратный из числа
Тригонометрические функции	
`Math.acos()`	Вычисляет арккосинус угла
`Math.asin()`	Вычисляет арксинус угла
`Math.atan()`	Вычисляет арктангенс угла
`Math.cos()`	Вычисляет косинус угла
`Math.sin()`	Вычисляет синус угла
`Math.tan()`	Вычисляет тангенс угла
Математические константы	
Свойство	**Описание**
`Math.E`	Постоянная Эйлера, основание натуральных логарифмов (приблизительно 2,718)
`Math.LN2`	Натуральный логарифм 2 (приблизительно 0,693)
`Math.LN10`	Натуральный логарифм 10 (приблизительно 2,302)
`Math.LOG10E`	Логарифм *e* по основанию 10 (приблизительно 0,434)
`Math.LOG2E`	Логарифм *e* по основанию 2 (приблизительно 1,442)
`Math.PI`	Отношение длины окружности к ее диаметру (приблизительно 3,142)
`Math.SQRT1_2`	Величина, обратная квадратному корню из 2 (приблизительно 0,707)
`Math.SQRT2`	Квадратный корень из 2 (приблизительно 1,414)

Класс *Mouse*. Контроль мыши

Метод	Описание
`Mouse.addListener()`	Регистрирует объект-приемник на принятие сообщений о событиях от объекта `Mouse`
`Mouse.hide()`	Отключает отображение стандартного курсора мыши
`Mouse.removeListener()`	Отключает объект-приемник
`Mouse.show()`	Включает отображение стандартного курсора мыши
Событие	**Описание**
`Mouse.onMouseDown`	Событие генерируется при нажатии левой кнопки мыши в пределах окна проигрывателя
`Mouse.onMouseMove`	Событие генерируется периодически во время перемещения курсора внутри окна проигрывателя
`Mouse.onMouseUp`	Событие генерируется при отпускании левой кнопки мыши в пределах окна проигрывателя
`Mouse.onMouseWheel`	Событие генерируется при прокрутке колеса мыши

Класс *MovieClip*. Работа с клипами

Метод	Описание
`MovieClip.attachMovie()`	Динамически создает экземпляры символов, находящихся в библиотеке документа
`MovieClip.createEmptyMovieClip()`	Создает новый пустой клип непосредственно в процессе воспроизведения фильма
`MovieClip.createTextField()`	Создает пустое текстовое поле
`MovieClip.duplicateMovieClip()`	Динамически создает копии клипов, присутствующих на сцене
`MovieClip.getBounds()`	Возвращает объект с информацией о координатах ограничивающей рамки клипа
`MovieClip.getBytesLoaded()`	Возвращает количество загруженных байтов клипа или основной монтажной линейки
`MovieClip.getBytesTotal()`	Возвращает общее количество байтов, занимаемое объектом
`MovieClip.getDepth()`	Возвращает номер уровня стека (глубину), на котором расположен вызвавший этот метод экземпляр клипа

(продолжение)

Метод	Описание
`MovieClip.getInstanceAtDepth()`	Возвращает ссылку на клип, расположенный на уровне стека, номер которого указан в качестве параметра. Если этот уровень не содержит клипа, возвращается значение `undefined`
`MovieClip.getNextHighestDepth()`	Возвращает номер ближайшего свободного уровня стека, расположенного над всеми занятыми уровнями стека объекта
`MovieClip.getSWFVersion()`	Возвращает номер версии Flash Player, в которой был опубликован клип
`MovieClip.getTextSnapshot()`	Возвращает объект `TextSnapshot`, содержащий статический текст из клипа
`MovieClip.getURL()`	Загружает заданную HTML-страницу в окно браузера или отправляет запрос серверному сценарию
`MovieClip.globalToLocal()`	Позволяет выразить координаты точки, заданные в координатах основной монтажной линейки, в локальных координатах клипа
`MovieClip.gotoAndPlay()`	Переводит воспроизводящую головку на заданный кадр монтажной линейки клипа и продолжает воспроизведение клипа
`MovieClip.gotoAndStop()`	Переводит воспроизводящую головку на заданный кадр и останавливает воспроизведение клипа
`MovieClip.hitTest()`	Обнаруживает пересечение ограничивающих рамок клипов или попадание точки с заданными координатами в область, занимаемую клипом
`MovieClip.loadMovie()`	Загружает внешний файл в формате SWF или JPEG
`MovieClip.loadVariables()`	Загружает в Flash-фильм переменные из внешнего источника
`MovieClip.localToGlobal()`	Позволяет выразить координаты точки, заданные в системе координат клипа в глобальные координаты основной монтажной линейки
`MovieClip.nextFrame()`	Переводит воспроизводящую головку в следующий кадр монтажной линейки клипа и останавливает ее воспроизведение
`MovieClip.play()`	Запускает воспроизведение клипа
`MovieClip.prevFrame()`	Переводит воспроизводящую головку в предыдущий кадр монтажной линейки клипа и останавливает ее воспроизведение

(продолжение)

Метод	Описание
`MovieClip.removeMovieClip()`	Позволяет удалять экземпляры клипов, расположенные на уровнях стека с положительными номерами
`MovieClip.setMask()`	Осуществляет динамическое маскирование клипа, используя в качестве маски заданный клип
`MovieClip.startDrag()`	Инициирует перетаскивание клипа вслед за курсором мыши
`MovieClip.stop()`	Останавливает воспроизведение монтажной линейки клипа
`MovieClip.stopDrag()`	Прекращает перетаскивание клипа, инициированное вызовом метода `startDrag()`
`MovieClip.swapDepths()`	Меняет местами значения уровней двух клипов, изменяя их взаиморасположение в стеке, или помещает клип на заданный уровень
`MovieClip.unloadMovie()`	Удаляет SWF-файл, загруженный с помощью методов `loadMovie()` или `MovieClipLoader.loadClip()`
Методы динамического рисования	**Описание**
`MovieClip.beginFill()`	Применяет сплошную заливку для заполнения динамически создаваемого контура
`MovieClip.beginGradientFill()`	Применяет градиентную заливку для динамически создаваемого контура
`MovieClip.clear()`	Удаляет все графические объекты, созданные с помощью методов динамического рисования в стеке объекта, вызвавшего данный метод
`MovieClip.curveTo()`	Рисует криволинейный сегмент
`MovieClip.endFill()`	Прекращает действие методов `beginFill()` или `beginGradientFill()`
`MovieClip.lineStyle()`	Задает параметры линии, создаваемой методами динамического рисования
`MovieClip.lineTo()`	Рисует прямолинейный отрезок
`MovieClip.moveTo()`	Устанавливает начальную позицию рисования
Свойство	**Описание**
`MovieClip._alpha`	Позволяет контролировать степень прозрачности клипа

(продолжение)

Свойство	Описание
MovieClip._currentframe	Возвращает номер текущего кадра монтажной линейки объекта
MovieClip._droptarget	Возвращает строку, содержащую абсолютный путь к целевому объекту, над которым находился перетаскиваемый объект в момент прекращения перетаскивания
MovieClip.enabled	Активизирует или отключает работу клипа-кнопки или обычной кнопки
MovieClip.focusEnabled	Позволяет экземпляру клипа принимать фокус
MovieClip._focusrect	Определяет, включен ли режим, при котором клип, находящийся в фокусе, подсвечивается желтой рамкой
MovieClip._framesloaded	Возвращает количество загруженных кадров основной монтажной линейки или клипа
MovieClip._height	Определяет размер клипа по вертикали (в пикселах)
MovieClip.hitArea	Определяет тип поведения клипа-кнопки (или обычной кнопки)
MovieClip._highquality	Определяет степень сглаживания при воспроизведении фильма
MovieClip._lockroot	Определяет, будет ли идентификатор _root в динамически загруженном в клип SWF-документе возвращать ссылку на его собственную основную монтажную линейку или на основную монтажную линейку принимающего документа
MovieClip.menu	Связывает объект ContextMenu с экземпляром клипа
MovieClip._name	Хранит имя экземпляра клипа в виде строки
MovieClip._parent	Возвращает ссылку на родительский клип
MovieClip._rotation	Определяет угол поворота клипа
MovieClip._soundbuftime	Возвращает количество секунд перед началом воспроизведения потокового звука
MovieClip.tabChildren	Определяет, включены ли вложенные экземпляры клипа в автоматический порядок переноса фокуса

(продолжение)

Свойство	Описание
MovieClip.tabEnabled	Определяет, включен ли экземпляр клипа в автоматический порядок переноса фокуса
MovieClip.tabIndex	Определяет индекс клипа в используемом порядке переноса фокуса
MovieClip._target	Содержит строку, представляющую абсолютный путь к объекту, вызвавшему данное свойство
MovieClip._totalframes	Возвращает количество кадров на монтажной линейке объекта
MovieClip.trackAsMenu	Определяет тип поведения клипа-кнопки
MovieClip._url	Содержит URL документа SWF, в котором создан клип
MovieClip.useHandCursor	Определяет вид курсора при наведении на клип-кнопку
MovieClip._visible	Позволяет отключить отображение клипа
MovieClip._width	Задает размер клипа по горизонтали (в пикселах)
MovieClip._x	Координата клипа по горизонтали
MovieClip._xmouse	Возвращает координату курсора мыши по горизонтали
MovieClip._xscale	Определяет ширину клипа в процентном соотношении к его исходной ширине
MovieClip._y	Координата клипа по вертикали
MovieClip._ymouse	Возвращает координату курсора мыши по вертикали
MovieClip._yscale	Определяет высоту клипа в процентном соотношении к его исходной высоте
Обработчик события	**Описание**
MovieClip.onData	Вызывается при загрузке внешних данных в фильм
MovieClip.onDragOut	Вызывается, когда курсор выходит за пределы области реагирования клипа при нажатой кнопке мыши
MovieClip.onDragOver	Вызывается, когда курсор выходит за пределы области реагирования клипа при нажатой кнопке мыши, после чего вновь возвращается в область реагирования, кнопка мыши остается нажатой

(продолжение)

