Решения для разработчиков приложений
на платформе Adobe® Flash® и Adobe Flex™

ActionScript 3.0
Сборник рецептов

Символ®
O'REILLY®

Джои Лотт,
Деррон Шалл,
Кейт Питерс

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-091-X, название «ActionScript 3.0. Сборник рецептов» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

ActionScript 3.0 Cookbook™

Joey Lott, Darron Schall, and Keith Piters

O'REILLY®

ActionScript 3.0
Сборник рецептов

Джои Лотт, Деррон Шалл
и Кейт Питерс

Санкт-Петербург — Москва
2008

Джои Лотт, Деррон Шалл и Кейт Питерс

ActionScript 3.0. Сборник рецептов

Перевод Н. Шатохиной

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*М. Антипин*
Редактор	*В. Овчинников*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Лотт Дж., Шалл Д., Питерс К.
ActionScript 3.0. Сборник рецептов. – Пер. с англ. – СПб: Символ-Плюс, 2007. – 608 с., ил.

ISBN-10: 5-93286-091-X
ISBN-13: 978-5-93286-091-5

Если вам надо быстро найти решение при работе с программным обеспечением Adobe Flash ® или Adobe Flex™, возьмите «ActionScript 3.0. Сборник рецептов». Более 300 рецептов этого практического инструментария дадут ответы на все вопросы, касающиеся ActionScript 3.0. Среди них: определение версии Flash Player или операционной системы пользователя, форматирование дат и представление сумм в разных валютах, обработка пользовательского ввода и работа с текстовыми строками, создание графических объектов во время выполнения, работа с аудио- и видеоданными, удаленный вызов процедур с использованием технологии Flash Remoting, а также загрузка, отправка и поиск XML-данных.

Книга адресована разработчикам во Flash и Flex 2 и содержит готовые решения задач, с которыми они ежедневно сталкиваются. Все рецепты снабжены кратким комментарием, поясняющим механизм работы рекомендуемого программного кода, что облегчает его адаптирование под конкретную задачу.

ISBN-10: 5-93286-091-X
ISBN-13: 978-5-93286-091-5
ISBN-10: 0-596-52695-4 (англ)
ISBN-13: 978-0-596-52695-5 (англ)

© Издательство Символ-Плюс, 2007

Authorized translation of the English edition © 2007 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

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

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

Подписано в печать 30.10.2007. Формат 70x100 1/16 . Печать офсетная.
Объем 38 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 14

1. Основы ActionScript 25

1.0. Введение 25
1.1. Создание проекта ActionScript 26
1.2. Настройка свойств приложения 27
1.3. Где размещать код ActionScript 28
1.4. Как трассировать сообщение 33
1.5. Обработка событий 35
1.6. Реакция на события мыши и клавиатуры 36
1.7. Применение математических операторов 38
1.8. Проверка равенства или сравнение значений 41
1.9. Выполнение действий по условию 45
1.10. Проверка сложных условий 50
1.11. Многократное повторение операции 51
1.12. Повторение задачи в течение некоторого промежутка времени 56
1.13. Создание многократно используемого кода 58
1.14. Обобщение метода 60
1.15. Выход из метода 62
1.16. Получение результата метода 63
1.17. Обработка ошибок 64

2. Специальные классы 69

2.0. Введение 69
2.1. Создание специального класса 69
2.2. Где сохранять класс 74
2.3. Создание свойств, которые ведут себя как методы 75
2.4. Создание статических методов и свойств 77
2.5. Создание подклассов 78
2.6. Реализация версий методов надкласса в подклассе 81
2.7. Создание констант 83
2.8. Передача событий 84

3. Среда времени выполнения 85

3.0. Введение 85
3.1. Определение версии проигрывателя 85
3.2. Определение операционной системы 87
3.3. Проверка типа проигрывателя 88
3.4. Проверка языка системы 89
3.5. Определение настроек экрана 91
3.6. Масштабирование ролика 92
3.7. Изменение выравнивания 93
3.8. Сокрытие пунктов меню Flash Player 95
3.9. Определение аудиовозможностей устройства 96
3.10. Определение видеовозможностей устройства 97
3.11. Как подсказать пользователю изменить настройки проигрывателя 98
3.12. Безопасность системы 99

4. Числа и математические операции 102

4.0. Введение 102
4.1. Представление чисел в разных системах счисления 102
4.2. Преобразование из одной системы счисления в другую 104
4.3. Округление чисел 106
4.4. Как добавить ведущие или завершающие нули или пробелы 108
4.5. Форматирование чисел для отображения без применения маски 112
4.6. Форматирование денежных сумм 113
4.7. Генерирование случайных чисел 115
4.8. Моделирование подбрасывания монеты 116
4.9. Моделирование игры в кости 120
4.10. Моделирование карточной игры 122
4.11. Генерирование уникального числа 124
4.12. Преобразование единиц измерений углов 124
4.13. Вычисление расстояния между двумя точками 126
4.14. Определение координат точки окружности 127
4.15. Преобразование единиц измерения 130

5. Массивы 132

5.0. Введение 132
5.1. Добавление элементов в начало или конец массива 133
5.2. Перебор всех членов массива 135
5.3. Поиск элементов массива, соответствующих некоторому критерию 136
5.4. Удаление элементов 141

5.5. Как вставить элементы в середину массива 143
5.6. Преобразование строки в массив 144
5.7. Преобразование массива в строку 145
5.8. Создание отдельной копии массива 146
5.9. Хранение сложных или многомерных данных 150
5.10. Сортировка или обращение массива 153
5.11. Реализация специальной сортировки 158
5.12. Расположение элементов массива в случайном порядке 160
5.13. Получение наименьшего и наибольшего элементов 161
5.14. Сравнение массивов 162
5.15. Создание ассоциативного массива 164
5.16. Чтение элементов ассоциативного массива 166

6. Список отображения 168

6.0. Введение 168
6.1. Добавление элемента в список отображения 172
6.2. Удаление элемента из списка отображения 178
6.3. Изменение визуального порядка размещения объектов 181
6.4. Создание специальных визуальных классов 186
6.5. Создание простых кнопок 188
6.6. Загрузка внешних изображений во время выполнения 194
6.7. Загрузка и взаимодействие с внешними роликами 198
6.8. Организация взаимодействия с мышью 202
6.9. Перемещение объектов с помощью мыши 206

7. Программное создание изображений и масок 215

7.0. Введение 215
7.1. Как задать стиль линии 216
7.2. Как задавать градиентные стили линии 217
7.3. Как нарисовать линию 218
7.4. Как нарисовать кривую 220
7.5. Как нарисовать дугу 221
7.6. Как нарисовать прямоугольник 221
7.7. Как нарисовать круг 223
7.8. Как нарисовать эллипс 224
7.9. Как нарисовать треугольник 224
7.10. Как нарисовать правильный многоугольник 226
7.11. Как нарисовать звезду 227
7.12. Закрашивание формы прозрачным или непрозрачным цветом 227
7.13. Градиентное закрашивание формы 228
7.14. Заливка формы растровым изображением 231
7.15. Создание масок 232

8. Растровые изображения 234

8.0. Введение 234
8.1. Создание объекта BitmapData 235
8.2. Как добавить растровое изображение в список отображения 236
8.3. Наполнение растрового изображения графическим содержимым 237
8.4. Как загрузить внешнее изображение в экземпляр класса Bitmap 238
8.5. Работа с пикселами 240
8.6. Создание прямоугольной заливки 242
8.7. Создание заливки области 243
8.8. Копирование пикселов 244
8.9. Копирование каналов 246
8.10. Создание шума 247
8.11. Создание шума Перлина 249
8.12. Использование порогового значения 253
8.13. Применение фильтра к растровому изображению 256
8.14. Растворение одного растрового изображения в другом 260
8.15. Прокрутка растрового изображения 263

9. Текст 264

9.0. Введение 264
9.1. Создание контура вокруг текстового поля 265
9.2. Создание фона в текстовом поле 265
9.3. Создание текстового поля ввода 266
9.4. Создание поля для ввода пароля 267
9.5. Фильтрация текстового ввода 268
9.6. Задание максимальной длины поля 269
9.7. Вывод текста на экран 270
9.8. Вывод на экран текста, форматированного тегами HTML 271
9.9. Сжатие пробелов 272
9.10. Изменение размера текстового поля с целью вмещения содержимого 272
9.11. Реализация прокрутки текста в программе 274
9.12. Ответ на события прокрутки 277
9.13. Форматирование текста 278
9.14. Форматирование текста, вводимого пользователем 283
9.15. Форматирование части существующего текста 284
9.16. Задание шрифта текстового поля 285
9.17. Встраивание шрифтов 287
9.18. Создание текста, который можно поворачивать 288
9.19. Вывод на экран символов Unicode 289
9.20. Перемещение фокуса на текстовое поле 290

9.21. Как выделить текст средствами ActionScript 291
9.22. Как задать положение курсора ввода в текстовом поле 292
9.23. Реакция на установку или снятие фокуса с текстового поля 292
9.24. Реакция на ввод текста пользователем 294
9.25. Как добавить в текст гиперссылку 295
9.26. Вызов ActionScript из гиперссылок 297
9.27. Расширенное форматирование текста 297
9.28. Применение особого сглаживания текста 300
9.29. Замена текста 302
9.30. Получение списка системных шрифтов 302

10. Фильтры и трансформации 303

10.0. Введение 303
10.1. Изменение цвета 303
10.2. Применение оттенков цветов 304
10.3. Восстановление исходного цвета 305
10.4. Сдвиг 306
10.5. Применение базовых фильтров 307
10.6. Применение дополнительных эффектов (выдавливание и т. д.) 309
10.7. Выдавливание 311
10.8. Выделение краев 312
10.9. Повышение резкости 313
10.10. Создание цифрового негатива 314
10.11. Применение эффекта полутонов 314
10.12. Изменение насыщенности 315
10.13. Изменение яркости 316
10.14. Изменение контрастности 317

11. Программная анимация 318

11.0. Введение 318
11.1. Перемещение объекта 319
11.2. Перемещение объекта в заданном направлении 321
11.3. Замедление 323
11.4. Ускорение 325
11.5. Пружины 327
11.6. Тригонометрические выражения 329
11.7. Применение методов анимации к другим свойствам 332

12. Строки 336

12.0. Введение 336
12.1. Объединение строк 337
12.2. Кавычки и апострофы в строках 340

12.3. Как вставить специальные пробельные символы 341
12.4. Поиск подстроки 343
12.5. Извлечение подстроки 348
12.6. Как разобрать строку по словам 351
12.7. Удаление и замена символов и слов 354
12.8. Как извлекать символы по одному 357
12.9. Изменение регистра 359
12.10. Как удалить лишние пробелы 360
12.11. Изменение порядка слов или символов в строке на обратный 362
12.12. Преобразование строк в коды Unicode или ASCII и наоборот 363

13. Регулярные выражения 367

13.0. Введение 367
13.1. Понимание шаблонов регулярных выражений 368
13.2. Тестирование регулярных выражений 374
13.3. Поиск соответствий шаблону 378
13.4. Удаление и замена символов и слов с помощью шаблонов 380
13.5. Создание нежадного шаблона 383
13.6. Проверка данных, вводимых пользователем 386

14. Даты и время 391

14.0. Введение 391
14.1. Как получить текущие дату и время 391
14.2. Как получить значения даты 394
14.3. Как получить название дня или месяца 395
14.4. Форматирование даты и времени 396
14.5. Представление значений секунд или миллисекунд в виде минут и секунд 398
14.6. Преобразование из формата DMYHMSM в миллисекунды эпохи и наоборот 399
14.7. Работа с таймерами 400
14.8. Вычисление прошедшего времени или интервалов между датами 401
14.9. Получение даты из строки 407

15. Программирование звука 408

15.0. Введение 408
15.1. Создание объекта Sound и загрузка звука 408
15.2. Начало и прекращение воспроизведения звука 410
15.3. Как задать аудиобуфер 411
15.4. Смещение начала воспроизведения 412
15.5. Многократное (циклическое) воспроизведение звука 413

15.6. Как получить размер звукового файла 414
15.7. Чтение тега ID3 звукового файла 417
15.8. Как узнать, когда закончилось воспроизведение звукового файла 418
15.9. Отслеживание процесса воспроизведения звука 420
15.10. Приостановка и возобновление воспроизведения звука 423
15.11. Чтение уровня звука 425
15.12. Прекращение воспроизведения всех звуков 426
15.13. Чтение спектрального состава звука 427
15.14. Изменение громкости звука или баланса стереодорожек 429
15.15. Создание приложения работы со звуком 430

16. Работа с видеоданными 437

16.0. Введение 437
16.1. Загрузка и воспроизведение видеоданных 437
16.2. Управление звуком видеофайла 439
16.3. Как узнать время воспроизведения 440
16.4. Как получить длительность видеозаписи 440
16.5. Управление временем воспроизведения 442
16.6. Масштабирование видеоизображения 443
16.7. Контроль и управление буферизацией и загрузкой 444
16.8. Обнаружение сигнальных точек 446
16.9. Применение фильтров к видеоизображению 447
16.10. Приостановка и возобновление воспроизведения видеозаписи 447
16.11. Остановка воспроизведения 448
16.12. Перемотка видеозаписи 449
16.13. Очистка области видеоизображения 451
16.14. Определение пропускной способности соединения пользователя 451

17. Хранение данных 455

17.0. Введение 455
17.1. Создание и открытие локального совместно используемого объекта 456
17.2. Запись данных в LSO 457
17.3. Сохранение локального совместно используемого объекта 458
17.4. Чтение данных из совместно используемого объекта 461
17.5. Удаление данных из совместно используемого объекта 462
17.6. Сериализация специальных классов 463
17.7. Совместное использование данных Flash-приложениями 465
17.8. Управление размерами локально используемых объектов 468

18. Обмен информацией между роликами 470

18.0. Введение 470
18.1. Создание локальных соединений 471
18.2. Отправка данных 476
18.3. Подтверждение получения при передаче сообщений по локальным соединениям 479
18.4. Локальный обмен данными с роликами других доменов 481

19. Отправка и загрузка данных 483

19.0. Введение 483
19.1. Загрузка переменных из текстового файла 485
19.2. Загрузка переменных из сценария, выполняющегося на стороне сервера 488
19.3. Загрузка блока текста (включая HTML и XML) 491
19.4. Контроль за процессом загрузки 493
19.5. Доступ к данным в процессе загрузки 494
19.6. Отправка данных в сценарий, выполняющийся на стороне сервера 496
19.7. Отправка переменных и обработка возвращенного результата 498

20. XML 501

20.0. Введение 501
20.1. Изучение структуры XML (чтение и запись XML) 503
20.2. Создание объекта XML 506
20.3. Добавление элементов в объект XML 508
20.4. Добавление текстовых узлов в объект XML 511
20.5. Добавление атрибутов в элемент XML 512
20.6. Чтение элементов дерева XML 514
20.7. Поиск элементов по имени 515
20.8. Чтение текстовых узлов и их значений 517
20.9. Чтение атрибутов элемента 520
20.10. Удаление элементов, текстовых узлов и атрибутов 522
20.11. Загрузка XML 524
20.12. Загрузка XML из других доменов 525
20.13. Отправка XML 526
20.14. Поиск в XML-документах 532
20.15. Использование HTML и специальных символов в XML 535

21. Веб-сервисы и удаленное взаимодействие во Flash 537

21.0. Введение 537
21.1. Вызов методов веб-сервисов 538
21.2. Обработка ответов веб-сервисов 539
21.3. Обработка ошибок веб-сервисов 540

21.4. Flash Remoting: вызов методов 541
21.5. Flash Remoting: обработка ответов 542

22. Создание интегрированных приложений 544

22.0. Введение 544
22.1. Вызов функций JavaScript 544
22.2. Вызов функций ActionScript 545
22.3. Передача параметров из HTML 547

23. Работа с файлами 549

23.0. Введение 549
23.1. Загрузка файлов 549
23.2. Определение момента выбора файла пользователем 552
23.3. Отслеживание процесса загрузки 553
23.4. Просмотр файлов 554
23.5. Фильтрация файлов, которые будут отображаться в окне просмотра файлов 555
23.6. Определение момента выбора пользователем файла для передачи 556
23.7. Передача файлов 557
23.8. Отслеживание процесса передачи файла 558

24. Работа с сокетами 559

24.0. Введение 559
24.1. Соединение с сокет-сервером 560
24.2. Отправка данных 563
24.3. Получение данных 565
24.4. Подтверждение установления связи с сервером 570
24.5. Разрыв соединения с сервером 573
24.6. Обработка ошибок сокетного соединения 574

A. Таблица кодов Unicode для символов Latin 1 576

Алфавитный указатель 581

Предисловие

Книга «ActionScript 3.0. Сборник рецептов» адресована разработчикам на ActionScript любого уровня. Она написана для обычных людей, которым нужны практические решения часто встречающихся задач. Положите эту книгу рядом с компьютером, и она поможет ответить на вопросы, которые возникают в ходе разработки программных продуктов. Она полна реальных примеров, замечательных и доступных решений, предлагает анализ ситуаций, с которыми, несомненно, сталкиваются разработчики во Flash и на ActionScript.

Книга написана в классическом для серии «O'Reilly Cookbook» (Сборники рецептов O'Reilly) формате. Каждый рецепт представлен разделами «Задача», «Решение» и «Обсуждение». Читатель может быстро найти рецепт, наиболее соответствующий его ситуации, и получить ответ. Чтобы понять предлагаемый код, ему не придется перечитывать всю книгу. В разделе «Обсуждение» каждого рецепта приводится углубленный анализ решения, обсуждаются его возможные варианты и последствия их применения. Таким образом, вы убиваете сразу двух зайцев – получаете быстрый и легкий доступ к необходимым ответам и возможность глубже взглянуть на природу как проблемы, так и решения. «ActionScript 3.0. Сборник рецептов» помогает понять основные принципы на реальных примерах.

Чего нет в этой книге

В данной книге представлен большой объем информации по широкому диапазону тем. Она охватывает весь спектр вопросов по применению ActionScript для создания программных продуктов, работающих на клиентской стороне. Каждый рецепт приводится в контексте практического примера. Необходимая теория, конечно, дается, но это не учебник. ActionScript подробно рассмотрен в массе хороших книг, справочников и документов. «ActionScript 3.0. Сборник рецептов» преследует иные цели. Особый формат этой книги призван помочь читателям в решении конкретных задач.

Совместимость

Книга называется «ActionScript 3.0. Сборник рецептов». То есть все примеры кода в ней базируются на ActionScript 3.0 и совместимы только с продуктами, поддерживающими ActionScript 3.0. Flex 2.0 и Flash 9 позволяют писать на ActionScript 3.0. Flash Player 9 поддерживает ActionScript 3.0. Если продукт, с которым вы работаете, не поддерживает ActionScript 3.0, то код, приведенный в данной книге, скорее всего функционировать не будет.

Принятые наименования платформы Flash

ActionScript 3.0 – важная часть Flash Platform, включающей множество взаимосвязанных технологий, ориентированных на Flash Player. Сегодня технологий Flash Platform так много, что отследить их все очень сложно. Еще более усугубляет ситуацию неформальное, неаккуратное и даже неправильное употребление терминов многими разработчиками. Мы не претендуем на роль блюстителей чистоты терминологии, но стремимся к максимальной точности и ясности при рассмотрении этих технологий в данной книге и поэтому будем оперировать терминами весьма конкретно. В табл. 1 собраны употребляемые нами названия и их значения.

Таблица 1. Принятые в данной книге наименования Flash Platform

Имя	Значение
Flex framework	Библиотека классов ActionScript, поставляемая как часть Flex SDK и Flex Builder.
Flex Builder	Интегрированная среда разработки Adobe для создания Flex-приложений.
Flex SDK	Компилятор и инфраструктура Flex, используемая для создания Flex-приложений без Flex Builder.
Flex	Технология, применяемая для создания SWF-файлов из документов MXML и файлов ActionScript. Если не указан номер версии, речь идет о Flex 2.0.
Flash Player	Платформа для работы SWF-файлов, созданных во Flash или Flex. Если не указан номер версии, речь идет о Flash Player 9.
Flash	Среда разработки, используемая для создания SWF-файлов. Если не указан номер версии, речь идет о Flash 9.*

* Официально продукт называется Flash CS3. – *Примеч. науч. ред.*

Код

В данной книге много ActionScript 3.0 (попросту говоря, кода). Во многих рецептах предлагаются специальные классы, которые, на наш взгляд, бесценны. Уверены, что и вы найдете их полезными.

Скачать библиотеки ActionScript 3.0 полностью можно по адресу *http://www.rightactionscript.com/ascb*. После загрузки файлы библиотеки надо добавить в путь к классам ваших проектов. Инструкции по настройке пути к классам ActionScript 3.0 также можно найти на этом сайте.

Аудитория книги

Подходит ли вам эта книга? Конечно, мы надеемся, что да. Но чтобы исключить сомнения, сделаем краткий обзор целей, которые она преследует, и знаний, необходимых для работы с ней.

Что надо знать

Предполагается, что читатель уже хорошо знаком с продуктом или продуктами, которые применяет для создания информационного наполнения Flash Player. Здесь не рассматриваются основы Flex SDK, Flex Builder или Flash. Данная книга будет вам полезнее, если вы сначала освоите азы работы с этими продуктами, в частности научитесь компилировать и развертывать проект.

Кроме того, прежде чем читать эту книгу, не помешает узнать базовые принципы программирования. Мы много говорим об основных приемах программирования в контексте решения конкретных задач (например, как перебрать все элементы массива), но все-таки это не подробное руководство по базовым навыкам программирования.

Для кого написана эта книга

Для всех разработчиков на ActionScript 3.0. Надеемся, что все – от новичка до эксперта – откроют в ней что-то полезное для себя. Она идеально подходит для тех, кому надо быстро найти решение.

Кому эта книга ни к чему

Данная книга посвящена ActionScript 3.0 – языку программирования, который выполняется во Flash Player. По большей части в ней не обсуждаются подробно решения, работающие на серверной стороне, или другие языки программирования, которые могут применяться на стороне клиента. Так, здесь рассматривается код на ActionScript для работы с Flash Remoting (технологией для осуществления удаленных вызовов процедур) и приводятся примеры, но не объясняется, как писать соответствующий код, работающий на стороне сервера (например, компонент ColdFusion). Аналогично обсуждается применение Action-

Script для вызова функций JavaScript, но не рассказывается, как писать JavaScript. Если вы ищете книгу по ActionScript 3.0, то она перед вами. Но подробное обсуждение тем, выходящих за рамки ActionScript, надо искать в других источниках.

Структура книги

Книга состоит из двадцати четырех глав и одного приложения.

Глава 1 «Основы ActionScript»

Базовые понятия программирования, такие как операторы цикла, таймеры и т. д.

Глава 2 «Специальные классы»

Написание специальных классов для ActionScript 3.0.

Глава 3 «Среда времени выполнения»

Получение информации об операционной системе, устройстве и версии проигрывателя, а также о системе безопасности.

Глава 4 «Числа и математические операции»

Работа с числами в ActionScript, включая синтаксический разбор из строк, преобразование в форматированные строки и применение различных систем счисления.

Глава 5 «Массивы»

Работа с индексированными наборами данных, которые называют массивами, – от добавления и удаления элементов до сортировки.

Глава 6 «Список отображения»

Применение отображаемых объектов для вывода визуальных данных на экран.

Глава 7 «Программное создание изображений и масок»

Программное создание изображений и масок с помощью ActionScript.

Глава 8 «Растровые изображения»

Низкоуровневая работа с растровыми изображениями.

Глава 9 «Текст»

Все о тексте – от отображения до загрузки и форматирования.

Глава 10 «Фильтры и трансформации»

Применение различных эффектов к отображаемым объектам с помощью трансформаций (цветовых и геометрических) и фильтров, таких как тени, фаски и даже эффекты тиснения и выделения краев.

Глава 11 «Программная анимация»

Анимирование отображаемых объектов с помощью ActionScript.

Глава 12 «Строки»

Работа со строковыми данными – от поиска подстрок до работы с Unicode.

Глава 13 «Регулярные выражения»

Создание собственных регулярных выражений для сопоставления шаблонов и строк.

Глава 14 «Даты и время»

Работа с датами и временем, включая преобразования между часовыми поясами с помощью таймеров и форматирование дат.

Глава 15 «Программирование звука»

Работа с аудиоданными, включая загрузку MP3, чтение тегов ID3 и отображение звуковых волн.

Глава 16 «Работа с видеоданными»

Программирование для Flash-видео.

Глава 17 «Хранение данных»

Применение совместно используемых объектов для хранения данных на клиентском компьютере.

Глава 18 «Обмен информацией между роликами»

Использование локальных соединений для обмена данными между содержимым, выполняющимся в экземпляре Flash Player на одном компьютере.

Глава 19 «Отправка и загрузка данных»

Организация двунаправленного обмена данными между веб-сервером и Flash Player.

Глава 20 «XML»

Работа с XML с использованием поддержки E4X во Flash Player.

Глава 21 «Веб-сервисы и удаленное взаимодействие во Flash»

Работа с вызовами удаленных процедур, основанная на технологиях веб-сервисов и Flash Remoting.