Обработчик события	Описание
MovieClip.onEnterFrame	Вызывается периодически с частотой, равной скорости воспроизведения, используемой в документе
MovieClip.onKeyDown	Вызывается при обнаружении нажатия любой клавиатурной клавиши
MovieClip.onKeyUp	Вызывается при отпускании нажатой клавиатурной клавиши
MovieClip.onKillFocus	Вызывается, когда фокус переходит с текущего клипа на другой объект
MovieClip.onLoad	Вызывается, как только клип впервые появляется на сцене, то есть сразу после его загрузки
MovieClip.onMouseDown	Вызывается при каждом нажатии левой кнопки мыши во время нахождения курсора в пределах окна проигрывателя
MovieClip.onMouseMove	Вызывается непрерывно во время перемещения курсора мыши в пределах окна проигрывателя Flash Player независимо от смены кадров монтажной линейки
MovieClip.onMouseUp	Вызывается всякий раз, когда левая кнопка мыши отпускается в момент нахождения курсора в пределах окна проигрывателя
MovieClip.onPress	Вызывается, когда кнопка мыши нажата при нахождении курсора в области реагирования клипа
MovieClip.onRelease	Вызывается, когда кнопка мыши нажата и затем отпущена при нахождении курсора в области реагирования клипа
MovieClip.onReleaseOutside	Вызывается, когда курсор выходит за пределы области реагирования клипа при нажатой кнопке мыши, после чего кнопка мыши отпускается вне области реагирования
MovieClip.onRollOut	Вызывается, когда курсор выходит за пределы области реагирования клипа при ненажатой кнопке мыши
MovieClip.onRollOver	Вызывается, когда курсор входит в область реагирования клипа

(окончание)

Обработчик события	Описание
`MovieClip.onSetFocus`	Вызывается, когда происходит передача фокуса текущему клипу
`MovieClip.onUnload`	Вызывается при удалении клипа из фильма

Класс *MovieClipLoader*. Управление процессом загрузки внешних документов и изображений

Метод	Описание
`MovieClipLoader.addListener()`	Регистрирует объект-приемник на принятие сообщений о событиях от объекта `MovieClipLoader`
`MovieClipLoader.getProgress()`	Возвращает загруженный объем информации на текущий момент времени
`MovieClipLoader.loadClip()`	Инициирует загрузку внешнего файла
`MovieClipLoader.removeListener()`	Отключает объект-приемник
`MovieClipLoader.unloadClip()`	Прерывает загрузку внешнего документа или выгружает полностью загруженный документ из клипа или с уровня стека
Событие	**Описание**
`MovieClipLoader.onLoadComplete()`	Событие генерируется после того, как внешний файл будет полностью загружен и записан на диск
`MovieClipLoader.onLoadError()`	Событие генерируется в случае невозможности выполнения загрузки или в результате ее прерывания
`MovieClipLoader.onLoadInit()`	Событие генерируется после инициализации загруженного клипа, то есть после того, как был выполнен весь программный код в его первом кадре
`MovieClipLoader.onLoadProgress()`	Событие периодически генерируется в ходе процесса загрузки при получении и сохранении на диске очередной порции загружаемой информации
`MovieClipLoader.onLoadStart()`	Событие генерируется, если метод `loadClip()` начал загружать внешний документ

Класс *NetConnection*. Установка локального соединения для динамической загрузки FLV-файлов

Метод	Описание
`NetConnection.connect()`	Открывает соединение, через которое будет выполняться передача FLV-файла из Интернета или из локальной файловой системы

Класс *NetStream*. Динамическая загрузка и воспроизведение FLV-файлов

Метод	Описание
`NetStream.close()`	Прекращает загрузку и воспроизведение FLV-файла, выполняемые с помощью объекта `NetStream`
`NetStream.pause()`	Останавливает или продолжает воспроизведение видео и синхронизированного с ним звука
`NetStream.play()`	Запускает загрузку и воспроизведение FLV-файла
`NetStream.seek()`	Позволяет переместить текущую позицию воспроизведения к заданному в секундах от начала воспроизведения моменту времени
`NetStream.setBufferTime()`	Устанавливает заданный размер буфера
Свойство	**Описание**
`NetStream.bufferLength`	Возвращает продолжительность видео (звука), находящегося в буфере в данный момент
`NetStream.bufferTime`	Возвращает размер буфера, заданный с помощью метода `setBufferTime()`
`NetStream.bytesLoaded`	Возвращает количество байт видеофайла, загруженных на момент вызова данного метода
`NetStream.bytesTotal`	Возвращает общий объем видеофайла
`NetStream.currentFps`	Возвращает количество кадров в секунду (видеоролика) на текущий момент
`NetStream.time`	Возвращает значение текущей позиции воспроизведения для динамически загруженного FLV-файла

(окончание)

Обработчик события	Описание
`NetStream.onStatus`	Вызывается в момент появления изображения в области видеообъекта

Класс *PrintJob*. Печать из Flash-фильма

Метод	Описание
`PrintJob.start()`	Выполняет подготовку к печати и приводит к появлению диалогового окна с настройками параметров печати (формат и тип бумаги, ориентация, цветность и т. д.) текущего принтера
`PrintJob.addPage()`	Формирует страницу для вывода на печать и отправляет ее в спулер печати
`PrintJob.send()`	Выводит на печать страницы, находящиеся в спулере печати

Класс *Selection*. Контроль выделения и передачи фокуса

Метод	Описание
`Selection.addListener()`	Регистрирует объект-приемник на принятие сообщений о событии `onSetFocus`, генерируемом объектом `Selection`
`Selection.getBeginIndex()`	Возвращает индекс первого символа, входящего в диапазон выделения
`Selection.getCaretIndex()`	Возвращает текущее положение текстового курсора
`Selection.getEndIndex()`	Возвращает индекс последнего символа, входящего в диапазон выделения
`Selection.getFocus()`	Определяет, какой объект в данный момент находится в фокусе
`Selection.removeListener()`	Прекращает трансляцию события `onSetFocus` объектом `Selection` объекту-приемнику
`Selection.setFocus()`	Передает фокус экземпляру текстового блока, кнопке или клипу

(окончание)

Событие	Описание
`Selection.setSelection()`	Выполняет выделение заданного диапазона текста в текстовом блоке, находящемся в фокусе
`Selection.onSetFocus`	Событие генерируется всякий раз при переходе фокуса с одного объекта на другой (текстовый блок, кнопка или клип) или в случае утраты фокуса

Класс *SharedObject*. Хранение информации на стороне клиента

Метод	Описание
`SharedObject.clear()`	Удаляет локальный объект с диска пользователя
`SharedObject.flush()`	Записывает текущий локальный объект `SharedObject` на диск
`SharedObject.getLocal()`	Создает текущий объект `SharedObject`
`SharedObject.getSize()`	Возвращает объем в байтах, занимаемый текущим объектом `SharedObject`
Свойство	**Описание**
`SharedObject.data`	Содержит ссылку на объект, свойства которого используются для локального сохранения данных и позволяют считывать данные из локально сохраненного объекта `SharedObject`
Обработчик события	**Описание**
`SharedObject.onStatus`	Генерируется при выборе пользователем в диалоговом окне **Macromedia Flash Player Settings** одной из двух опций **Allow** — разрешить или **Deny** — запретить увеличение объема дискового пространства для хранения локальных объектов

Класс *Sound*. Управление звуком

Метод	Описание
`Sound.attachSound()`	Присоединяет указанный звук, расположенный в библиотеке, к объекту, звуковое содержимое которого контролируется объектом `Sound`

(окончание)

Метод	**Описание**
`Sound.getBytesLoaded()`	Возвращает загруженный на момент вызова данного метода объем звукового файла в байтах
`Sound.getBytesTotal()`	Возвращает объем загружаемого в объект `Sound` звука в байтах
`Sound.getPan()`	Возвращает значение баланса для вызвавшего данный метод объекта `Sound`
`Sound.getTransform()`	Возвращает значения свойств объекта трансформации, установленных для конкретного объекта `Sound`
`Sound.getVolume()`	Возвращает текущее значение громкости, установленное для вызвавшего его объекта `Sound`
`Sound.loadSound()`	Динамически загружает внешний MP3-файл
`Sound.setPan()`	Управляет балансом громкости левого/правого каналов
`Sound.setTransform()`	Управляет выводом сигналов для левого и правого каналов
`Sound.setVolume()`	Устанавливает уровень (громкости) звука
`Sound.start()`	Запускает воспроизведение динамически присоединенного или загруженного из внешнего файла звука
`Sound.stop()`	Останавливает воспроизведение звука
Свойства	**Описание**
`Sound.duration`	Возвращает длительность присоединенного или загруженного данным объектом звука в миллисекундах
`Sound.ID3`	Предоставляет доступ к ID3-тегам динамически загруженного MP3-файла
`Sound.position`	Определяет текущую позицию воспроизведения звука
Событие	**Описание**
`Sound.onID3`	Событие генерируется после окончания загрузки ID3-тегов
`Sound.onLoad`	Событие генерируется объектом `Sound` в момент завершения загрузки звукового файла или в результате ее прекращения по тем или иным причинам
`Sound.onSoundComplete`	Событие генерируется при завершении динамически присоединенных или загруженных звуков

Класс *Stage*. Контроль параметров документа

Метод	Описание
`Stage.addListener()`	Регистрирует объект-приемник на получение сообщений от объекта `Stage`
`Stage.removeListener()`	Отключает объект-приемник
Свойства	**Описание**
`Stage.align`	Устанавливает параметры выравнивания фильма относительно границ окна проигрывателя Flash Player
`Stage.height`	Возвращает высоту сцены в пикселах
`Stage.scaleMode`	Устанавливает режим масштабирования
`Stage.showMenu`	Включает или отключает стандартное контекстное меню Flash Player
`Stage.width`	Возвращает ширину сцены в пикселах
Событие	**Описание**
`Stage.onResize`	Событие генерируется при изменении размеров окна проигрывателя при условии, что текущим режимом масштабирования является режим `"noScale"`