Глава 22 «Создание интегрированных приложений»

Использование программного интерфейса Flash Player для интегрирования его содержимого с хост-приложением, например для вызова функций JavaScript из ActionScript или для вызова функций ActionScript из JavaScript.

Глава 23 «Работа с файлами»

Передача и загрузка файлов.

Глава 24 «Работа с сокетами»

Применение сетевых соединений для передачи XML и двоичных данных в приложениях с малым периодом ожидания.

Приложение «Таблица кодов Unicode для символов Latin 1»

В приложении приводится список кодов для кодировки Latin 1 с эквивалентными им Unicode-кодами в диапазоне от U+0000 до U+00FF (т. е. управляющих символов C0, базовых символов Latin, управляющих символов C1 и дополнительных символов Latin 1).

Как работать с этой книгой

Пусть она будет вашим другом и советчиком. Не ставьте ее на полку. Положите на свой рабочий стол, чтобы как можно чаще обращаться к ней за помощью. Засомневавшись, что понимаете, как работает тот или иной элемент кода или как решить задачу, возьмите эту книгу и найдите нужный(ые) рецепт(ы). Ее формат призван помочь читателю быстро получить ответы на свои вопросы. А поскольку это книга, то она никогда не посмеется над вами и вашими вопросами. А важных или неважных вопросов не бывает.

Ее можно прочитать от корки до корки, но мы советуем обращаться к ней, когда понадобится ответ на конкретный вопрос. Она не будет мучить вас лекциями по теории, а постарается помочь решить поставленные задачи. Эта книга для полевых работ, а не для исследовательской лаборатории.

Типографские обозначения

В данной книге приняты следующие типографские обозначения:

Обычный текст

Показывает заголовки меню, опции меню, кнопки меню и «быстрые» клавиши (такие как Alt и Ctrl)

Курсив

Показывает новые термины, URL, адреса электронной почты, имена файлов, расширения файлов, имена путей, каталоги и утилиты UNIX.

`Моноширинный`

Показывает команды, параметры, переключатели, переменные, атрибуты, ключи, функции, типы, классы, пространства имен, методы, модули, свойства, параметры, значения, объекты, события, обработчики событий, теги XML и HTML, макросы, содержимое файлов или результат выполнения команд.

`Моноширинный полужирный`

Показывает команды или другой текст, который должен быть введен пользователем как есть.

Моноширинный курсив

Показывает текст, который должен быть заменен пользовательскими значениями.

При описании свойств или методов объектов и классов действуют следующие обозначения:

- При записи констант уровня класса как имя класса, так и свойство записываются моноширинным шрифтом, потому что они должны вводиться «как есть», например `Event.COMPLETE`.
- Для свойств уровня экземпляра экземпляр класса или объекта записывается *моноширинным курсивом*, потому что они должны быть заменены конкретным экземпляром. Само свойство записывается моноширинным шрифтом и должно вводиться без изменений, например *`Button.enabled`*.
- Имена методов и функций, а также класс или объект, к которым они относятся, всегда выделяются курсивом и оканчиваются круглыми скобками: *BitmapData.clone( )*. В оперативной справке можно узнать, включать ли имя класса (т. е. относится ли метод к так называемым статическим методам), как в *String.fromCharCode( )*, или заменять его именем экземпляра, как в *exampleBitmap.clone( )*.
- Для краткости при обсуждении свойства или метода мы всегда опускаем имя его класса. Например, когда речь идет о свойстве `htmlText` класса `TextField`, если говорится «зададим свойство `htmlText`», вы должны по контексту понимать, что имеется в виду «зададим свойство *`exampleTextField`*`.htmlText`, где *`exampleTextField`* — это идентификатор конкретного текстового поля».

Кроме того, чтобы привлечь внимание читателя и оживить изложение, в текст встроены следующие примечания:

Это подсказка, предложение или совет общего характера.

Это предупреждение или предостережение.

Примеры кода

«ActionScript 3.0. Сборник рецептов» призван помочь разработчику в его труде. Вы можете заимствовать код из этой книги и поместить его в свои программы и документацию. Если в программе используется несколько блоков кода из этой книги, то обращаться к нам за разрешением не надо. А вот продажа или распространение CD-ROM с примерами из книг O'Reilly *требует* специального разрешения. Вы можете

свободно ссылаться на книгу и цитировать примеры кода, но для включения больших фрагментов кода из этой книги в документацию вашего продукта *требуется* наше согласие.

Будем благодарны, но не настаиваем на указании авторства. Обычно ссылка на источник включает название, автора, издателя и ISBN. Например: «ActionScript 3.0 Cookbook, by Joey Lott, Darron Schall, and Keith Piters. Copyright 2007 O'Reilly Media, Inc., 978-0-596-52695-5».

Если вам кажется, что использование вами примеров кода выходит за рамки, оговоренные выше, не стесняйтесь обратиться к нам по адресу *permissions@oreilly.com*.

Сборники рецептов O'Reilly

Где взять рецепт решения программистской задачи? Возьмите сборник от O'Reilly, и больше вам ничего не понадобится. В каждом из них вы найдете сотни сценариев, программ и последовательностей команд, которые вы можете использовать для решения конкретных задач.

Общая формула рецептов из этого сборника такова:

Задача

Все задачи, рассматриваемые здесь, четко сформулированы, конкретны и реальны.

Решение

Решение доступно для понимания и реализации.

Обсуждение

Обсуждение разъясняет суть задачи и решения. В нем также приводится пример кода, показывающий, как сделать все наилучшим образом. А самое замечательное, что все примеры, приведенные в сборнике, можно скачать с веб-сайта книги по адресу *http://www.oreilly.com/catalog/actscpt3ckbk*.

См. также

Раздел «См. также» отсылает читателя к дополнительной информации по рассматриваемой в рецепте теме – другим рецептам, книгам (включая изданные не O'Reilly), сайтам и т. д.

Дополнительную информацию о серии «Сборники рецептов O'Reilly» можно найти на сайте *http://cookbooks.oreilly.com*.

Safari® Enabled

Если на обложке книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи

лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

Как с нами связаться

Комментарии и вопросы издателю по этой книге направляйте по адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

У этой книги есть веб-страница, на которой перечислены опечатки, примеры и другая дополнительная информация. Ее можно найти по адресу:

http://www.oreilly.com/catalog/actscpt3ckbk

Авторы создали сайт этой книги по адресу:

http://www.rightactionscript.com/ascb

Чтобы прокомментировать или задать технические вопросы по книге, присылайте электронные письма по адресу:

bookquestions@oreilly.com

Для получения дополнительной информации о наших книгах, конференциях, информационных центрах и O'Reilly Network посетите наш веб-сайт по адресу:

http://www.oreilly.com

О научных редакторах

Стейси Малкахи (Stacey Mulcahy) – разработчик во Flex и Flash, у которой однажды возник роман с Macromedia Director. Когда она не создает многофункциональные интернет-приложения, которые нравятся даже дизайнерам, ее можно найти в учебных классах, где в качестве инструктора по Flash она учит других делать то же самое. Она властвует на своем блоге *http://www.bitchwhocodes.com*, посвященном Flash.

Сэм Роббинс (Sam Robbins) работает с Flash и ActionScript более шести лет и почти два года экспериментирует с Flex. Все это время основные свои силы он направляет на разработку многофункциональных интернет-приложений. Кроме того, он организатор подгруппы в Бостонской группе пользователей Flash Platform (Boston Flash Platform User Group, BFPUG), которая ежемесячно тестирует шаблоны проектирования.

В свободное время Сэм пытается вести свой блог (*http:// pixelconsumption.com*), играет в Xbox и стрижет газон. Он живет в Массачусетсе.

Стивен Шелтер (Steven Schelter) – программист, в настоящее время работающий в Schematic. Прекрасно владеет различными языками сценариев и программирования серверной и клиентской частей, но основное его занятие – разработка на ActionScript. Стивен перешел в индустрию интерактивных систем, поскольку Flash позволил сочетать его дизайнерские способности со знаниями математики и логистики. Он также интересуется 3D-моделированием, искусственным интеллектом и интерактивными пользовательскими приложениями.

Роджер Браунштейн (Roger Braunstein) – разработчик и дизайнер из Бруклина. Одержим красивым кодом, разработкой анимации, программированием графических объектов и видеоиграми. Также интересуется кулинарией, фотографией, велосипедами и 8-разрядной музыкой. Он надеется, что когда-нибудь у него будет настоящий сайт по адресу *http://www.partlyhuman.com*.

Мюон Ти Ван (Muon Thi Van) – разработчик программного обеспечения из компании Schematic, интерактивного агентства полного цикла, имеющего офисы в Лос-Анджелесе и Нью-Йорке. Она является архитектором, разработчиком и дизайнером множества Flash-приложений и игр для Всемирной паутины и мобильных устройств, выступает на профессиональных конференциях, таких как Flashbelt и Flashforward. Мюон получила степень бакалавра по вычислительной технике в Северо-Западном университете.

Дэниел Вильямс (Daniel Williams) – разработчик программного обеспечения компании Schematic, Нью-Йорк, где живет и дышит ActionScript. Он даже видит его во сне. Кроме этого он активно интересуется физикой, быстрыми двухколесными транспортными средствами и человеческим мозгом. Время от времени делится своими мыслями и опытом на своем сайте с неподходящим названием *http://www.danieldoesdallas.com*.

Благодарности

Эта книга представляет собой результат работы трех авторов, Джои Лотта, Деррона Шалла и Кейта Питерса, чьи имена вы видите на обложке. Однако, безусловно, книга бы не состоялась без помощи многих людей, которые не значатся среди авторов. Мы хотели бы выразить им нашу коллективную признательность.

Спасибо Чаку Топореку (Chuck Toporek) за его преданность, терпение и фантастическое редактирование.

Также спасибо Стиву Вейсу (Steve Weiss) за веру в книгу и ее авторов. Мы благодарны за неизменно полезные и позитивные советы.

Спасибо Тиму О'Рейли (Tim O'Reilly) за высокую планку в технической литературе и за то, что сделал возможным выход этой книги в таком формате. Для нас было честью работать с такой прогрессивной и передовой компанией.

Без работы всего редакторского и технического состава O'Reilly эта книга не была бы там, где она находится сейчас. Спасибо всем, кто вложил в нее свое время и энергию.

Мы хотели бы поблагодарить нашего агента Марго Хатчинсон (Margot Hutchinson) из Waterside Productions за помощь в согласовании всех необходимых деталей.

Помощь команд Flex, Flash и Flash Player из Adobe всегда бесценна и чрезвычайно полезна. Они отвечают на наши вопросы, а мы можем ответить на ваши. Спасибо всем в Adobe.

Также хотим выразить благодарность всем научным редакторам, при содействии которых эта книга обрела лучшую форму, какая только возможна.

Джои Лотт

Спасибо вам, Кейт и Деррон, за помощь в создании книги. Для меня было честью работать с двумя самыми выдающимися экспертами в этой области.

Я также хотел бы выразить признательность моим друзьям и семье за их поддержку и энтузиазм. Я благодарен судьбе за все.

Деррон Шалл

Спасибо тебе, Джои, за то, что предоставил мне возможность заполнить эти страницы. Кейт, и тебе спасибо за совместную работу. Писать с вами было просто замечательно, и я горд быть вашим соавтором. Чак и Стив, ваша поддержка неоценима. Я, несомненно, не справился бы со всем этим без вас.

Спасибо моей красавице жене Джен за понимание и бесконечное терпение. Я люблю тебя.

Всей моей семье, особенно моему деду Эдвину, спасибо за вашу поддержку и веру в меня.

Кейт Питерс

Спасибо тебе, Джои, за возможность стать частью этого проекта. Также благодарю Стива, Чака и Деррона за помощь в ходе работы и, как обычно, Казуми и Кристин за то, что слишком часто им приходится терпеть меня, уткнувшегося в монитор.

1

Основы ActionScript

1.0. Введение

ActionScript позволяет создавать Flash-приложения, которые делают практически все, что только можно себе представить. Но прежде чем углубиться в безграничные возможности, обратимся к основам. Положительный момент в том, что команды ActionScript соответствуют четкой схеме, имеют схожий синтаксис и концепции. Освоение азов грамматики – первый шаг к овладению ActionScript.

В данной главе рассматриваются распространенные задачи и проблемы, связанные с базовым знанием ActionScript. Независимо от того, новичок вы или эксперт (или занимаете промежуточное положение на этой шкале), эти рецепты помогут вам справиться с ситуациями, возникающими в каждом проекте на ActionScript.

Предполагается, что на вашем компьютере успешно установлена копия Flex Builder 2. Также не будет лишним некоторый опыт работы с предыдущей версией ActionScript.

При запуске Flex Builder 2 появляется стартовая страница Eclipse IDE с различными вариантами начала работы и получения более подробной информации о Flex и ActionScript 3 в виде ссылок на документацию, учебные пособия и т. д. Эту страницу можно закрыть, щелкнув маленький «x» на ее корешке (tab). Теперь вы находитесь в самой Eclipse IDE, готовой к написанию кода. А что же дальше?

Flex Builder 2 позволяет создавать три типа проектов: проект Flex, проект Flex Library и проект ActionScript. Разница в том, что проекты Flex имеют доступ ко всей инфраструктуре Flex Framework, которая включает все компоненты Flex, управление компоновкой, переходы, стили, темы, привязку данных и все остальные возможности, предназначенные для создания многофункциональных интернет-приложений Flex.

Приложения Flex пишутся на языке MXML (форма XML), который описывает компоновку и отношения между компонентами. ActionScript служит для описания бизнес-логики. Хотя полученные здесь знания по ActionScript могут применяться и в приложениях Flex, эта книга ориентирована исключительно на проекты ActionScript.

Итак, если вы знакомы с Flash 8 или предыдущими версиями Flash IDE, то будете немного озадачены, впервые открыв Flex Builder 2. Здесь нет ни временной диаграммы, ни библиотеки, ни инструментов для рисования или палитры цветов. Очень многое придется делать посредством кода, вот почему это *проект ActionScript*, а не *проект Flash*. Поэтому сначала рассмотрим, как создавать проект, а затем перейдем к вводу первых выражений ActionScript.

1.1. Создание проекта ActionScript

Задача

Вы запустили Flex Builder 2 и хотите создать проект ActionScript.

Решение

Для настройки своего проекта воспользуйтесь мастером создания проекта ActionScript – New ActionScript Project Wizard.

Обсуждение

Проект ActionScript обычно состоит из минимум одного файла класса и папки *bin*, содержащей файлы SWF и HTML, создаваемые компилятором. Также он включает множество внутренних настроек, сообщающих компилятору, где что находится и как все это компилировать. В случае применения New ActionScript Project Wizard большую часть работы за вас делает Flex Builder 2. Существует несколько способов запустить этот мастер. Можно воспользоваться меню File→New→ActionScript Project (Файл→Новый→Проект ActionScript) или щелкнуть по кнопке New (Новый) в верхнем левом углу и выбрать из списка доступных проектов Проект ActionScript (ActionScript Project). Также можно щелкнуть по маленькой стрелке рядом с кнопкой New (Новый), в результате чего откроется этот же список.

Любой из этих путей приведет вас к New ActionScript Project Wizard. Здесь надо ввести имя проекта, например *ExampleApplication*. После создания проекта основному файлу приложения автоматически будет присвоено имя, совпадающее с именем проекта, с расширением *.as*.

Щелкнув по кнопке Next (Далее), можно задать пути к классам (custom class paths), дополнительные библиотеки и определить другую выходную папку вместо заданной по умолчанию *bin*. Здесь больше ничего делать не надо, поэтому нажмите кнопку Finish (Закончить), чтобы выйти из мастера.

Теперь Flex Builder 2 создаст необходимые папки и файлы для вашего проекта и применит все стандартные настройки компилятора. В окне Navigator должен появиться проект *ExampleApplication*, содержащий папку *bin* и файл класса *ExampleApplication.as*. Обратите внимание, что этот основной файл класса был создан автоматически и открыт для редактирования в окне кода. Также в окне Outline можно увидеть представление класса в виде дерева, включая его методы, свойства и все выражения импорта.

Чтобы запустить новое приложение, можно нажать одну из двух кнопок панели инструментов. На первой изображен жук, и по ее нажатию начинается отладка приложения, т. е. она включает некоторую дополнительную информацию о состоянии приложения и обеспечивает вывод сообщений в консоль. На соседней кнопке, которая запускает приложение, изображен круг со стрелкой. В обоих случаях будут созданы файлы *.swf* и HTML, и HTML-файл будет запущен в вашем броузере.

Конечно, пока что в приложении ничего нет, т. е. мы тестируем пустой файл *.fla* во Flash IDE. Но все равно сделайте это, просто чтобы проверить, что все настроено правильно. Вы должны получить пустую вебстраницу с голубым фоном.

1.2. Настройка свойств приложения

Задача

Требуется изменить размеры выходного файла *.swf* или его фоновый цвет, частоту смены кадров и т. д.

Решение

Задайте свойства как аргументы компилятора ActionScript или метаданные в файле класса.

Обсуждение

В отличие от предыдущих версий Flash, компилятор ActionScript 3.0 фактически представляет собой компилятор, работающий из командной строки. Можно было бы создавать собственные классы и каталоги и запускать компилятор из командной строки с длинной цепочкой параметров. Однако намного проще позволить Eclipse отслеживать все эти параметры, добавлять их и запускать компилятор, когда вы попросите об этом.

При создании нового проекта ActionScript задаются параметры по умолчанию. В результате получается файл *.swf* с размером сцены 500×375 пикселов, частотой 24 кадра в секунду (frames per second, fps) и голубым фоном, который вы уже видели. Все эти настройки можно изменить и добавить многое другое. Естественно, это можно сделать по-разному.

Первый способ изменения настроек компилятора состоит в том, чтобы задать аргументы компилятора ActionScript. Делается это так: щелкаем правой кнопкой мыши по проекту в окне Navigator и выбираем пункт меню Properties (Свойства). Далее в списке слева выбираем ActionScript Compiler (Компилятор ActionScript). Это даст возможность изменить несколько аспектов работы компилятора. Найдите текстовое поле Additional compiler arguments (Дополнительные аргументы компилятора). Все, что вводится в это поле, передается как аргумент в командной строке непосредственно компилятору.

Вот самые распространенные аргументы, которые вы, вероятно, будете использовать:

```
-default-size ширина высота
-default-background-color цвет
-default-frame-rate fps
```

Они вводятся точно, как представлены, только вместо аргументов указываются конкретные числовые значения, например:

```
-default-size 800 600
-default-background-color 0xffffff
-default-frame-rate 31
```

Первая строка задает размер результирующего *.swf* в 800×600 пикселов, вторая – белый цвет фона, а последняя – частоту смены кадров 31 fps. Несколько аргументов размещаются просто друг за другом в одну строку:

```
-default-size 800 600 -default-frame-rate 31
```

В файлах справки Flex Builder 2 в разделе *mxmlc options* (*параметры mxmlc*) можно найти полный список всех аргументов командной строки, которые могут здесь применяться.

Второй способ изменить эти свойства – через метаданные в файле основного класса. Метаданные состоят из выражений, которые не интерпретируются непосредственно как ActionScript, но с их помощью компилятор определяет, как компилировать окончательные выходные файлы. Запись метаданных, эквивалентная предыдущему примеру, выглядит следующим образом:

```
[SWF(width="800", height="600", backgroundColor="#ffffff", frameRate="31")]
```

Эта строка помещается в блок основного пакета, но вне описаний классов (обычно непосредственно перед или после всех операторов импорта).

1.3. Где размещать код ActionScript

Задача

Вы создали новый проект ActionScript и не знаете, куда поместить его код, чтобы он выполнялся соответствующим образом.

Решение

Код ActionScript размещается в конструкторе и дополнительных методах класса.

Обсуждение

В ActionScript 1.0 и 2.0 существовало множество вариантов размещения кода: на временной диаграмме, на кнопках и клипах, на временной диаграмме клипов, во внешних файлах *.as*, включенных директивами *#include*, или во внешних файлах классов. ActionScript 3.0 полностью основан на классах, поэтому весь код должен размещаться в методах классов проекта.

При создании нового проекта ActionScript основной класс автоматически создается и открывается в окне Code. Он должен выглядеть примерно так:

```
package {
    import flash.display.Sprite;

    public class ExampleApplication extends Sprite
    {
        public function ExampleApplication()
        {

        }
    }
}
```

Даже если вы хорошо знакомы с классами в ActionScript 2.0, здесь есть несколько нововведений. В главе 2 этот вопрос рассматривается подробно, а сейчас ознакомимся с базовыми аспектами.

Первое, что бросится в глаза, – слово *package* (пакет) в начале кода. Пакеты служат для группировки классов со взаимосвязанной функциональностью. В ActionScript 2.0 пакеты подразумевались в структуре каталогов, предназначенных для размещения файлов классов. Однако в ActionScript 3.0 пакеты необходимо определять явно. Например, может существовать пакет утилитных классов, что описывается так:

```
package com.as3cb.utils {
}
```

Если имя пакета не указано, класс создается по умолчанию в пакете верхнего уровня. Но все равно ключевое слово *package* и фигурные скобки должны быть включены.

Далее располагаются все выражения `import` (импортировать). Импорт класса делает его доступным в коде файла и обеспечивает возможность сокращенной записи, чтобы каждый раз, когда необходимо сослаться на этот класс, не приходилось вводить полное имя пакета. Пример директивы `import`:

```
import com.as3cb.utils.StringUtils;
```

После этого на класс *StringUtils* можно ссылаться напрямую без указания его полного имени. Как показано в предыдущем примере, класс *Sprite* надо будет импортировать из пакета *flash.display*, поскольку основной класс по умолчанию расширяет класс *Sprite*.

Далее идет основной класс `ExampleApplication`. Возможно, вы обратили внимание на ключевое слово *public* (открытый) перед описанием класса. В пакете не может быть закрытых (private) классов, но все-таки необходимо явно указать, что класс открытый (public). Заметьте, что основной класс расширяет класс *Sprite*. Сам *.swf* также представляет собой спрайт или клип, поэтому *.swf* можно загрузить в другой *.swf* и обращаться с ним так, как если бы это был всего лишь еще один вложенный спрайт или клип. Этот основной класс представляет *.swf* в целом, поэтому он должен расширять класс *Sprite* или любой другой класс, расширяющий *Sprite* (например, *MovieClip*).

Наконец, открытая функция (или метод в терминологии классов) с таким же именем, как и имя класса. Это делает ее *конструктором*. Конструктор класса выполняется автоматически при создании экземпляра класса. В данном случае он выполняется при загрузке файла *.swf* во Flash Player. Куда же надо поместить код, чтобы он был выполнен? Обычно начинают с размещения кода в методе-конструкторе. Вот очень простой пример, который отрисовывает случайные линии на экране:

```
package {
    import flash.display.Sprite;
    public class ExampleApplication extends Sprite {
        public function ExampleApplication() {
            graphics.lineStyle(1, 0, 1);
            for(var i:int=0;i<100;i++) {
                graphics.lineTo(Math.random() * 400, Math.random() * 400);
            }
        }
    }
}
```

Сохраните и запустите приложение. Броузер должен открыть результирующий HTML-файл и вывести на экран *.swf* с сотней случайных линий. Как видите, конструктор был выполнен, как только файл был загружен в проигрыватель.

На практике в конструкторе стараются оставить минимально необходимое количество кода. В идеале там находится только код для вызова другого метода, инициализирующего приложение. Более подробно об этих методах рассказывается в рецептах 1.13 и 1.14.

Теперь, когда вы знаете, куда вводить код, мы приводим букварь по терминологии для начинающих. Здесь даются краткие определения, чтобы наметить ориентиры для тех, кто никогда до этого не занимался программированием. Более полные определения можно найти в справочных файлах Flash.

Переменные

Переменные (*variables*) – это удобные контейнеры для данных в коде. Им можно присвоить любое имя (кроме тех, которые уже зарезервированы ActionScript), начинающееся с буквы, знака подчеркивания или знака доллара (но не числа). В справочных файлах, устанавливаемых с Flex Builder 2, содержится список зарезервированных слов. В переменных удобно сохранять промежуточные данные, например сумму чисел, или с их помощью ссылаться на что-то, например текстовое поле или спрайт. Переменные *объявляются* ключевым словом `var` при первом упоминании в сценарии. Переменной можно присвоить значение с помощью знака равенства (=), который также называют *оператором присваивания*. Если переменная объявляется вне метода класса, то это *переменная класса*. Переменные класса, или *свойства*, могут иметь модификаторы доступа: `public`, `private`, `protected` (защищенный) или `internal` (внутренний). Закрытая переменная доступна только в пределах самого класса, тогда как открытые переменные доступны объектам других классов. Переменные `protected` доступны из экземпляра класса или экземпляра любого подкласса, а переменные `internal` доступны классам пакета. Если модификатор доступа не указан, по умолчанию применяется `internal`.

Функции

Функции – это блоки кода, которые что-то выполняют. Функцию можно *вызвать* или *инициировать* (т. е. выполнить ее) с помощью ее имени. Если функция является частью класса, ее называют *методом* класса. К методам применимы все те же модификтаоры, что и к свойствам.