Класс *StyleSheet*. Поддержка и использование CSS

Метод	Описание
`TextField.StyleSheet.clear()`	Очищает таблицу стилей, представленную объектом `StyleSheet`
`TextField.StyleSheet.getStyle()`	Возвращает объект, представляющий стиль, который содержит набор свойств, отражающих параметры форматирования, определенные для указанного стиля
`TextField.StyleSheet.getStyleNames()`	Возвращает массив, содержащий имена стилей (в виде строк), помещенных в таблицу стилей, представленную объектом `StyleSheet`
`TextField.StyleSheet.load()`	Загружает внешнюю таблицу стилей
`TextField.StyleSheet.parseCSS()`	Добавляет стиль, заданный в виде строки, в таблицу стилей

(окончание)

Метод	Описание
`TextField.StyleSheet.setStyle()`	Добавляет стиль, представленный объектом `Object`, в таблицу стилей
Событие	**Описание**
`TextField.StyleSheet.onLoad`	Событие генерируется непосредственно после выполнения операции, инициированной вызовом метода `load()`, то есть после полного окончания загрузки внешней таблицы стилей или, в случае прекращения загрузки, вследствие ее недоступности или возникшей ошибки

Класс *System*. Информация о системе и оборудовании

Метод	Описание
`System.setClipboard()`	Помещает в буфер обмена текстовую строку, замещая его текущее содержимое
`System.showSettings()`	Выводит стандартную панель настроек проигрывателя
Свойство	**Описание**
`System.exactSettings`	Определяет, какие правила политики безопасности будут использоваться при доступе к локальным данным
`System.useCodepage`	Указывает проигрывателю, должна ли при загрузке внешних текстовых данных использоваться кодировка операционной системы или `Unicode`
Событие	**Описание**
`System.onStatus`	Универсальное событие генерируется объектом `System` в результате различных процессов, связанных с объектами `LocalConnection`, `NetStream`, `SharedObject`

Класс *TextField*. Контроль состояния и параметры текстового блока

Метод	Описание
`TextField.addListener()`	Регистрирует объект-приемник на получение уведомления о наступлении события, обнаруживаемого объектом `TextField`

(продолжение)

Метод	Описание
`TextField.getFontList()`	Возвращает массив, содержащий перечень названий всех шрифтов, установленных на компьютере
`TextField.getDepth()`	Возвращает номер уровня стека (глубину), на котором расположен вызвавший этот метод текстовый блок
`TextField.getNewTextFormat()`	Возвращает информацию о параметрах форматирования вновь добавленного или введенного пользователем шрифта
`TextField.getTextFormat()`	Возвращает объект `TextFormat`, содержащий информацию о параметрах форматирования отдельного символа, диапазона символов или всего текущего содержимого текстового блока
`TextField.removeListener()`	Отключает объект-приемник `MoveClip createTextField()`
`TextField.removeTextField()`	Удаляет текстовый блок, который был создан методом
`TextField.replaceSel()`	Заменяет выделенный диапазон символов указанным текстом
`TextField.setNewTextFormat()`	Задает форматирование, которое по умолчанию применяется к новому тексту, вводимому в текстовый блок пользователем или средствами ActionScript
`TextField.setTextFormat()`	Позволяет применить настройки форматирования, определенные в объекте `TextFormat`, к текущему содержимому указанного текстового блока
Свойство	**Описание**
`TextField._alpha`	Задает величину прозрачности текстового блока
`TextField.autoSize`	Автоматически подгоняет размеры текстового блока под выводимое в нем содержимое
`TextField.background`	Включает/отключает отображение фона текстового блока
`TextField.backgroundColor`	Задает цвет фона текстового блока
`TextField.border`	Включает/отключает отображение рамки вокруг текстового блока
`TextField.borderColor`	Задает цвет рамки текстового блока
`TextField.bottomScroll`	Возвращает номер самой нижней видимой строки текстового блока
`TextField.embedFonts`	Определяет, используется ли в текстовом блоке встроенный шрифт или системный шрифт

(продолжение)

Свойство	Описание
`TextField._height`	Задает высоту текстового блока в пикселах. Не влияет на толщину рамки и размеров символов шрифта
`TextField._highquality`	Задает степень сглаживания при воспроизведении фильма
`TextField.hscroll`	Определяет ширину текста, который срезается левой границей текстового блока
`TextField.html`	Определяет, используется ли в текстовом блоке форматирование с применением HTML
`TextField.htmlText`	Описывает содержимое текстового блока с применением HTML-форматирования
`TextField.length`	Возвращает количество символов, находящихся в текстовом блоке
`TextField.maxChars`	Задает максимальное количество символов, которое пользователь сможет ввести в пользовательское текстовое поле
`TextField.maxhscroll`	Определяет, какая часть текста должна быть срезана левой границей текстового блока для того, чтобы в текстовом блоке был виден правый конец строки
`TextField.maxscroll`	Возвращает максимальное значение, которое может принимать свойство `scroll` для указанного текстового блока
`TextField.menu`	Связывает объект `ContextMenu` с экземпляром текстового блока
`TextField.mouseWheelEnabled`	Определяет, разрешено ли выполнение прокрутки с помощью колеса мыши
`TextField.multiline`	Определяет, может ли пользователь при вводе текста в пользовательский текстовый блок выполнять перенос строки
`TextField._name`	Содержит имя экземпляра текстового блока в виде строки
`TextField._parent`	Возвращает ссылку на родительский клип
`TextField.password`	Устанавливает режим пароля, при котором все символы отображаются в текстовом поле звездочками (`*`)
`TextField._quality`	Задает качество отображения графики в документе (глобальное свойство)
`TextField.restrict`	Позволяет указать произвольный набор символов, которые могут быть введены пользователем в текстовый блок типа **Input**

(продолжение)

Свойство	Описание
TextField._rotation	Задает угол поворота текстового блока в градусах
TextField.scroll	Определяет номер самой верхней видимой строки текстового блока
TextField.selectable	Определяет, может ли текст, находящийся в текстовом блоке, выделяться пользователем
TextField.tabEnabled	Включает обьект в автоматический порядок переноса фокуса
TextField.tabIndex	Задает индекс, определяющий порядок переноса фокуса по табуляции
TextField._target	Возвращает абсолютный путь к текстовому блоку
TextField.text	Принимает строковое значение, определяющее содержимое текстового блока
TextField.textColor	Задает цвет текста
TextField.textHeight	Фактическая высота в пикселах текста, расположенного в текстовом блоке
TextField.textWidth	Фактическая ширина в пикселах текста, расположенного в текстовом блоке
TextField.type	Указывает тип текстового блока — динамический или пользовательский
TextField._url	Содержит URL документа SWF, в котором создан текстовый блок
TextField.variable	Принимает строковое значение, определяющее переменную, сопоставленную данному блоку
TextField._visible	Включает/отключает отображение текстового блока
TextField._width	Определяет ширину текстового блока в пикселах. Не влияет на толщину рамки и размеры символов шрифта
TextField.wordWrap	Включает/отключает автоматический перенос текста на следующую строку
TextField._x	Определяет координату текстового блока по горизонтали
TextField._xmouse	Возвращает текущую координату курсора мыши по горизонтали относительно левого верхнего угла текстового блока
TextField._xscale	Определяет ширину текстового блока в процентном отношении к его исходной ширине
TextField._y	Определяет координату текстового блока по вертикали

(окончание)

Свойство	Описание
`TextField._ymouse`	Возвращает текущую координату курсора мыши по вертикали относительно левого верхнего угла текстового блока
`TextField._yscale`	Определяет высоту текстового блока в процентном отношении к его исходной высоте
Обработчик события	**Описание**
`TextField.onChanged`	Вызывается при изменении содержимого пользовательского текстового блока в результате действий пользователя
`TextField.onKillFocus`	Вызывается при переходе фокуса с экземпляра текстового блока на другой объект (или при утрате фокуса)
`TextField.onScroller`	Вызывается при выполнении прокрутки содержимого текстового блока (то есть в результате изменения значения одного из свойств, связанных с прокруткой: `scroll`, `maxscroll`, `bottomScroll`, `hscroll` или `maxhscroll`)
`TextField.onSetFocus`	Вызывается при передаче фокуса экземпляру текстового блока
Событие	**Описание**
`TextField.onChanged`	Событие генерируется при изменении содержимого текстового блока, выполняемом в результате действий пользователя
`TextField.onScroller`	Событие генерируется, когда происходит изменение значения свойств `scroll`, `maxscroll`, `bottomScroll`, `hscroll` или `maxhscroll`

Класс *TextFormat*. Динамическое форматирование текста

Метод	Описание
`TextFormat.getTextExtent()`	Возвращает информацию о метриках шрифта с учетом применяемого к нему форматирования
Свойство	**Описание**
`TextFormat.align`	Устанавливает выравнивание текста относительно боковых границ текстового блока

(окончание)

Свойство	Описание
`TextFormat.blockIndent`	Задает отступ всех строк текста от левой границы текстового блока
`TextFormat.bold`	Отображает текст с применением жирного начертания
`TextFormat.bullet`	Отображает содержимое тестового блока в виде маркированного списка
`TextFormat.color`	Изменяет цвет текста
`TextFormat.font`	Задает шрифт, используемый при выводе и наборе текста в текстовом блоке
`TextFormat.indent`	Задает отступ красной строки, то есть расстояние от левого поля текстового блока до первого символа абзаца
`TextFormat.italic`	Отображает текст с применением курсивного начертания
`TextFormat.leading`	Устанавливает интерлиньяж, то есть промежуток между строками текста, расположенного в текстовом блоке
`TextFormat.leftMargin`	Устанавливает ширину левого поля в пикселах
`TextFormat.rightMargin`	Устанавливает ширину правого поля в пикселах
`TextFormat.size`	Устанавливает размер символов текста, выводимых в текстовом блоке
`TextFormat.tabStops`	Устанавливает позицию табуляции
`TextFormat.target`	Задает целевое окно или фрейм, куда будет осуществляться загрузка сетевого ресурса
`TextFormat.underline`	Отображает текст с применением подчеркивания
`TextFormat.url`	Указывает URL, с которым связан текст, используемый в качестве гиперссылки

Класс *TextSnapShot*. Контроль статического текста

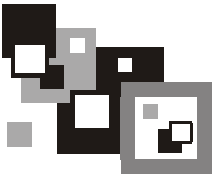
Метод	Описание
`TextSnapshot.findText()`	Указывает первое вхождение заданного набора символов в строку
`TextSnapshot.getCount()`	Возвращает информацию о количестве символов в объекте `TextSnapshot`

(окончание)

Метод	Описание
`TextSnapshot.getSelected()`	Определяет, имеется ли в заданном диапазоне хотя бы один выделенный символ
`TextSnapshot.getSelectedText()`	Извлекает выделенный текст
`TextSnapshot.getText()`	Возвращает текст, помещенный в статический текстовый блок клипа
`TextSnapshot.hitTestTextNearPos()`	Возвращает местоположение курсора относительно символа текста
`TextSnapshot.setSelectColor()`	Задает цвет выделения статического текста
`TextSnapshot.setSelected()`	Устанавливает или снимает выделение статического (невыделяемого) текста

Класс *Video*. Работа с видеообъектами

Метод	Описание
`Video.attachVideo()`	Связывает видеообъект с конкретным видео, которое должно выводиться в занимаемой им области
`Video.clear()`	Очищает область видеообъекта от выводимого в текущий момент изображения, не удаляя при этом сам видеообъект
Свойство	**Описание**
`Video.height`	Возвращает высоту видеообъекта в пикселах
`Video.smoothing`	Включает/отключает сглаживание видео
`Video.width`	Возвращает ширину видеообъекта в пикселах



Приложение 2

Описание компакт-диска

На компакт-диске название папки соответствует номеру главы книги, а имя исходного файла — порядковому номеру листинга в тексте. В тексте все программные коды, исходные файлы с которыми читатель сможет найти на диске, выделены заголовками Листинг N.N. с названием примера (например, Листинг 7.1. Кнопка-флаг). В главах 18, 25—27 названия ряда исходных файлов внешних функций и классов соответствуют их именам. Такие случаи в тексте оговорены особо. Каждому FLA-файлу, имеющему графическое представление, соответствует файл SWF-документ.

FLA-файлы, для которых на диске нет SWF-документов, предназначены для просмотра в среде тестирования Flash.

Внешние функции, которые загружаются с помощью директивы #include, а также внешние классы, сохранены в файлах с расширением as.

Для некоторых файлов представлены HTML-страницы с размещенными на них SWF-документами. Для некоторых примеров созданы FLP-файлы проекта, включающие в себя все файлы, относящиеся к данному примеру. Для некоторых файлов созданы проекторы с расширением exe.

Название папки/ файла на диске	**Название главы/листинга**
Глава 7	**Управляющие структуры: Операторы ветвления. Циклы**
7_1.fla	Кнопка-флаг
7_2.fla	Проверка окончания испытательного срока
7_3.fla	Трехкадровый цикл. Перемещение внутри прямоугольной области
Глава 8	**Пользовательские функции**
8_1.fla	Функция для возведения числа в целую степень
8_2.fla	Функция — арифметический калькулятор
8_3.fla	Простой конвертер валют
8_4.fla	Движение за курсором при помощи функции `setInterval()`
8_5.fla	Применение функций `setInterval()` и `clearInterval()` для контроля анимации
8_6.fla	Использование рекурсивной функции для вычисления факториала числа

(продолжение)

Название папки/ файла на диске	**Название главы/листинга**
8_7.fla	Рекурсивный поиск индексов всех вхождений заданного набора символов в строку
Глава 9	**Массивы**
9_1.fla	Функция сравнения для сортировки дат по возрастанию
9_2.fla	Формирование матрицы с заданным числом строк и столбцов
9_3.fla	Транспонирование матрицы
Глава 10	**Класс `Button`. Работа с кнопками**
10_1.fla	Перехват событий кнопки
Глава 11	**Событийная модель ActionScript**
11_1.fla	Пользовательский курсор
11_2.fla	Применение методов обработчиков событий для программного контроля анимации прозрачности
11_3.fla	Контроль двойного щелчка мыши
Глава 12	**Класс `MovieClip`. Работа с клипами**
12—17	Модель реализации навигации посредством загрузки в стек проигрывателя внешних документов
pics	Растровые изображения для примера 12.18
12_1.fla	Приведение размеров двух объектов
12_2.fla	Плавное увеличение клипа при наведении курсора мыши
12_3.fla	Выполнение зеркального отражения объекта по горизонтали и/или по вертикали
12_4.fla	Анимация прозрачности
12_5.fla	Реализация интерфейса со сменой изображений при наведении на кнопки
12_6.fla	Реализация функциональности типа drag and drop
12_7.fla	Покадровый предварительный загрузчик
12_8.fla	Выпадающий список
12_9.fla	Управление качеством прорисовки графики
12_10.fla	Изменение скорости воспроизведения за счет пропускания кадров
12_11.fla	Управление воспроизведением основной монтажной линейки при помощи кнопок
12_12.fla	Программный реверс анимации
12_13.fla	Создание регулярного заполнения на основе динамического присоединения экземпляров
12_14.fla	Анимация на основе копирования и вращения
12_15.fla	Модель игры на основе динамического создания и удаления экземпляров

(продолжение)

Название папки/ файла на диске	Название главы/листинга
12_16.fla	Графический предзагрузчик
12_18.fla	Интерактивное слайд-шоу с динамической загрузкой изображений
12_19.fla	Установка равных промежутков между клипами с произвольными размерами и расположением точки регистрации, находящимися на разных монтажных линейках
12_20.fla	Контроль попадания объекта в заданную область с применением свойства `_droptarget`
12_21.fla	Модель простой игры с мячом и битой
12_22.fla	Контроль попадания объекта в заданную область с применением метода `hitTest()`
Глава 13	**Динамическое рисование**
13_1.fla	Создание инструмента "карандаш"
13_2.fla	Функция для рисования правильных выпуклых многоугольников и звезд
13_3.fla	Бесконечное движение линейного градиента
13_4.fla	Создание градиента с помощью матрицы общего преобразования
13_5.fla	Динамическое управление кривыми
13_6.fla	Эффект линзы с применением динамического маскирования
Глава 14	**Применение динамического и пользовательского текста**
Css	Css-документы, которые используются в ряде примеров
images	Растровые изображения, которые содержатся в ряде примеров
swf	SWF-документы, которые используются в ряде примеров
xml	XML-документы, которые используются в ряде примеров
14_1.fla	Создание всплывающей подсказки
14_2.fla	Проверка наличия у пользователя заданного шрифта
14_3.fla	Вывод текста бегущей строкой
14_4.fla	Организация непрерывной построчной прокрутки текста
14_5.fla	Создание полосы прокрутки текста
14_6.fla	Проверка наличия незаполненных текстовых полей в форме
14_7.fla	Выделение текстового поля с фокусом прямоугольной рамкой
14_8.fla	Определение границ выделения и положения курсора
14_9.fla	Создание текстового поля для ввода данных с автоматической подсказкой
14_10.fla	Изменение регистра выделенного диапазона текста с помощью кнопок
14_11.fla	Вывод координат мыши в текстовое поле

(продолжение)

Название папки/ файла на диске	**Название главы/листинга**
14_12.fla	Обработка событий объектов, внедренных в текстовый блок
14_13.fla	Применение стилей, определенных с помощью объектов `Object`
14_14.fla	Применение стилей, определенных с помощью строки
14_15.fla	Применение внешней таблицы стилей
14_16.fla	Вывод информации о стилях, помещенных в таблицу стилей
14_17.fla	Вывод содержимого XML-документа, отформатированного с применением внешней таблицы стилей
14_18.fla	Организация выделения и копирования статического текста с помощью мыши
Глава 15	**Класс `Math`**
15_1.fla	Определение наибольшего и наименьшего из набора значений
15_2.fla	"Розы" Гвидо Гранди_ Эффект перемещения с формированием шлейфа
15_3.fla	Поворот объекта в направлении курсора
15_4.fla	Эффект падающего снега
Глава 16	**Класс `Date`. Контроль времени и даты**
16_1.fla	Часы
16_2.fla	Календарь
16_3.fla	Таймер
Глава 17	**Классы `Key` и `Mouse`. Контроль клавиатуры и мыши**
17_1.fla	Шлейф мыши
17_2.fla	Управление объектом с помощью клавиатуры (I)
17_3.fla	Определение кода нажатой клавиши
17_4.fla	Управление объектом с помощью клавиатуры (II)
17_5.fla	Перемещение клипа при помощи клавиатуры с учетом инерции и контроля ускорения
Глава 18	**Класс `Color`. Работа с цветом**
RGBtoHSB.as	Перевод значения цвета из модели RGB в модель HSB
HSBtoRGB.as	Перевод значения цвета из модели HSB в модель RGB
18_3.fla	Анимация перехода цвета по окружности цветового круга
web216.as	Формирование цветовых значений "безопасной" Web-палитры
18_5.fla	Формирование каталога цветовых образцов "безопасной" Web-палитры, ориентированных по тону
18_6.fla	Управление цветовой трансформацией
18_7.fla	Создание панели синтеза цвета на основе модели HSB, представленной в виде цветового круга

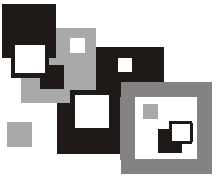
(продолжение)

Название папки/ файла на диске	**Название главы/листинга**
18_8.fla	Применение панелей синтеза цвета для создания цветового редактора
colorMixer.fla	FLA-файл, который динамически подгружается в примере 18_8
Глава 19	**Класс `Sound`. Управление звуком**
19_1.fla	Бесконечное зацикливание звука
19_2.fla	Создание кнопки воспроизведения/паузы присоединенного звука
19_3.fla	Предварительный загрузчик потокового звука
19_4.fla	Создание эффекта эха
19_5.fla	Управление балансом каналов
19_6.fla	Управление выводом звуковых каналов на динамики
19_7.fla	Интерактивная перкуссия
Ebony_Samba.mp3	Используется в ряде примеров
Глава 20	**Управление процессом загрузки внешних документов и изображений. Взаимодействие между SWF-документами**
20_1.fla	Предзагрузчик с возможностью локального тестирования
20_2.fla	Управление загрузкой нескольких документов
20_3	Интерактивный навигатор
20_4	Организация взаимодействия между файлами
content	Документы, которые используются в ряде примеров
Глава 21	**Класс `SharedObject`. Хранение информации на стороне клиента**
media	Документы, которые используются в примере
21_1.fla	Локальный счетчик посещений
21_2.fla	Пример хранения пользовательских настроек интерфейса
Глава 22	**Контроль параметров документа и проигрывателя. Информация о системе и оборудовании**
22_1.fla	Установка режима масштабирования фильма
22_2.fla	Установка режима выравнивания фильма
22_3.fla	Управление размещением объекта при масштабировании окна проигрывателя
22_4.fla	Затенение содержимого сцены при выводе контекстного меню
22_5.fla	Дублирование и удаление объектов при помощи команд контекстного меню
22_6.fla	Управление анимацией клипа при помощи контекстного меню