Область видимости

Область видимости переменной описывает, когда и где код может использовать переменную в ролике. Область видимости определяет продолжительность жизни переменной и ее доступность другим блокам кода. Область видимости определяет, как долго существует переменная и откуда можно задавать или извлекать ее значение. Область видимости функции определяет, где и когда функция доступна другим блокам кода. Вопросы области видимости обсуждаются в рецепте 1.13.

Обработчик события

Обработчик – это функция или метод, который выполняется в ответ на некоторое событие, например щелчок мыши, нажатие клавиши или перемещение головки воспроизведения на временной диаграмме.

Объекты и классы

Объект – это то, чем можно манипулировать программно в ActionScript, например спрайт. Есть другие типы объектов, которые,

например, применяются для работы с цветами, датами и текстовыми полями. Объекты – это экземпляры *классов*. Это означает, что класс является шаблоном для создания объектов, и объект – это конкретный экземпляр данного класса. Если запутались, давайте обратимся к биологии: вы можете рассматривать себя как объект (экземлпяр), принадлежащий к общему классу людей.

Методы

Метод – это функция, ассоциированная с объектом, которая осуществляет над ним некоторые действия. Например, метод *replaceSelectedText( )* объекта текстовое поле может служить для замены выделенного текста в поле.

Свойства

Свойство – это атрибут объекта, который может быть считан и/или задан. Например, горизонтальное положение спрайта определяется его свойством `x`, которое может быть и проверено, и задано. С другой стороны, свойство `length` текстового поля, которое показывает число символов в поле, может быть проверено, но не может быть задано напрямую (однако на него можно воздействовать косвенно, добавляя или удаляя текст из поля).

Выражения

Команды ActionScript вводятся как последовательности одного или более *выражений*. Выражение может указать головке воспроизведения перейти к определенному кадру или изменить размер спрайта. Большинство выражений ActionScript оканчиваются точкой с запятой (`;`). В этой книге термины *выражение* и *действие* употребляются взаимозаменяемо.

Комментарии

Комментарии – это примечания в коде, предназначенные для людей и игнорируемые Flash. В ActionScript однострочные комментарии начинаются с `//` и заканчиваются автоматически с окончанием строки. Многострочные комментарии начинаются с `/*` и оканчиваются `*/`.

Интерпретатор

Интерпретатор ActionScript – это часть Flash Player, которая проверяет ваш код и пытается понять и выполнить его. Строгое соблюдение грамматических правил ActionScript гарантирует, что интерпретатор сможет без труда понять код. Если интерпретатор встречает ошибку, он часто дает сбой без каких-либо уведомлений, просто отказываясь выполнять код без формирования специального сообщения об ошибке.

Не переживайте, если не понимаете всех тонкостей. Все решения рецептов можно применять, не вдаваясь в технические подробности, и этот словарик должен помочь разобраться с терминологией.

См. также

Рецепты 1.13 и 1.14

1.4. Как трассировать сообщение

Задача

Требуется вывести в консоль сообщение или некоторое значение во время выполнения.

Решение

Используйте функцию *trace*: передайте в нее данные, выполните приложение и ищите сообщение в консоли (Console) Eclipse.

Обсуждение

С помощью *trace* можно вывести сообщение, значение переменной или практически любые данные точно так же, как это делалось в предыдущих версиях ActionScript. Несколько примеров:

```
trace("Hello, world");
trace(userName);
trace("My name is " + userName + ".");
```

Теперь *.swf* выполняется во внешнем броузере, и кое-кто может подумать, что отследить эти сообщения нельзя. К счастью, это возможно. Данную функциональность во Flex Builder 2 обеспечивает окно Console, эквивалентное панели Output во Flash IDE. Оно не открывается сразу при запуске Eclipse, но появляется в случае необходимости.

Единственное условие для применения *trace* и окна Console – использование режима Debug (Отладка) для тестирования приложения. Тем самым в *.swf* включаются некоторые дополнительные возможности, позволяющие возвращать информацию в Console и передавать любые сообщения. Следующий класс создает переменную, присваивает ей значение и трассирует еще кое-какие строковые данные:

```
package {
    import flash.display.Sprite;

    public class ExampleApplication extends Sprite {
        public function ExampleApplication() {
            var userName:String = "Bill Smith";
            trace("My name is " + userName + ".");
        }
    }
}
```

При запуске отладки приложение открывается, как обычно, в броузере. Закройте броузер и вернитесь в Eclipse. Теперь окно Console открыто и в нем отображаются выведенные при помощи *trace( )* данные.

Для запуска отладочной версии приложения должна быть установлена отладочная версия Flash Player. Если данной версии нет, появится сообщение об ошибке, уведомляющее об этом. Скачать отладочную версию Flash Player можно по адресу *http://www.adobe.com/support/flashplayer/downloads.html*.

Кроме того, отладочная версия Flash Player может записывать результат трассировки в файл. При записи Flash Player руководствуется настройками, указанными в файле *mm.cfg*, который хранится в следующих каталогах:

Операционная система	**Местоположение**
Windows XP	C:\Documents and Settings\[*имя пользователя*]\mm.cfg
Windows 2000	C:\mm.cfg
Mac OS X	MacHD:Library:Application Support:macromedia:mm.cfg

Файл *mm.cfg* обеспечивает возможность задавать следующие переменные:

`TraceOutputFileEnable`

Может принимать значения `0` (не записывать результат трассировки в файл) или `1` (записывать в файл).

`TraceOutputFileName`

Путь к файлу записи. Если значение не задано, содержимое записывается в файл *flashlog.txt*, находящийся в том же каталоге, что и *mm.cfg*.

`ErrorReportingEnable`

Может принимать значения `0` (не записывать ошибки в файл) или `1` (записывать ошибки в файл). По умолчанию равна `0`.

`MaxWarnings`

Максимальное количество ошибок, записываемое в файл. Если задано значение `0`, ограничения нет. Если задано большее значение, то все ошибки сверх этого количества не записываются в файл.

Как минимум, чтобы включить возможность записи в файл, *mm.cfg* должен содержать следующее:

```
TraceOutputFileEnable=1
```

Если задаются несколько переменных, все они располагаются в разных строках:

```
TraceOutputFileEnable=1
TraceOutputFileName=C:\flex.log
```

1.5. Обработка событий

Задача

Необходимо, чтобы код выполнялся неоднократно.

Решение

Добавьте слушатель события *enterFrame* и назначьте метод-обработчик.

Обсуждение

В ActionScript 2.0 обрабатывать событие *enterFrame* было крайне просто. Требовалось всего лишь создать функцию *onEnterFrame внутри клипа*, и она автоматически вызывалась при каждой смене кадров. В ActionScript 3.0 предоставляются намного большие возможности управления различными событиями в *.swf*, но для доступа к ним требуется немного больше усилий.

Если вы хорошо знакомы с классом *EventDispatcher* из ActionScript 2.0, то работа с методом обработки событий ActionScript 3.0 не вызовет у вас трудностей. Надо сказать, *EventDispatcher* превратился из внешнего класса в базовый класс всех интерактивных объектов, таких как спрайты.

Чтобы отреагировать на событие *enterFrame*, надо указать приложению слушать это событие и определить, какой метод должен вызываться при его возникновении. Это делается с помощью метода *addEventListener*, который выглядит так:

```
addEventListener(type:String, listener:Function)
```

В файлах справки можно найти дополнительные параметры, но это минимальная реализация.

Параметр *type* – это тип ожидаемого события, в данном случае строка «`enterFrame`». Однако такое обращение со строковыми литералами делает код чреватым ошибками, которые компилятор не сможет перехватить. Если случайно ввести, например, «`enterFrane`», то приложение будет просто слушать событие «`enterFrane`». Чтобы защититься от этого, рекомендуется использовать статические свойства класса *Event* (Событие). Класс *Event* уже должен быть импортирован, поэтому можно вызывать его метод *addEventListener* следующим образом:

```
addEventListener(Event.ENTER_FRAME, onEnterFrame);
```

Если теперь по ошибке написать `Event.ENTER_FRANE`, то компилятор пожалуется, что такого свойства не существует.

Второй параметр, `onEnterFrame`, ссылается на другой метод класса. Обратите внимание, что в ActionScript 3.0 нет требования, чтобы этот ме-

тод назывался именно `onEnterFrame`. Однако составлять имя метода-обработчика из `on` и имени ожидаемого события – это общепринятая традиция. При вызове этот метод принимает экземпляр класса *Event*, поэтому потребуется импортировать этот класс и определить метод, чтобы он принимал объект события:

```
import flash.events.Event;
private function onEnterFrame(event:Event) {
}
```

Объект `event` содержит информацию о событии, которая может пригодится при его обработке. Даже если он не используется, все равно надо настроить обработчик на его прием. Те, кто знает версию *EventDispatcher* в ActionScript 2.0, заметят здесь различие в реализации. В предыдущей версии существовала проблема с областью видимости функции, предназначенной для обработки события, для решения которой часто требовалось применение класса *Delegate*. В ActionScript 3.0 областью видимости метода-обработчика остается его класс, поэтому нет необходимости в классе *Delegate* для корректировки.

Вот простое приложение для последовательной отрисовки случайных линий, в котором задействованы все концепции, обсуждаемые в этом рецепте:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class ExampleApplication extends Sprite {

        public function ExampleApplication() {
            graphics.lineStyle(1, 0, 1);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
        private function onEnterFrame(event:Event):void {
            graphics.lineTo(Math.random() * 400, Math.random() * 400);
        }
    }
}
```

1.6. Реакция на события мыши и клавиатуры

Задача

Необходимо предусмотреть реакцию на события мыши или клавиатуры.

Решение

Организуйте прослушивание и обработку событий мыши или клавиатуры.

Обсуждение

Обработка событий мыши и клавиатуры очень похожа на обработку события *enterFrame*, обсуждаемого в рецепте 1.5, но требует некоторой дополнительной работы. В случае событий мыши основной класс приложения не будет получать их напрямую, поэтому он должен слушать их по другому объекту списка отображения. (Список отображения подробно рассмотрен в главе 5.) В следующем примере создается спрайт, добавляется в список отображения, и в нем отрисовывается прямоугольник.

```
package {
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class ExampleApplication extends Sprite {
        private var _sprite:Sprite;

        public function ExampleApplication() {
            _sprite = new Sprite();
            addChild(_sprite);
            _sprite.graphics.beginFill(0xffffff);
            _sprite.graphics.drawRect(0, 0, 400, 400);
            _sprite.graphics.endFill();
```

Обратите внимание, что имена событий мыши определяются в классе *MouseEvent* и методы-обработчики получают экземпляр класса *MouseEvent*, поэтому данный класс надо импортировать. Тогда можно добавить слушателей в этот спрайт:

```
            _sprite.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            _sprite.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        }
```

Далее определяем два метода-обработчика, `onMouseDown` и `onMouseUp`:

```
        private function onMouseDown(event:MouseEvent):void {
            _sprite.graphics.lineStyle(1, 0, 1);
            _sprite.graphics.moveTo(mouseX, mouseY);
            _sprite.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        }

        private function onMouseUp(event:MouseEvent):void
        {
            _sprite.removeEventListener(MouseEvent.MOUSE_MOVE, onMouseMove);
        }
```

Методы `onMouseDown` задают стиль линии нового спрайта и перемещают рисующий курсор в позицию, соответствующую положению мыши. Затем вводится третий слушатель событий мыши для события *MouseMove*.

Методы `onMouseUp` удаляют слушателя с помощью метода *removeEventListener*. Его синтаксис аналогичен *addEventListener*, но указывает классу прекратить слушать заданное событие.

И наконец, определяется onMouseMove, а класс и пакет закрываются:

```
        private function onMouseMove(event:MouseEvent):void {
            _sprite.graphics.lineTo(mouseX, mouseY);
        }
    }
}
```

Получилась простая управляемая событиями программа создания изображений.

События клавиатуры обрабатывать немного проще. Чтобы слушать и отвечать на события клавиатуры, объект, получающий события, всего лишь должен иметь фокус. Для основного приложения это делается путем добавления строки:

```
stage.focus = this;
```

В следующем примере показан простой класс, который слушает событие *keyDown* и выводит код символа для этой клавиши. Здесь также демонстрируется, как использовать некоторые данные, содержащиеся в объекте события, передаваемом в метод-обработчик. Обратите внимание, что для событий клавиатуры применяется класс *KeyboardEvent*.