(окончание)

Название папки/ файла на диске	Название главы/листинга
Глава 23	**Класс `PrintJob`. Печать из Flash-фильма**
23_1.fla	Вывод текущего кадра клипа на печать по нажатию кнопки
23_2.fla	Печать текущего кадра клипа с применением пропорционального масштабирования до размера печатной области листа
23_3.fla	Печать всех кадров монтажной линейки клипа
23_4.fla	Печать с помощью пользовательских элементов контекстного меню
Глава 24	**Динамическая загрузка и воспроизведение видео**
24_1.fla	Проигрыватель FLV-файла
Глава 25	**Создание пользовательских классов**
RandomColor.as	Случайное позиционирование и окрашивание клипов. Перемещение по круговой траектории со случайной скоростью
Lift.as	Обеспечение базовой функциональности лифта
PassangerLift.as	Реализация регулирования скорости лифта и возможности его вызова на заданный этаж
randomColorClip	Сценарий кадра в FLA-файле, реализующий графический интерфейс и визуализацию системы лифтов
Глава 26	**Трехмерное моделирование и анимация**
Calculus.fla	Математические операции, используемые для выполнения трехмерных преобразований
Vertex3D.fla	Трехмерные преобразования и визуализация точек
26_1.fla	Управление центрированным кубом со срезанным углом при помощи клавиатуры
26_2.fla	Управление трехмерной плоскостью с заданным числом точек при помощи мыши
Глава 27	**Создание пользовательских компонентов**
DustAndScratches.as	Управляющий класс компонента DustAndScratches
AnimateDust.as	Реализация поведения пылинок
AnimateScratch.as	Реализация поведения царапин
DustAndScratches.mxi	MXI-файл, используемый для упаковки расширения
Глава 28	**Обработка исключительных ситуаций. Отладка сценариев. Тестирование проекта**
28_1.fla	Возбуждение исключительной ситуации из блока предложения `try`
28_2.fla	Возбуждение исключительной ситуации из тела функции
28_3.fla	Обработка нескольких исключений единственным обработчиком
28_4.fla	Рекурсивная обработка нескольких исключительных ситуаций разными обработчиками

Приложение 3



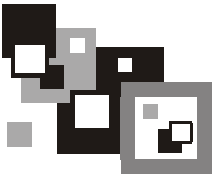
Виртуальные коды клавиш

Клавиша	Код клавиши	Клавиша	Код клавиши	Клавиша	Код клавиши
Клавиши букв и цифр					
<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
Специальные клавиши					
<Numbpad 0>	96	<F5>	116	<Page Down>	34
<Numbpad 1>	97	<F6>	117	<End>	35
<Numbpad 2>	98	<F7>	118	<Home>	36
<Numbpad 3>	99	<F8>	119	<Left Arrow>	37
<Numbpad 4>	100	<F9>	120	<Up Arrow>	38
<Numbpad 5>	101	<F10>	121	<Right Arrow>	39
<Numbpad 6>	102	<F11>	122	<Down Arrow>	40
<Numbpad 7>	103	<F12>	123	<Insert>	45

(окончание)

Клавиша	Код клавиши	Клавиша	Код клавиши	Клавиша	Код клавиши
Специальные клавиши					
<Numbpad 8>	104	<F13>	124	<Delete>	46
<Numbpad 9>	105	<F14>	125	<Help>	47
<Multiply>	106	<F15>	126	<Num Lock>	144
<Add>	107	<Clear>	12	; :	186
<Enter>	108	<Enter>	13	= +	187
<Subtract>	109	<Shift>	16	- _	189
<Decimal>	110	<Control>	17	/ ?	191
<Divide>	111	<Alt>	18	~	192
<F1>	112	<Caps Lock>	20	[{	219
<F2>	113	<Esc>	27	\ |	220
<F3>	114	<Spacebar>	32	] }	221
<F4>	115	<Page Up>	33	" '	222

Приложение 4

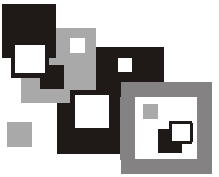


Горячие клавиши инструкций ActionScript

Инструкция	Горячие клавиши	Примечание
Глобальные функции. Управление монтажной линейкой		
`gotoAndPlay()`	<Esc>+<gp>	Переход в заданный кадр фильма/клипа и его воспроизведение
`gotoAndStop()`	<Esc>+<gs>	Переход в заданный кадр фильма/клипа и остановка его воспроизведения
`nextFrame()`	<Esc>+<nf>	Переход к следующему кадру
`nextScene()`	<Esc>+<ns>	Переход к следующей сцене
`play()`	<Esc>+<pl>	Воспроизведение фильма/клипа
`prevFrame()`	<Esc>+<ps>	Переход к предыдущему кадру
`stop()`	<Esc>+<st>	Остановка фильма/клипа
`stopAllSounds()`	<Esc>+<ss>	Остановка воспроизведения всех звуков
Глобальные функции загрузки данных и управления автономным проигрывателем		
`fscommand()`	<Esc>+<fs>	Управление автономным проектором
`getURL()`	<Esc>+<gu>	Загрузка сетевого ресурса в браузер
`loadMovie()`	<Esc>+<lm>	Загрузка внешнего фильма
`loadMovieNum()`	<Esc>+<ln>	Загрузка внешнего фильма на указанный уровень
`loadVariables()`	<Esc>+<lv>	Загрузка внешних переменных
`loadVariablesNum()`	<Esc>+<vn>	Загрузка внешних переменных на указанный уровень
`unloadMovie()`	<Esc>+<um>	Выгрузка внешнего фильма
`unloadMovieNum()`	<Esc>+<un>	Выгрузка внешнего фильма с указанного уровня

(окончание)

Инструкция	Горячие клавиши	Примечание
Глобальные функции. Управление мувиклипом		
`duplicateMovieClip()`	<Esc>+<dm>	Дублирование мувиклипа
`on()`	<Esc>+<on>	Обработчик события кнопки
`onClipEvent()`	<Esc>+<oc>	Обработчик события мувиклипа
`removeMovieClip()`	<Esc>+<rm>	Удаление мувиклипа
`setProperty()`	<Esc>+<sp>	Установка свойств мувиклипа
`startDrag()`	<Esc>+<dr>	Активизация перетаскивания мувиклипа
`stopDrag()`	<Esc>+<sd>	Остановка перетаскивания мувиклипа
Функции печати		
`print()`	<Esc>+<pr>	Вывод на печать
`printAsBitmap()`	<Esc>+<pb>	Вывод на печать с предварительным растрированием
`printAsBitmapNum()`	<Esc>+<bn>	Печать кадров уровня документа как растровых изображений
`printNum()`	<Esc>+<pn>	Печать кадров уровня документа как векторных изображений
Различные функции		
`trace()`	<Esc>+<tr>	Вывод сообщения в панель **Output**
Директивы компилятору		
`#initclip`	<Esc>+<ic>	Управление порядком выполнения блоков кода
`#endinitclip`	<Esc>+<ec>	
`#include ""`	<Esc>+<in>	Загрузка сценария из внешнего текстового документа
Устаревшие инструкции		
`call ()`	<Esc>+<ca>	Вызов подпрограммы
`ifFrameLoaded () {…}`	<Esc>+<il>	Проверка загрузки кадра
`tellTarget () {…}`	<Esc>+<tt>	Указание целевого пути
`toggleHighQuality()`	<Esc>+<tq>	Установка качества изображения



Приложение 5

Интернет-ресурсы, посвященные Flash

Ресурс	Примечание
www.actionscript.org	Англоязычный ресурс, содержащий огромное количество обучающих статей и уроков, освещающих разнообразные области практического применения ActionScript. Здесь также находится своеобразная международная биржа труда Flash-разработчиков
www.flasher.ru	Русскоязычный портал, содержащий большой учебный материал по различным аспектам работы с Flash: статьи, исходные документы, форум и др.
www.flashkit.com	Крупнейший источник информации по Flash. На сайте представлено все, что может иметь отношение к Flash (исходные файлы, примеры, учебники, новости, статьи, чат, галерея и др.)
www.flash-ripper.com	Русскоязычный ресурс, ориентирован на продвинутых Flash-разработчиков. Здесь регулярно публикуются свежие новости, имеющие отношение к Flash-технологии и охватывающие как русскоязычную, так и остальную часть Flash-сообщества. Сайт содержит большое количество полезных практических материалов: статей, практических советов, ссылок на другие ресурсы, утилит
www.macromedia.com	Официальный сайт компании Macromedia. Содержит большое количество информации о продуктах, выпускаемых этой компанией (в том числе и о Flash), и связанных с ними новостях. На этом сайте можно бесплатно скачать полнофункциональные демонстрационные версии продуктов (с ограниченным сроком действия), получить официальную документацию, которая регулярно обновляется (возможно не так часто, как хотелось бы). Раздел **http://www.macromedia.com/go/flash_support** посвящен исключительно вопросам использования Flash. Здесь наряду с документацией находится множество статей, имеется форум, в котором можно задать интересующий вопрос. В разделе **http://www.macromedia.com/go/flash_exchange** можно найти множество готовых компонентов и расширений, созданных сторонними разработчиками

(окончание)

Ресурс	Примечание
www.multikov.net	Русскоязычный ресурс, целиком посвященный Flash-анимации. Этот сайт посвящен не программированию, а созданию анимации в Flash и, соответственно, ориентирован не на разработчиков, а на художников и дизайнеров. Содержит множество образцов работ и учебных материалов по созданию двумерной анимации
www.ultrashock.com	Англоязычный портал для разработчиков Flash. Содержит большое количество учебных примеров и исходных файлов, сосредоточенных на ActionScript. Здесь представлено множество интересных примеров, демонстрирующих создание разнообразных интерактивных эффектов, а также статьи и уроки, посвященные программированию на ActionScropt

Предметный указатель

_

__proto__, свойство 894
_alpha, свойство 304; 352; 501
_css, объект 582; 595
_currentframe, свойство класса MovieClip 359
_droptarget, свойство 372; 438
_focusrect, свойство 304; 374; 548
_framesloaded, свойство класса MovieClip 359
_global, объект 126
_height, свойство 304; 347; 501
_highquality, свойство 304; 371
_level, идентификатор 394
_listeners, скрытое свойство 338
_lockroot, свойство класса MovieClip 423
_lockroot, свойство класса MovieClip 373
_name, свойство 304; 353; 501
_parent, свойство 304; 501
_quality, глобальное свойство 304; 370
_rotation, свойство 304; 352; 501
_soundbuftime, свойство 304; 707
_styles, объект 583; 595
_target, свойство 304; 373; 501
_totalframes, свойство класса MovieClip 358
_url, свойство 304; 371; 501
_visible, свойство 304; 354; 501
_width, свойство 304; 347; 501
_x, свойство 304; 345; 501
_xmouse, свойство 304; 355
_xscale, свойство 304; 350; 501
_y, свойство 304; 345; 501
_ymouse, свойство 304; 355
_yscale, свойство 304; 350; 501
<a>, тег HTML 562
<b>, тег HTML, 561

, тег HTML 563
<font>, тег HTML 561
<i>, тег HTML 562
<img>, тег HTML 566
, тег HTML 564
<p>, тег HTML 563
, тег HTML 566
<textformat>, тег HTML 565
<u>, тег HTML 562

A

abs(), статический метод класса Math 615
acos(), статический метод класса Math 621
Actions, панель 78
ActionScript version, список для выбора версии языка 76
addEventListener(), метод класса EventDispatcher 989
addListener(), метод 334
addPage(), метод класса PrintJob 814
align, свойство класса TextFormat 514
align, статическое свойство класса Stage 782
allowDomain(), метод объекта System.security 743
allowDomain, обработчик события 756
allowInsecureDomain(), метод объекта System.security 743
allowInsecureDomain, обработчик события 758

apply(), метод класса Function 248
Arguments, класс 236
arguments, переменная 236
Array(), глобальная функция 265
Array(), конструктор 263
Array, класс 262
Array.CASEINSENSITIVE, константа класса Array 279
Array.DESCENDING, константа класса Array 279
Array.NUMERIC, константа класса Array 280
Array.RETURNINDEXEDARRAY, константа класса Array 280
Array.UNIQUESORT, константа класса Array 280
AsBroadcaster, глобальный объект 338
asfunction, протокол 562
asin(), статический метод класса Math 621
ASSetPropFlags(), глобальная функция 223
atan(), статический метод класса Math 621
atan2(), статический метод класса Math 622
attachAudio(), метод класса MovieClip 840
attachMovie(), метод класса MovieClip 394; 396
attachVideo(), метод класса Video 833
autoSize, свойство класса TextField 506

B

background, свойство класса TextField 503
backgroundColor, свойство класса TextField 503
Bandwidth Profiler, окно 1030
beginFill(), метод класса MovieClip 454
beginGradientFill(), метод класса MovieClip 463
blockIndent, свойство класса TextFormat 516
bold, свойство класса TextFormat 514
border, свойство класса TextField 504
borderColor, свойство класса TextField 505
bottomScroll, свойство класса TextField 529
box, тип матрицы трансформации градиента 465
break, предложение 211; 224
broadcastMessage(), метод объекта AsBroadcaster 339
bufferlength, свойство класса NetStream 837
bufferTime, свойство класса NetStream 837
builtInItems, свойство класса ContextMenu 795
bullet, свойство класса TextFormat 515
Button, класс 301
bytesLoaded, свойство класса NetStream 836
bytesTotal, свойство класса NetStream 836

C

call(), метод класса Function 250
callee, свойство объекта arguments 238
caller, свойство объекта arguments 239
capabilities, статическое свойство класса System 805
caption, свойство класса ContextMenuItem 799
case, предложение 210
catalog.xml, документ 1009
catch, ключевое слово 1016
ceil(), статический метод класса Math 617
charAt(), метод класса String 155
charCodeAt(), метод класса String 162
classpath, понятие 858
clear(), метод класса Shared Object 772
clear(), метод класса StyleSheet 595
clear(), метод класса Video 834

clearInterval(), глобальная функция 254
close(), метод класса Local Connection 751
close(), метод класса NetStream 846
Color, класс 677
color, свойство класса TextFormat 514
Component Inspector, панель 960
Components, панель 1010
concat(), метод класса Array 273
concat(), метод класса String 155
condenseWhite, свойство класса TextField 560
connect(), метод класса Local Connection 747
connect(), метод класса NetConnection 835
constructor, свойство 894
ContextMenu, класс 792
ContextMenuItem, класс 798
continue, предложение 225
copy(), метод класса ContexMenuItem 800
copy(), метод класса ContextMenu, 797
cos(), статический метод класса Math 618
create(), статический метод класса Delegate 996
createEmptyMovieClip(), метод класса MovieClip 409
createTextField(), метод класса MovieClip 500
CSS, поддерживаемые свойства 574
currentFps, свойство класса NetStream 839
curveTo(), метод класса MovieClip 480
customItems, свойство объекта ContextMenu 801

D

data, свойство класса SharedObject 766
data, событие клипа 322
Date, класс 629
Debugger, панель 1039
default, предложение 211
Delegate, класс 996
delete, оператор 136
dispatchEvent(), метод класса EventDispatcher 989; 990
do while, предложение 218
dragOut, событие 307
dragOver, событие 307
duplicateMovieClip(), метод класса MovieClip 400
duration, свойство класса Sound 704
dynamic, ключевое слово 887

E

E, константа класса Math 611
else if, предложение 206
else, предложение 202
embedFonts, свойство класса TextFormat 513
enabled, свойство 304; 362
enabled, свойство класса ContextMenuItem 799
endFill(), метод класса MovieClip 456
enterFrame, событие клипа 320
Error, класс 1018
Escape-последовательность 150
eval(), глобальная функция 372
Event, тег метаданных 979
EventDispatcher, класс 988
exactSettings, статическое свойство класса System 763
exp(), статический метод класса Math 613
Export in first frame, опция программного экспорта 395
extends, ключевое слово 889
Extensions Manager, утилита 1010

F

finally, ключевое слово 1016
findText(), метод класса TextSnapshot 604
Flash Video Exporter, плагин 828

floor(), статический метод класса Math 617
flush(), метод класса SharedObject 768
focusEnabled, свойство 374; 541
font, свойство класса TextFormat 512
for, предложение 219
for...in, предложение 222
fromCharCode(), метод класса String 161
fscommand(), глобальная функция 790

G

get, ключевое слово 883
get/set методы 882
getAscii(), статический метод класса Key 658
getBeginIndex(), статический метод класса Selection 544
getBounds(), метод класса MovieClip 431
getBytesLoaded(), метод класса MovieClip 412
getBytesLoaded(), метод класса Sound 708
getBytesTotal(), метод класса MovieClip 411
getBytesTotal(), метод класса Sound 708
getCaretIndex(), статический метод класса Selection 545
getCode(), статический метод класса Key 656
getCount(), метод класса TextSnapshot 603
getDepth(), метод класса MovieClip 406
getEndIndex(), статический метод класса Selection 544
getFocus(), статический метод класса Selection 542
getFontList(), статический метод класса TextField 526
getInstanceAtDepth(), метод класса MovieClip 407
getLocal(), статический метод класса SharedObject 763
getNewTextFormat(), метод класса TextField 523
getNextHighestDepth(), метод класса MovieClip 406
getPan(), метод класса Sound 716
getProgress(), Метод класса MovieClipLoader 730
getRGB(), метод класса Color 679
getSelected(), метод класса TextSnapshot 607
getSelectedText(), метод класса TextSnapshot 607
getSize(), метод класса Shared Object 771
getStyle(), метод класса StyleSheet 593
getStyleNames(), метод класса StyleSheet 594
getText(), метод класса TextSnapshot 603
getTextExtent(), метод класса TextFormat 523
getTextFormat(), метод класса TextField 522
getTextSnapShot(), метод класса MovieClip 602
getTime(), метод класса Date 638
getTimer(), глобальная функция 343; 644
getTimezoneOffset(), метод класса Date 633
getTransform(), метод класса Color 688
getURL(), глобальная функция 389
getVolume(), метод класса Sound 714
globalToLocal(), метод класса MovieClip 430
gotoAndPlay(), глобальная функция 381
gotoAndPlay(), метод класса MovieClip 381
gotoAndStop(), глобальная функция 378
gotoAndStop(), метод класса MovieClip 380

H

handleEvent(), метод объекта-приемника 993
hasOwnProperty(), метод класса Object 868
height, свойство класса Video 834
height, статическое свойство класса Stage 784
hide(), статический метод класса Mouse 649
hideBuiltInItems(), метод класса ContextMenu 796
hitArea, свойство класса MovieClip 317; 362
hitTest(), метод класса MovieClip 440; 443
hitTestTextNearPos(), метод класса TextSnapshot 607
HSB, цветовая модель 668
hscroll, свойство класса TextField 530
htmlText, свойство класса TextField 559

I

IconFile, тег метаданных 980
ID3 теги 710
id3, свойство класса Sound 710
if, предложение 199
import, ключевое слово 865
indent, свойство класса TextFormat 516
indexOf(), метод класса String 156
infinity, числовая константа 143
initialize(), метод объекта AsBroadcaster 338
initialize(), статический метод класса EventDispatcher 989
InspectableList, тег метаданных 978
instanceof, оператор 265
interface, ключевое слово 902
isDown(), статический метод класса Key 660
isFinite(), глобальная функция, 143
isNaN(), глобальная функция, 144
isToggled(), статический метод класса Key, 659
italic, свойство класса TextFormat 514

J

join(), метод класса Array 275

K

Key, класс 655
keyDown, событие клипа 324
keyPress, событие 307; 311
keyUp, событие клипа 324