```
package {
    import flash.display.Sprite;
    import flash.events.KeyboardEvent;

    public class ExampleApplication extends Sprite {
        public function ExampleApplication() {
            stage.focus = this;
            addEventListener(KeyboardEvent.KEY_DOWN, onKeyDown);
        }

        private function onKeyDown(event:KeyboardEvent):void {
            trace("key down: " + event.charCode);
        }
    }
}
```

См. также

Рецепт 1.5.

1.7. Применение математических операторов

Задача

Требуется время от времени изменять некоторые параметры, например разворот или положение спрайта.

Решение

Для изменения переменной или свойства обратитесь к составным операторам присваивания. А при пошаговом приращении или уменьшении – к префиксным или постфиксным операторам инкремента или декремента соответственно.

Обсуждение

Часто новое значение переменной или свойства зависит от предыдущего значения. Например, требуется переместить спрайт на 10 пикселов вправо от его текущего положения.

В выражении присваивания – любом выражении, содержащем оператор присваивания (знак равенства) – справа от знака равенства располагается вычисляемое выражение, а результат сохраняется в переменной или свойстве, находящихся в левой части. Следовательно, можно изменять значение переменной в правой части равенства и присваивать полученное новое значение этой же переменной в левой части.

Дальнейшее может удивить тех, кто еще помнит школьную алгебру, но для программистов приравнивать переменную самой себе плюс какое-то число – обычное дело:

```
// Добавляем 6 к текущему значению quantity и опять
// присваиваем это новое значение quantity.
// Например, если quantity была равна 4,
// это выражение присвоит ей значение 10.
quantity = quantity + 6;
```

Однако для математических операций часто удобнее применить один из *составных операторов присваивания*, которые сочетают математический оператор с оператором присваивания. Самыми распространенными составными операторами присваивания являются `+=`, `-=`, `*=` и `/=`. При их выполнении значение в левой части добавляется, вычитается, умножается или делится на значение переменной в правой части, а новое значение присваивается этой же переменной. Вот несколько примеров эквивалентных выражений.

Оба этих выражения добавляют 6 к существующему значению `quantity`:

```
quantity = quantity + 6;
quantity += 6;
```

Оба этих выражения вычитают 6 из существующего значения `quantity`:

```
quantity = quantity - 6;
quantity -= 6;
```

Оба этих выражения умножают `quantity` на `factor`:

```
quantity = quantity * factor;
quantity *= factor;
```

Оба этих выражения делят quantity на factor:

```
quantity = quantity / factor;
quantity /= factor;
```

Между символами, образующими составной оператор присваивания, не должно быть пробела. Кроме того, если переменная увеличивается или уменьшается на 1, можно использовать операторы инкремента или декремента (приращения или уменьшения на единицу).

Это выражение увеличивает quantity на 1:

```
quantity++;
```

и имеет тот же эффект, что и любое из следующих выражений:

```
quantity = quantity + 1;
quantity += 1;
```

Это выражение вычитает 1 из quantity:

```
quantity --;
```

и имеет тот же эффект, что и любое из следующих выражений:

```
quantity = quantity - 1;
quantity -= 1;
```

Операторы инкремента и декремента (-- и ++) могут находиться как перед, так и после переменной или свойства, над которым они осуществляют действие. Если они расположены перед операндом, их называют *префиксными операторами*, а если после операнда, то *постфиксными операторами*. Обе формы изменяют операнд одинаково, но в разные моменты времени. В некоторых обстоятельствах между ними фактически нет никакой разницы, но чаще всего они все-таки различны. Префиксный оператор меняет значение до вычисления оставшейся части выражения, а постфиксный – после выполнения всего остального выражения. Обратите внимание, что в первом примере приращение quantity происходит после вывода на экран его значения, тогда как во втором примере – до:

```
var quantity:Number = 5;
trace(quantity++);  // На экран выводится: 5
trace(quantity);    // На экран выводится: 6
var quantity:Number = 5;
trace(++quantity);  // На экран выводится: 6
trace(quantity);    // На экран выводится: 6
```

В нашей исходной задаче эти операторы можно применить для изменения свойства во времени. А здесь заданный спрайт при каждом вызове метода поворачивается на 5 градусов:

```
private function onEnterFrame(event:Event) {
  _sprite.rotation += 5;
}
```

Обратите внимание, что для правильной работы этого примера в ActionScript 3.0 надо добавить слушатель события `enterFrame` и определить этот метод как обработчик события. В рецепте 1.5 рассказывается, как обрабатывать событие `enterFrame`.

См. также

Рецепт 1.5.

1.8. Проверка равенства или сравнение значений

Задача

Необходимо проверить равенство двух значений.

Решение

Два значения можно сравнить при помощи операторов равенства (неравенства) или строгого равенства (строгого неравенства). Действительность числового значения проверяется при помощи функции *isNaN( )*.

Обсуждение

Операторы равенства всегда возвращают Булево значение, которое показывает, равны ли значения. Операторы равенства (и неравенства) могут быть как обычными, так и строгими. Обычные операторы равенства и неравенства проверяют, могут ли сравниваемые выражения быть приведены к одному и тому же значению после преобразования в один тип данных. Например, обратите внимание, что строка «6» и число 6 считаются эквивалентными, потому что перед сравнением строка «6» преобразуется в число 6:

```
trace(5 == 6);    // На экран выводится: false
trace(6 == 6);    // На экран выводится: true
trace(6 == "6");  // На экран выводится: true
trace(5 == "6");  // На экран выводится: false
```

Следует отметить, что в проекте с настройками по умолчанию приведенный пример кода даже не скомпилируется, потому что компиляция в этом случае происходит при установленном флаге `strict` (строгий), что заставляет компилятор строже проверять типы данных. Появится сообщение о сравнении величины типа *int* со *String*. Чтобы отключить флаг `strict`, необходимо отправиться в раздел ActionScript Compiler (Компилятор ActionScript) свойств проекта и отключить кнопку-флажок рядом с пунктом Enable compile-time type checking (-strict) («Включить контроль типов (строгий) во время компиляции»). Однако в большинстве проектов рекомендуется сохранять строгий контроль типов включенным, поскольку это обеспечивает более надежную защиту от ошибок по невнимательности.

Логический оператор неравенства (!=) возвращает false, если два значения равны, и true, если нет. В случае необходимости перед сравнением операнды преобразуются к одному типу данных:

```
trace(5 != 6);    // На экран выводится: true
trace(6 != 6);    // На экран выводится: false
trace(6 != "6");  // На экран выводится: false
trace(5 != "6");  // На экран выводится: true
```

Как и предыдущий, этот пример скомпилируется, только если отключен строгий контроль типов.

С другой стороны, если флаг strict отключен, строгое сравнение в каком-то разделе кода можно выполнить при помощи операторов строгого равенства и неравенства, === и !==. Эти операторы перед сравнением проверяют тип операндов. Если типы различны, оператор строгого равенства возвратит false, а оператор строгого неравенства – true.

```
trace(6 === 6);    // На экран выводится: true
trace(6 === "6");  // На экран выводится: false
trace(6 !== 6);    // На экран выводится: false
trace(6 !== "6");  // На экран выводится: true
```

Между оператором присваивания (=) и оператором равенства (==) есть большая разница. Если применить оператор присваивания вместо оператора равенства, то значение переменной изменится, а сравнения не будет.

Ошибочное применение операторов приводит к неожиданным результатам. В следующем примере quantity сначала равно 5. Поэтому можно ожидать, что последующее выражение *if* всегда будет давать в результате false, предотвращая выполнение функции *trace( )*:

```
var quantity:int = 5;
// Следующий код неверен. Должно быть
// if (quantity == 6), а не
if (quantity = 6) {
  trace("Rabbits are bunnies.");
}
trace("quantity is " + quantity);  // На экран выводится: quantity is 6
```

Однако в примере по ошибке вместо оператора равенства (==) применен оператор присваивания (=). То есть выражение *quantity = 6* присваивает переменной quantity значение 6, когда требуется проверить, равна ли quantity 6. В блоке *if* выражение *quantity = 6* интерпретируется как число 6. Поскольку любое отличное от нуля число в проверяющем выражении дает Булево true, вызывается *trace( )*. Необходимо подставить выражение *quantity == 6*. К счастью, компилятор ActionScript 3.0 достаточно разумен и распознает эту распространенную ошибку. Поэтому, хотя код по-прежнему компилируется, формируется предупреждение.

Тип элемента можно проверить с помощью оператора *is*:

```
var quantity:int = 5;
if (quantity is int) {
  trace("Yippee. It's an integer.");
}
```

Обратите внимание, что новые типы ActionScript 3.0 *int* и *uint* также будут распознаваться как *Number*.

Однако некоторые числовые значения являются недействительными. В результате выполнения следующего примера переменная `quantity` приравнивается `NaN` (константа, представляющая недействительные числа, сокращение от «нечисло» (not a number)), поскольку осмысленное вычисление невозможно:

```
var quantity:Number = 15 - "rabbits";
```

Несмотря на его название, `NaN` является распознаваемым значением типа *Number*:

```
trace(typeof quantity);   // На экран выводится: "number"
```

Поэтому, чтобы проверить, является ли переменная не просто числом, а действительным числом, попытаемся сделать так:

```
var quantity:Number = 15 - "rabbits";
if (quantity is Number) {
  // Хорошая попытка, но из этого ничего не выйдет
  if (quantity != NaN) {
    trace("Yippee. It's a number.");
  }
}
```

Но чтобы проверить, является ли значение действительным числом, нельзя просто сравнивать его с константой `NaN`.[1] Компилятор ActionScript 3.0 даже выдает предупреждение в этом случае. Для проведения проверки необходимо воспользоваться специальной функцией *isNaN( )*.

Для проверки числа на действительность числа предназначена функция *isNaN( )*:

```
var quantity:Number = 15 - "rabbits";
if (isNaN(quantity)) {
```

[1] Дело в том, что в соответствии с форматом представления чисел с плавающей точкой (IEEE-754) NaN – это диапазон чисел, а не одно число. Если арифметическое выражение вычислить невозможно, его результатом является одно из чисел диапазона NaN, и это число не всегда одно и то же. – *Примеч. науч. ред.*

```
    trace("Sorry, that is not a valid number.");
}
```

Противоположный вариант (т. е. невыполнение условия) проверяется оператором логического отрицания (!). Например, чтобы проверить, содержит ли переменная *действительное* число, используется *!isNAN( )*:

```
var quantity:Number = 15 - "rabbits";
if (!isNaN(quantity)) {
  // Число не недействительное, следовательно,
  // оно должно быть действительным
  trace ("That is a valid number.");
}
```

Конечно, сравнения можно проводить с помощью хорошо знакомых операторов сравнения. Например, чтобы сравнить два значения, можно воспользоваться операторами < и >.

```
trace(5 < 6);    // На экран выводится: true
trace(5 > 5);    // На экран выводится: false
```

Аналогично операторы <= и >= позволяют проверить, является ли одно значение меньшим/большим или равным другому значению:

```
trace(5 <= 6);   // На экран выводится: true
trace(5 >= 5);   // На экран выводится: true
```

Необходимо также помнить, что ActionScript сравнивает типы данных по-разному. Типы данных ActionScript можно разделить на *элементарные* (*string*, *number* и *Boolean*) и *составные* (*object*, *sprite* и *array*). ActionScript сравнивает элементарные типы по их значениям. В этом примере переменные `quantity` и `total` считаются равными, потому что обе содержат значение 6:

```
var quantity:Number = 6;
var total:Number = 6;
trace (quantity == total);   // На экран выводится: true
```

Однако составные типы ActionScript сравнивает «по ссылке». Это означает, что два элемента считаются равными, только если оба указывают на один и тот же объект, а не на объекты с одинаковым содержимым. Например, два массива, содержащих абсолютно одинаковые значения, не считаются равными:

```
// Создаем два массива с одинаковыми элементами.
var arrayOne:Array = new Array("a", "b", "c");
var arrayTwo:Array = new Array("a", "b", "c");
trace(arrayOne == arrayTwo);  // На экран выводится: false
```

Два составных элемента равны, только если они оба ссылаются на один и тот же объект, массив или спрайт. Например :

```
// Создаем один массив
var arrayOne:Array = new Array("a", "b", "c");
```

```
// Создаем другую переменную, которая ссылается на этот же массив.
var arrayTwo:Array = arrayOne;
trace(arrayOne == arrayTwo);  // На экран выводится: true
```

См. также

Рецепт 5.8.

1.9. Выполнение действий по условию

Задача

Действие выполняется только при соблюдении некоторого условия.

Решение

Используйте выражения *if* или *switch*.

Обсуждение

Часто в коде ActionScript необходимо принимать решения, например выполнять определенное действие или группу действий или нет. Для осуществления действий в зависимости от обстоятельств используется одно из *условных* выражений ActionScript: *if*, *switch* или тернарный условный оператор (`? :`).

Условные выражения обеспечивают возможность принимать логические решения. Какое их них лучше всего подходит в той или иной ситуации, вы поймете на практике. Например, выражение *if* лучше всего применять, когда Flash-ролик должен осуществлять какое-то действие только при выполнении определенного условия (например, когда условие возвращает `true`). Если надо проверять выполнение нескольких возможных условий, лучше обратиться к выражению *switch*. А тернарный условный оператор позволяет проверять выполнение условия и присваивание в одной строке.

Сначала рассмотрим выражение *if*. Из всех условных выражений ActionScript *if* – самое важное для понимания. Основная форма этого выражения образована ключевым словом `if`, за которым следует выражение, результат проверки которого определяет, какое действие будет выполнено. Проверяемое выражение заключается в круглые скобки, а действия, которые должны быть выполнены, – в фигурные (последнее обязательно, если в блоке больше одного действия).

Здесь выполняется проверка наличия слова «turtle» (черепаха) в `animalName` (имя животного). Так можно проверять ответы на вопросы викторины (в данном случае `animalName` – это переменная, в которой хранятся ответы пользователя). Обратите внимание, что эквивалентность двух элементов проверяется с помощью удвоенного знака равенства (`==`). Его нельзя путать с одинарным знаком равенства (`=`), который присваивает значение элементу.