L

lastIndexOf(), метод класса String 156
leading, свойство класса TextFormat 516
leftMargin, свойство класса TextFormat 516
length, свойство класса Array 267
length, свойство класса String 154
length, свойство класса TextField 510
lineStyle(), метод класса MovieClip 448
lineTo(), метод класса MovieClip 448
Linkage Properties, диалоговое окно 395; 896
List Objects, команда панели Output 1036
List Variables, команда панели Output 1038
Live Preview, режим 1007
LN10, константа класса Math 611
LN2, константа класса Math 611
load(), метод класса StyleSheet 590
load(), метод класса XML 596
load, событие клипа 321
loadClip(), Метод класса MovieClipLoader 728
loadMovie(), метод класса MovieClip 418

loadMovieNum(), глобальная функция 415
loadSound(), метод класса Sound 706
loadVariables(), метод класса MovieClip 425
loadVariablesNum(), глобальная функция 425
Local Storage, вкладка панели настроек проигрывателя Settings 761
LocalConnection, класс 744
localToGlobal(), метод класса MovieClip 428
log(), статический метод класса Math 614
LOG10E, константа класса Math 611
LOG2E, константа класса Math 611

M

Math, класс 611
max(), статический метод класса Math 614
MAX_VALUE, свойство класса Number 142
maxChars, свойство класса TextField 509
maxhscroll, свойство класса TextField 530
maxscroll, свойство класса TextField 529
menu, свойство 304; 374; 501
min(), статический метод класса Math 614
MIN_VALUE, свойство класса Number 142
Mouse, класс 648
mouseDown, событие клипа 323
mouseMove, событие клипа 323
mouseUp, событие клипа 323
mouseWheelEnabled, свойство класса TextField 527
moveTo(), метод класса MovieClip 447
MovieClipLoader, класс 728
multiline, свойство класса TextField 506

N

NaN, свойство класса Number 143
NaN, числовое значение 144
NEGATIVE_INFINITY, свойство класса Number 143
NetConnection, класс 834
NetStream, класс 835
nextFrame(), метод класса MovieClip 382
nextScene(), глобальная функция 383

O

on, обработчик событий кнопок и клипов 309; 316; 991
onChanged, обработчик события 550
onClipEvent, обработчик событий клипов 319
onData, обработчик события 327; 426
onDragOut, обработчик события 327
onDragOver, обработчик события 327
onEnterFrame, обработчик события 327
onKeyDown, обработчик события 327; 655
onKeyUp, обработчик события 3 27; 655
onKillFocus, обработчик события 327; 551
onLoad, обработчик события 327; 590; 597; 709
onLoadComplete, обработчик события 734
onLoadError, обработчик события 736
onLoadInit, обработчик события 735
onLoadProgress, обработчик события 730; 734
onLoadStart, обработчик события 731
onMouseDown, обработчик события 327; 649
onMouseMove, обработчик события 327; 649
onMouseUp, обработчик события 327; 649

onMouseWheel, обработчик события 649
onPress, обработчик события 327
onRelease, обработчик события 327
onReleaseOutside, обработчик события 327
onResize, обработчик события 785
onRollOut, обработчик события 327
onRollOver, обработчик события 327
onScroller, обработчик события 531
onSelect, обработчик события 797; 800
onSetFocus, обработчик события 327; 543; 551
onSoundComplete, обработчик события 704
onStatus, обработчик события 748; 769; 842
onUnload, обработчик события 327
onUpdate(), метод компонента 1008
Output, панель 1035

P

parseCSS(), метод класса StyleSheet 587
parseFloat(), глобальная функция 171
parseInt(), глобальная функция 170
password, свойство класса TextField 509
pause(), метод класса NetStream 838
PI, константа класса Math 611
play(), метод класса MovieClip 377
play(), метод класса NetStream 835
pop(), метод класса Array 289
position, свойство класса Sound 704
POSITIVE_INFINITY, свойство класса Number 143
pow(), статический метод класса Math 612
press, событие 307
prevFrame(), метод класса MovieClip 383
prevScene(), глобальная функция 383
PrintJob, класс 810
private, ключевое слово 877
Project, панель 104
prototype, свойство 874
public, ключевое слово 876
push, метод класса Array 270

R

random(), статический метод класса Math 624
registerClass(), статический метод класса Object 898
release, событие 307
releaseOutside, событие 307
removeEventListener(), метод класса EventDispatcher 989
removeListener(), метод 337
removeMovieClip(), метод класса MovieClip 409
removeTextField(), метод класса TextField 557
replaceSel(), метод класса TextField 555
replaceText(), метод класса TextField 557
restrict, свойство класса TextField 508
return, предложение 240
reverse(), метод класса Array 276
RGB, цветовая модель 665
rightMargin, свойство класса TextFormat 516
rollOut, событие 307
rollOver, событие 307
round(), статический метод класса Math 616

S

scaleMode, статическое свойство класса Stage 780
scroll, свойство класса TextField 528
seek(), метод класса NetStream 839
selectable, свойство класса TextField 507
Selection, класс 540

send(), метод класса Local Connection 748
send(), метод класса PrintJob 816
separatorBefore, свойство класса ContextMenuItem 799
set, ключевое слово 883
set, предложение 117
setBufferTime(), метод класса NetStream 837
setClipboard(), метод класса System 610
setFocus(), статический метод класса Selection 540
setInterval(), глобальная функция 251
setMask(), метод класса MovieClip 483
setNewTextFormat(), метод класса TextField 512
setPan(), метод класса Sound 716
setRGB(), метод класса Color 678
setSelectColor(), метод класса TextSnapshot 606
setSelected(), метод класса TextSnapshot 606
setSelection(), статический метод класса Selection 547
setStyle(), метод класса StyleSheet 582
setTextFormat(), метод класса TextField 511
setTime(), метод класса Date 639
setTransform(), метод класса Color 687
setTransform(), метод класса Sound 718
setVolume(), метод класса Sound 714
SharedObject, класс 759
shift(), метод класса Array 290
show(), статический метод класса Mouse 649
showMenu, статическое свойство класса Stage 789
showSettings(), статический метод класса System 761
sin(), статический метод класса Math 618
size, свойство класса TextFormat 513
slice(), метод класса Array 288
slice(), метод класса String 159
smoothing, свойство класса Video 834
sort(), метод класса Array 277
sortOn(), метод класса Array 284
Sound, класс 698
splice(), метод класса Array 272
split(), метод класса String 160
sqrt(), статический метод класса Math 613
SQRT1_2, константа класса Math 611
SQRT2, константа класса Math 611
Stage, класс 779
start(), метод класса PrintJob 812
start(), метод класса Sound 701
startDrag(), глобальная функция 437
startDrag(), метод класса MovieClip 437
static, ключевое слово 880
stop(), метод класса MovieClip 376
stop(), метод класса Sound 702
stopAllSounds(), глобальная функция 703
stopDrag(), метод класса MovieClip 438
StyleSheet, класс 573
substr(), метод класса String 157
substring(), метод класса String 158
super, оператор 891
swapDepths(), метод класса MovieClip 408
switch, предложение 210
System.capabilities, встроенные свойства 805

T

tabChildren, свойство 374; 548
tabEnabled, свойство 304; 374; 548
tabIndex, свойство 304; 374; 549
tabStops, свойство класса TextFormat 517
tan(), статический метод класса Math 618
text, свойсто класса TextField 502
textColor, свойство класса TextField 503
TextField, класс 500

TextFormat, класс 510
textHeight, свойство класса TextField 509
TextSnapshot, класс 601
textWidth, свойство класса TextField 509
this, идентификатор 134
throw, предложение 1017
time, свойство класса NetStream 838
toLowerCase(), метод класса String 161
toString(), метод класса Array 275
toString(), метод класса Boolean 167
toString(), метод класса Error 1019
toString(), метод класса Number 147
toUpperCase(), метод класса String 161
trace(), глобальная функция 96
Track as Button, режим кнопки 306
Track as Menu Item, режим кнопки 306
trackAsMenu, свойство 304; 362
try, ключевое слово 1016
try...catch...finally, конструкция 1017
type, свойство класса TextField 502
typeof, унарный оператор 116

U

underline, свойство класса TextFormat 514
unload, событие клипа 322
unloadClip(), Метод класса MovieClipLoader 729
unloadMovie(), метод класса MovieClip 427
unshift(), метод класса Array 271
updateAfterEvent(), глобальная функция 253; 323
useHandCursor, свойство 304; 361
UTC(), статический метод класса Date 639

V

valueOf(), метод класса Boolean 167
valueOf(), метод класса Number 148
variable, свойство класса TextField 502
Video, класс 833
visible, свойство класса ContextMenuItem 799

W

watch(), метод класса Object 869
while, предложение 216
width, свойство класса Video 834
width, статическое свойство класса Stage 784
with, предложение 375
wordWrap, свойство класса TextField 505

A

Абзац, параметры 516
Анимация:
- автоматическая (Tweened animation) 45
- вращения 623; 657
- градиента 467
- движения (Motion tweening) 49
- движения по маршруту 345; 620
- запуск/остановка 204; 376; 803
- покадровая (Frame by frame) 45
- программная (Program animation) 345, 905
- программный реверс 386
- прозрачности 352; 420
- прокрутки текста 532
- с дублированием 404, 620
- с применением копирования и вращения 404
- случайных процессов 625; 896; 964
- трехмерных точек 949; 952
- ускорение/замедление 49; 380
- формы (Shape tweening) 47
- цвета 679

Б

Байткод 861
Безопасная Web-палитра 681
Безопасность:
  взаимодействия между документами 754
  загрузки внешних данных 44
  загрузки внешних документов 742
  локального хранения информации,762
Библиотека:
  документа 40
  общая 41; 497

В

Взаимодействие:
  между фильмами 744; 749; 752
  между экземплярами класса 881
Видео:
  включение/отключение звука 841
  динамическая загрузка 825; 835
  контроль загрузки 836
  масштабирование относительно экрана 843
  поиск 839
  проигрыватель 846
  размер буфера 835; 837
  экспорт в формат FLV, 828
Видеообъект 833
Виртуальный код клавиши 656
Возбуждение исключительной ситуации 1019
Время:
  всемирное 630
  всемирное скоординированное (UTC) 630; 636
  машинное 630
  местное 634; 636
Всплывающие подсказки 85; 517
Встраивание шрифта 493; 495; 512
Выравнивание:
  текста 514
  фильма в окне проигрывателя, 782

Г

Генератор случайных чисел 624
Гиперссылка 562
Градиент:
  единичный 471
  комбинированные преобразования 474
  масштабирование 471
  матрица трансформации 465
  поворот 473
  понятие 463
  скос 474
  смещение 472

Д

Дата 631
Директива:
  #endinitclip 899
  #include 103
  #initclip 899

З

Загрузка:
  CSS-документов 590
  XML-документов 595
  в клип 418
  видео 825
  внешних SWF-документов 415; 566, 728
  звука 706
  имитация 1030
  нескольких документов 737
  ошибка 736
  переменных 425
  растровых изображений 415; 566; 728
  текста, 425; 595
Звук:
  баланс, панорамирование 716
  динамическая загрузка 706
  динамическое присоединение 700

(окончание рубрики см. на с. 1104)

Звук *(окончание)*:
зацикливание 704
идентификатор связи 700
импорт 50
распределение сигнала по каналам 718
синхронизация 51
управление воспроизведением 701
уровень 714
форматы 50

И

Идентификатор 94
Имя экземпляра 305; 344; 489
Интерфейс 901
Исключительная ситуация 1014

К

Календарь 639
Качество отображения 369
Кегль 513
Клавиатурные константы класса Key 657
Клавиатурные константы события keyPress 312
Клавиша:
контроль нажатия 660
определение виртуального кода 656
определение кода символа 658
Класс:
динамический 886
определение типа данных 862
понятие 861
экземпляр 863
Клип:
динамическое создание 391; 409; 651
дублирование 400; 802
идентификатор связи 395
клип-кнопка 316
контроль наложения 440
маскирование 483
область видимости сценария 317; 325
область реагирования 317
оценка размеров 348
присоединение 394
связывание с классом 896
события 319
удаление 409
Кнопка:
встроенные свойства 304
идентификатор связи 395
клип-кнопка 316
область видимости сценария 313
обработчик событий 308
пустая 303
режимы 306; 307
события 307
создание 303
Кодировка 149; 658
Комментарии:
многострочные 93
однострочные 93
Компилированный клип 960
Компонент:
генерирование событий 988
динамическое присоединение 981
масштабирование 999; 1001
определение параметров 973; 980
пиктограмма 979
понятие 960
создание MXP-пакета 1012
создание SWC-архива 1010
создание клипа 967
трансляция событий 990
Конкатенация 151; 681
Конструктор 871
Контекстное меню проигрывателя:
копирование 797
обработка вызова 797; 800
отключение стандартных команд 789; 796
создание пользовательского элемента 798; 801
Координаты:
преобразование глобальных в локальные 430

преобразование локальных
в глобальные 428
Кривые Безье квадратичные 479
Кривые Гвидо Гранди розы 618
Курсор:
слежение 624; 652
управление отображением 649

Л

Ленивые вычисления 871
Литерал:
null 168
undefined 168
булев 164
массива 264
понятие 138
строковый 148
функции 233
числовой 141
Логарифм 613

М

Маска 483
Массив:
ассоциативный 290
вложенный 292
добавление элементов 269
литерал 264
многомерный 292
понятие 262
сортировка 276; 912
удаление элементов 289
Масштабирование объекта до размера печатной области 818
Масштабирование окна проигрывателя 779
Масштабирование фильма 780
Математические константы 611
Матрица:
моделирование с помощью двумерного массива 293; 294
общего преобразования градиента 470
понятие 919
проецирования 928
транспонирование 296
трансформации градиента 465
трехмерных преобразований 925; 927
Машинно-независимый шрифт:
маскирование 484
понятие 496
Метод 874
Метод обработчика события 326
Метрики шрифта 523

Н

Навигация:
между Web-страницами 391
на монтажной линейке 385
с использованием динамической загрузки внешних документов 417
Наследование 888; 894

О

Обработчик исключительной ситуации 1014
Объектно-ориентированное программирование, понятие 101
Объявление тегов 580
Ограничивающая рамка, 431
Однородные координаты 926
Оператор:
delete 136
instanceof 265
typeof 116
арифметический 144
ассоциативность 180
вызова функции, () 245
декрементирования, −− 182
доступа к элементу массива, [] 130
инкрементирования, ++ 182
логический И, && 166
логический ИЛИ, ||165

(окончание рубрики см. на с. 1106)

Оператор *(окончание)*:
логический НЕ, ! 166
неравенства 146
отношений 144
перегруженный 176
побитовый 184; 193
побитовый составной 192
побитовое И, &189
побитовое ИЛИ, | 189
побитовое исключающее ИЛИ, ^ 191
побитовое отрицание, ~ 192
побитового сдвига влево, << 196
побитового сдвига вправо, >> 194
побитового сдвига вправо без знака, >>> 196
приоритет 177
равенства 146
составные 182
условный 205, 213
Отладка 1035
в среде тестирования 1040
удаленная 1040

П

Пакет 864
Паттерн, регулярный узор 398
Перекрытие методов и свойств 890
Переменная:
$version 222
абсолютная адресация 131
время жизни 135
глобальная 126
динамическое создание 117
загрузка 425
локальная 123
модификация значения 180
неявное объявление 117
область видимости 121
относительная адресация 133
понятие 113
типизация 114; 120
удаление 136
явное объявление 114
Переопределение тегов 579
Перетаскивание объектов 356; 437
Перспективное преобразование 930
Пикселы, единицы измерения 813
Поддержка:
CSS 572
HTML, 559; 560
Подкласс 888
Покадровый график 1034
Ползунковый регулятор 693; 722; 849
Полноэкранный режим 790
Пользовательское событие 100
Потоковый график 1032
Пошаговое выполнение кода, 1047
Предзагрузчик:
внешний 732; 737
внутренний 412
звука 708
покадровый 359
Проектор 790
Проецирование 928
Прокрутка текста 527; 534
Путь к файлам классов 858

Р

Режим предварительного просмотра компонента 1007
Рисование:
кривых 479
ломаных 450
многоугольников и звезд 457
прямых 447

С

Свойства:
для чтения ID3-тегов 710
документа 23
класса 865
класса PrintJob 812
наследуемые 866
объекта System.capabilities 805
собственные 867; 872
Серверная строка 805

Символ 35
Синтаксис:
подсветка 81
предложений 92
проверка 83
Системное событие 100
Слайд-шоу 420
Событийный объект 991
Список:
выпадающий 363
маркированный 515; 564
Спулер печати 811
Стек:
графических объектов 392
документов 394; 416
контроль заполнения уровней 406
слоев документа 392
Стилевой класс 576
Стиль:
добавление в таблицу 581; 587
загрузка внешней таблицы 590
описание 593
определение с применением CSS 576
очистка таблицы 595
создание 573; 582
Суперкласс 888
Сценарий:
внешний 103
кадра 98
клипа 99
кнопки 99
носитель 89
последовательность выполнения 98
прикрепление 90

T

Таблица стилей 573
Табуляция:
передача фокуса с помощью клавиши <Tab> 548
установка позиции 517
Таймер 645
Твипы единицы измерения 813
Теги метаданных 963; 973
Текстовый блок:
внедрение внешних файлов 566
вывод и считывание текста 502
выключка 514
ограничение количества вводимых символов 509
ограничение набора вводимых символов 508
переменная 489; 493; 502
перенос строк 505
подгонка размеров 506
прокрутка 528; 529
размеры текста 509
рамка 504
режим пароля 509
свойства 501
события 550
создание 489; 500
тип 502
удаление 557
фон 503
форматирование в рабочей среде 490
форматирование динамическое 510
форматирование с помощью CSS 572
форматирование с помощью HTML 558
цвет текста 503
Тестирование:
загрузки 411
фильма 1028
Типы данных:
Boolean булев 164
null 168
Number числовой 140
String строковый 148
undefined 168
преобразование 169
ссылочные 139
элементарные 138
Точка прерывания 1046

Точки единицы измерения 813
Трансляция событий 333
Трехмерные координаты 924
Тригонометрические функции 617

У

Удаление документа 427
Умножение матриц 920; 921
Уровень:
  границы значений 396
  иерархии графических объектов 17
  стека графических объектов 392
  стека документов 394; 416

Ф

Флажок 774
Фокус:
  индикация рамкой 548
  контроль передачи 551
  обнаружение 542
  передача 540; 543
  понятие 540
  порядок автоматической передачи 548
  свойства управления порядком переноса 374
  управление порядком передачи 549
Форматы файлов:
  AS 103
  ASI 1009
  ASO 860
  EXE 64; 790
  FLA 6
  FLV 827
  HTML 59
  MP3 706
  MXI 1012
  MXP 1010
  SOL 761
  SWC 1009
  SWD 1040
  SWF 7; 56
Функция:
  время жизни 259
  вызов 245
  глобальная 258
  литерал 233
  область видимости 255
  параметры 234
  периодический вызов 250
  понятие 231
  рекурсивная 259
  сравнения 281

Ц

Цвет:
  аддитивное смешение 666
  глубина 664
  дополнительный 688
  первичный 665
  перевод из модели HSB в модель RGB 674
  перевод из модели RGB в модель HSB 672
  синтез 691
  сортировка по тону 682
  шестнадцатеричное представление 194; 667
Цветовая модель:
  HSB 668
  RGB 665
Цветовая трансформация:
  аддитивная 686
  управление 689
Цикл:
  do while 218
  for 219
  for...in 222
  while 216
  бесконечный 215
  вложенный 225; 293
  выход 224
  монтажной линейки 226
  понятие 215

Ч

Часы 635
Число:
возведение в степень 612
восьмеричное 141
двоичное 185
извлечение корня 612
логарифм 613
модуль 615
округление 616
проверка на дробность 203; 617
шестнадцатеричное 141
Эйлера 613

Ш

Шлейф мыши 651
Шрифт, гарнитура 512
Шрифтовой символ 498

Э

Экземпляр 35; 38
Эхо 714