```
if (animalName == "turtle") {
  // Это выражение trace() выполняется, только если
  // animalName равна "turtle".
  trace("Yay! 'Turtle' is the correct answer.");
}
```

Кроме того, в выражение *if* можно включить блок *else*, в котором содержатся альтернативные действия на случай невыполнения условия. Обратите внимание, выражение *trace( )* выполняется, только если файл *.swf* компилируется в режиме Debug, а не Run. Можно вызвать метод *showMessage( )*, который выводит на экран то или иное сообщение в зависимости от того, правильный или неправильный ответ был дан.

```
if (animalName == "turtle") {
  // Эти действия выполняются, только если
  // animalName равна "turtle".
  showMessage("Yay! 'Turtle' is the correct answer.");
}
else {
  // Эти действия выполняются, только если
  // animalName не равна "turtle".
  showMessage("Sorry, you got the question wrong.");
}
```

При тестировании можно создать метод *showMessage( )*, выводящий переданную в него строку. В реальной ситуации это сообщение можно отображать в текстовом поле или представлять пользователю как-то иначе, например в диалоговом окне.

Выражение *if* может включать блок *else...if*. Если условие блока *if* выполняется (`true`), блок *else...if* не обрабатывается. Если условие блока *if* не выполняется (`false`), интерпретатор ActionScript проверяет истинность условия в блоке *else...if*:

```
if (animalName == "turtle") {
  // Это выражение trace() выполняется,
  // только если animalName равна "turtle".
  showMessage ("Yay! 'Turtle' is the correct answer.");
}
else if (animalName == "dove") {
  // Это выражение trace() выполняется, только
  // если animalName не "turtle", а "dove".
  showMessage ("Sorry, a dove is a bird, not a reptile.");
}
```

А если предыдущий пример реализовать как два отдельных выражения *if* (одно для проверки `animalName` на равенство «turtle» (черепаха), а другое – «dove» (голубь))? Все получилось бы, но такой вариант менее эффективен. Применение выражения *else...if* гарантирует, что если `animalName` – это «turtle», то вторая проверка, на равенство «dove», не проводится.

Если два условия взаимоисключающие, то второе должно проверяться с помощью блока *else...if*. Если два условия не взаимоисключающие, и при их выполнении должны выполняться оба блока действий, то применяются отдельные выражения *if*.

Если в выражении *if* есть оба блока – и *else...if*, и *else*,– то блок *else* должен быть последним. Он удобен как ловушка: в него можно поместить действия, которые необходимо выполнить, если ни одно из условий не соблюдено.

```
if (animalName == "turtle") {
  // Это выражение trace() выполняется, только
  // если animalName равна "turtle".
  showMessage ("Yay! 'Turtle' is the correct answer.");
}
else if (animalName == "dove") {
   // Это выражение выполняется, только если
   // animalName не равна "turtle", а равна "dove".
  showMessage ("Sorry, a dove is a bird, not a reptile.");
}
else {
  // Это выражение выполняется, только если
  // animalName не равна ни "turtle", ни "dove".
  showMessage ("Sorry, try again.");
}
```

В выражение *if* можно включить несколько блоков *else...if*. Однако в этом случае лучше обратиться к выражению *switch*. Вообще выражения *switch* понятнее и лаконичнее, чем аналогичные выражения *if*. Когда важна производительность, некоторые разработчики на ActionScript предпочитают выражения *if*. Это обеспечивает несколько большие возможности для оптимизации.

Блок *switch* образован тремя частями:

Ключевое слово `switch`

Каждое выражение *switch* должно начинаться с ключевого слова `switch`.

Проверяемое выражение

Выражение, заключаемое в круглые скобки, значение которого необходимо проверить, чтобы определить, какое действие или действия выполнять.

Тело блока switch

Тело блока, заключенное в фигурные скобки, образуют *варианты* (*cases*). Каждый вариант состоит из следующих частей:

Ключевое слово `case` *или* `default`

Вариант должен начинаться с ключевого слова `case`. Исключение составляет вариант по умолчанию (аналог блока *else* опера-

тора *if*), для которого используется ключевое слово `default` (по умолчанию).

Выражение варианта

Выражение, значение которого сравнивается с проверяемым выражением блока *switch*. Если два значения равны, код тела варианта выполняется. Вариант по умолчанию (с ключевым словом `default`) не нуждается в выражении варианта.

Тело варианта

Одно или более выражений, обычно заканчивающихся оператором *break*, который выполняется, если этот *case* возвращает `true`.

За ключевым словом `switch` всегда следует проверяемое выражение в круглых скобках. Затем тело блока *switch*, заключенное в фигурные скобки. В теле *switch* может быть один или более операторов *case*. Каждый вариант (кроме варианта по умолчанию) начинается с ключевого слова `case`, за которым следует выражение варианта и двоеточие. Вариант по умолчанию (если таковой присутствует) начинается с ключевого слова `default`, за которым следует двоеточие. Таким образом, общий вид блока *switch* следующий:

```
switch (проверяемоеВыражение) {
  case выражениеВарианта:
    // тело варианта
  case выражениеВарианта:
    // тело варианта
  default:
    // тело варианта
}
```

Обратите внимание, что если выражение `case` возвращает значение `true`, то все действия во всех последующих вариантах тела оператора *switch* тоже выполняются. Скорее всего в данном примере это не совсем то, чего добивался разработчик.

Вот пример:

```
var animalName:String = "dove";
/* В следующем блоке switch первое trace()
   не выполняется, потому что animalName не равна
   "turtle". Но выполняются и второе и третье выражения
   trace(), поскольку вариант "dove" дает значение true.
*/
switch (animalName) {
  case "turtle":
    trace("Yay! 'Turtle' is the correct answer.");
  case "dove":
    trace("Sorry, a dove is a bird, not a reptile.");
  default:
    trace("Sorry, try again.");
}
```

Обычно в конце тела каждого варианта располагаются операторы *break*, обеспечивающие выход из *switch* после выполнения операций совпавшего варианта.

Оператор *break* прерывает выполнение текущего блока *switch*, предотвращая ненужное выполнение тел последующих вариантов.

Оператор *break* не нужен в конце последнего *case* или *default*, поскольку все равно дальше нет никаких вариантов, блок *switch* заканчивается.

```
var animalName:String = "dove";
// Здесь выполняется только второе выражение trace().
switch (animalName) {
  case "turtle":
    trace("Yay! 'Turtle' is the correct answer.");
    break;
  case "dove":
    trace("Sorry, a dove is a bird, not a reptile.");
    break;
  default:
    trace("Sorry, try again.");
}
```

Выражение *switch* особенно полезно, если необходимо выполнить одно и то же действие для одного из ряда подобных вариантов. В этом случае выражения вариантов просто размещаются друг за другом. Например:

```
switch (animalName) {
  case "turtle":
  case "alligator":
  case "iguana":
    trace("Yay! You named a reptile.");
    break;
  case "dove":
  case "pigeon":
  case "cardinal":
    trace("Sorry, you specified a bird, not a reptile.");
    break;
  default:
    trace("Sorry, try again.");
}
```

ActionScript также поддерживает тернарный условный оператор (`? :`), который позволяет проводить проверку и присваивание в одной строке. *Тернарному оператору* необходимо три операнда, в отличие от одного-двух в унарных или бинарных операторах. Первый операнд условного оператора – это условное выражение, которое может выполняться (`true`) или не выполняться (`false`). Второй операнд представляет собой значение, присваиваемое переменной, если условие выполняется

(true), и третий операнд – значение, присваиваемое, если условие не выполняется (false).

```
имяПеременной = (условное выражение) ? значениеЕслиTrue : значениеЕслиFalse;
```

1.10. Проверка сложных условий

Задача

Необходимо принять решение на основании нескольких условий.

Решение

Для создания составных условных выражений используются логические операторы *И* (&&), *ИЛИ* (||) и *НЕ* (!).

Обсуждение

Конструкции с условными операторами могут входить в состав многих выражений ActionScript, включая if, while и for. Истинность двух условий проверяется посредством логического оператора *И*, && (в главе 14 подробно рассматривается работа с датами):

```
// Проверяем, не 17-е ли апреля сегодня.
var current:Date = new Date();
if (current.getDate() == 17 && current.getMonth() == 3) {
  trace ("Happy Birthday, Bruce!");
}
```

Чтобы сделать логику более прозрачной, можно добавить дополнительные круглые скобки:

```
// Проверяем, не 17-е ли апреля сегодня.
if ((current.getDate() == 17) && (current.getMonth() == 3)) {
  trace ("Happy Birthday, Bruce!");
}
```

Здесь с помощью логического оператора *ИЛИ*, ||, проверяется истинность любого из условий:

```
// Проверяем, не выходной ли это.
if ((current.getDay() == 0) || (current.getDay() == 6) ) {
  trace ("Why are you working on a weekend?");
}
```

Также с помощью логического оператора *НЕ*, !, можно проверять ложность условий:

```
// Проводим проверку на то, что имя пользователя не Bruce.
if (!(userName == "Bruce")) {
  trace ("This application knows only Bruce's birthday.");
}
```

Предыдущий пример можно изменить, используя оператор неравенства `!=`:

```
if (userName != "Bruce") {
  trace ("This application knows only Bruce's birthday.");
}
```

Любое Булево значение или выражение, которое приводится к таковому, может выступать в качестве условия:

```
// Проверяем, является ли спрайт видимым.
// Если да, выводим на экран сообщение. Это условие является
// сокращенной записью _sprite.visible == true
if (_sprite.visible) {
  trace("The sprite is visible.");
}
```

Логический оператор *НЕ* позволяет проверить ложность (`false`), а не истинность (`true`):

```
// Проверяем, является ли спрайт невидимым. Если да, выводим на экран
// сообщение. Это условие является сокращенной записью
// _sprite.visible != true или _sprite.visible == false.
if (!_sprite.visible) {
  trace("The sprite is invisible. Set it to visible before trying this
action.");
}
```

Логический оператор *НЕ* часто применяется в объединенных условиях вместе с логическим оператором *ИЛИ*:

```
// Проводим проверку на то, что имя пользователя ни Bruce, ни Joey.
// (Это также можно было бы переписать с использованием двух
// операторов неравенства и логического И.)
if (!((userName == "Bruce") || (userName == "Joey"))) {
  trace ("Sorry, but only Bruce and Joey have access to this application.");
}
```

ActionScript не берется за вычисление второй половины логического *И*, если первая половина не возвращает значение `true`. Если первая половина `false`, значит, все выражение всегда дает в результате `false`, поэтому нет смысла заниматься второй половиной (кроме того, это негативно отразилось бы на производительности). Аналогично ActionScript не рассматривает вторую половину логического *ИЛИ*, если первая половина не `false`. Если первая половина `true`, все выражение всегда будет `true`.

1.11. Многократное повторение операции

Задача

Необходимо многократно выполнить некоторую операцию в одном кадре.

Решение

Для многократного повторения одной и той же операции в одном кадре применяется оператор цикла. Например, оператор *for*:

```
for (var i:int = 0; i < 10; i++) {
  // Вывести на экран значение i.
  trace(i);
}
```

Обсуждение

Если в одном кадре необходимо многократно выполнить одно и то же действие (или его вариации), используется оператор цикла. Это делает код лаконичным, более простым для восприятия и обновления. Для этого подходят и выражение *while* и выражение *for*, но, как правило, лучше остановить свой выбор на *for*. Оба выражения обеспечивают одинаковый результат, но блок *for* компактнее и привычнее большинству программистов.

Синтаксис блока *for* включает пять основных частей:

Ключевое слово for

Каждый блок *for* должен начинаться с ключевого слова `for`.

Инициализирующее выражение

В цикле обычно задействуется *индексная переменная* (*счетчик циклов*), которая инициализируется при первом выполнении блока. Инициализация осуществляется всего один раз, независимо от количества повторений цикла.

Проверяемое выражение

Цикл должен включать проверяемое выражение, возвращающее `true` или `false`. Проверяемое выражение вычисляется при каждом проходе цикла один раз. Как правило, оно сравнивает индексную переменную с некоторым значением, например максимальным числом итераций цикла. Чтобы тело цикла выполнилось, проверяемое выражение должно возвращать `true` (в противоположность циклу *do...while*, который выполняется по крайней мере один раз, даже если проверяемое выражение возвращает `false`). С другой стороны, если проверяемое выражение никогда не становится `false`, образуется бесконечный цикл. В результате выдается предупреждение о низкой производительности Flash Player.

Обновляющее выражение

Обновляющее выражение обычно обновляет значение переменной, фигурирующей в проверяемом выражении, так чтобы в некоторый момент условие возвратило значение `false`, и цикл завершился. Обновляющее выражение вычисляется при каждом проходе цикла один раз. Часто причиной возникновения бесконечного цикла яв-

ляется сбой при обновлении соответствующей переменной в обновляющем выражении (как правило, та же переменная фигурирует в проверяемом выражении).

Тело цикла

Тело цикла – это блок выражений, заключенных в фигурные скобки, которые выполняются при каждом проходе цикла. Если проверяемое выражение не возвращает значение `true`, тело блока *for* не выполняется.

Первым должно стоять ключевое слово `for`, за которым следуют инициализирующее, проверяемое и обновляющее выражения, заключенные в круглые скобки. Все три выражения разделяются точкой с запятой (инициализирующее, проверяемое и обновляющее выражения могут остутствовать, но точки с запятой обязательны). Далее следует тело оператора, заключенное в фигурные скобки:

```
for (инициализация; проверка; обновление) {
  тело оператора
}
```

Вот пример цикла *for*, который выводит числа от 0 до 999:

```
for (var i:int = 0; i < 1000; i++) {
  trace(i);
}
trace ("That's the end.");
```

Чтобы понять цикл *for*, можно поэтапно проследить за процессом обработки команд интерпретатором ActionScript. В предыдущем примере ключевое слово `for` указывает интерпретатору выполнять действия цикла *for* до тех пор, пока условное выражение истинно (`true`). Вот как это происходит:

1. Инициализирующее выражение выполняется только раз, в нем переменной `i` присваивается значение `0`.
2. Далее интерпретатор выполняет проверяемое выражение (*i < 1000*). Поскольку `i` равна `0`, т. е. меньше 1000, выражение возвращает `true`, и выполняется действие *trace( )* тела блока *for*.
3. Затем интерпретатор ActionScript выполняет обновляющее выражение, в данном случае `i++`, которое увеличивает `i` на `1`.
4. Интерпретатор повторяет цикл сначала (но пропускает этап инициализации). Таким образом, он опять проверяет истинность проверяемого выражения и, если возвращается `true`, выполняет тело цикла. И опять выполняется обновляющее выражение.

Этот процесс повторяется до тех пор, пока проверяемое выражение истинно. Последним в окне Output выводится значение 999, потому что как только `i` станет равной 1000, проверяемое выражение перестанет быть истинным, и цикл завершится. По завершении цикла выполнение продолжается с команды, следующей сразу за циклом.

И инициализирующее, и обновляющее выражения могут включать несколько действий, записанных через запятую. Но ключевое слово *var* в инициализирующем выражении встречается только один раз. В следующем примере одновременно происходит приращение на единицу переменной `i`, уменьшение на единицу переменной `j` и вывод их значений в окне Output:

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

Предыдущий пример не аналогичен двум вложенным операторам *for* (что можно увидеть в следующем блоке кода).

Также часто применяются вложенные циклы *for*. Во вложенном цикле *for* должна использоваться другая индексная переменная, чем во внешнем цикле *for*. В самом внешнем цикле *for* принято оперировать переменной `i`, а во вложенном – переменной `j`:

```
for (var i:int = 1; i <= 3; i++) {
  for (var j:int = 1; j <= 2; j++) {
    trace(i + " X " + j + " = " + (i * j));
  }
}
```

В результате выполнения предыдущего примера в окне Output выводится следующая таблица умножения:

```
1 X 1 = 1
1 X 2 = 2
2 X 1 = 2
2 X 2 = 4
3 X 1 = 3
3 X 2 = 6
```

Возможно несколько уровней вложенности операторов *for*. Для каждого следующего уровня в качестве индексной переменной принято назначать следующий по алфавиту символ. Таким образом, в операторе *for* третьего уровня вложенности обычно фигурирует индексная переменная `k`:

```
for (var i:int = 1; i <= 3; i++) {
  for (var j:int = 1; j <= 3; j++) {
    for (var k:int = 1; k <= 3; k++) {
      trace(i + " X " + j + " X " + k + " = " + (i * j * k));
    }
  }
}
```

Кроме того, в операторах *for* индексная переменная может получать отрицательное приращение или обновляться иначе, чем просто увеличиваться или уменьшаться на единицу:

```
// Обратный счет от 10 до 1.
for (var i:int = 10; i > 0; i--) {
  trace(i);
}
// Выводится последовательность квадратных корней.
for (var i:Number = 50000; i > 2; i = Math.sqrt(i)) {
  trace(i);
}
```

В этом случае в итоге в переменной `i` сохраняются значения, не являющиеся целыми, поэтому лучше объявить ее типом *Number*, а не *int*.

Оператор *for* нельзя рекомендовать для задач, выполнение которых привязано ко времени.

Многие программисты ошибочно используют операторы *for* для анимирования спрайтов. Например:

```
for (var i:int = 0; i < 20; i++) {
  _sprite.x += 10;
}
```

Этот код перемещает спрайт на 200 пикселов вправо от его исходного положения, и все обновления происходят в одном кадре. В этом случае присутствуют две проблемы. Во-первых, Сцена (Stage) в одном кадре обновляется только раз, поэтому отображается лишь последнее обновление (спрайт вдруг «перепрыгнет» на 200 пикселов, а не будет перемещаться постепенно, за 20 шагов). Во-вторых, даже если бы Сцена обновлялась чаще, каждая итерация цикла *for* занимает несколько миллисекунд, движение было бы слишком стремительным. Для действий, которые должны длиться некоторый промежуток времени, необходимо использовать событие *enterFrame* (см. рецепт 1.5) или таймер (см. рецепт 1.12).

Более того, быстро повторяющиеся циклы не годятся для представления длительных процессов (всего, что длится более доли секунды). Flash Player выводит на экран предупреждение, если один цикл выполняется более 15 секунд. Применение других методов (только что упомянутых) позволяет избежать предупреждений и обеспечивает Flash возможность осуществлять другие действия, кроме повторяющихся, которые являются частью цикла.

См. также

Рецепты 1.5 и 1.12. Оператор *for* применяется во многих реальных ситуациях, поэтому его примеры можно увидеть во многих рецептах данной книги. В рецептах 5.2 и 12.8 представлены некоторые практические примеры. В рецепте 5.16 обсуждаются циклы *for...in*, используемые для перечисления свойств объекта или массива.

1.12. Повторение задачи в течение некоторого промежутка времени

Задача

Некоторое действие (или действия) должно длиться некоторое время.

Решение

Используйте класс *Timer* или слушайте событие *enterFrame* спрайта.

Обсуждение

Класс *Timer* появился в ActionScript 3.0, и он призван заменить использовавшиеся ранее функции *setInterval( )* и *setTimeout( )*. Будучи созданным, экземпляр класса *Timer* вырабатывает события *timer* через равные промежутки времени. Интервал между событиями и их количество можно задать в конструкторе *Timer*:

```
var timer:Timer = new Timer(delay, repeatCount);
```

Настройка метода, обрабатывающего эти события, осуществляется с помощью метода *addEventListener*. После того как таймер создан и настроен слушатель его событий, запуск и остановка таймера осуществляются с помощью методов *start( )*и *stop( )*.

Класс *Timer* представляет собой часть пакета *flash.utils*; также есть класс *TimerEvent* в пакете *flash.events*. Их необходимо импортировать:

```
package {
    import flash.display.Sprite;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class ExampleApplication extends Sprite {
        // Объявляем и инициализируем переменную
        // для хранения предыдущего значения таймера.
        private var _PreviousTime:Number = 0;

        public function ExampleApplication() {
            var tTimer:Timer = new Timer(500, 10);
            tTimer.addEventListener(TimerEvent.TIMER, onTimer);
            tTimer.start();
        }

        private function onTimer(event:TimerEvent):void {
            // Выводим разницу между текущим значением таймера
            // и его значением на момент последнего вызова функции.
            trace(flash.utils.getTimer() - _PreviousTime);
            _PreviousTime = flash.utils.getTimer();
        }
    }
}
```

Функция *getTimer( )* (ранее функция верхнего уровня) также была перенесена в пакет *flash.utils*. Она просто возвращает количество миллисекунд, прошедших с момента запуска приложения.

В предыдущем примере интервал теоретически составляет 500 миллисекунд, но в реальности время зависит от производительности компьютера, который может выполнять и другие задачи, занимающие ресурсы процессора. Отсюда следует:

- Не стоит полагаться на таймеры, если требуется очень высокая точность.
- Не стоит полагаться на интервалы таймера, длительность которых меньше 10 миллисекунд.

Чтобы сымитировать работу функции *setInterval( )*, счетчик повторений устанавливается на нуль, тогда события таймера будут формироваться бесконечно. В этом случае метод *stop( )* аналогичен функции *clearInterval( )* и прекращает формирование событий таймером.

Аналогично, если необходимо создать дубликат функции *setTimeout( )*, счетчику повторений присваивается значение 1. Таймер ожидает заданное количество времени, формирует одно событие и останавливается.

Класс *Timer* позволяет создавать превосходные анимации, не зависящие от частоты смены кадров ролика, и осуществлять вызовы метода с любым интервалом. Вот пример, в котором заданы два таймера. Один для квадратного спрайта (с частотой 50 миллисекунд), и другой для круглого спрайта (с частотой 100 миллисекунд):

```
package {
    import flash.display.Sprite;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class ExampleApplication extends Sprite {
        private var _square:Sprite;
        private var _circle:Sprite;

        public function ExampleApplication() {
            // Создаем и отрисовываем два спрайта
            _square = new Sprite();
            _square.graphics.beginFill(0xff0000);
            _square.graphics.drawRect(0, 0, 100, 100);
            _square.graphics.endFill();
            addChild(_square);
            _square.x = 100;
            _square.y = 50;

            _circle = new Sprite();
            _circle.graphics.beginFill(0x0000ff);
            _circle.graphics.drawCircle(50, 50, 50);
            _circle.graphics.endFill();
            addChild(_circle);
```

```
            _circle.x = 100;
            _circle.y = 200;

            // Создаем и запускаем два таймера
            var squareTimer:Timer = new Timer(50, 0);
            squareTimer.addEventListener(TimerEvent.TIMER, onSquareTimer);
            squareTimer.start();

            var circleTimer:Timer = new Timer(100, 0);
            circleTimer.addEventListener(TimerEvent.TIMER, onCircleTimer);
            circleTimer.start();
        }

        // Определяем два метода-обработчика
        private function onSquareTimer(event:TimerEvent):void {
            _square.x++;
        }

        private function onCircleTimer(event:TimerEvent):void {
            _circle.x++;
        }
    }
}
```

Для выполнения повторяющегося в течение некоторого времени действия (или действий) можно использовать событие *enterFrame* спрайта. Но технология *Timer* предлагает некоторые преимущества по сравнению с событием *enterFrame*. Важнее всего, что появляется возможность создавать временные интервалы, отличающиеся от частоты смены кадров *.swf*, а в случае применения *enterFrame* обработчик вызывается с частотой смены кадров.

О чем это говорит? О том, что по-прежнему существуют ситуации, когда лучше применять *enterFrame*. Например, что-то должно происходить с частотой смены кадров *.swf*. Один из таких сценариев – воспроизведение кадров клипа в обратном порядке.

1.13. Создание многократно используемого кода

Задача

Время от времени требуется выполнять ряд одних и тех же действий, но избегать при этом ненужного дублирования кода.

Решение

Создать метод и затем вызывать (т. е. инициировать) его всегда, когда требуется выполнить эти действия. Если функция является членом класса, ее называют *методом*.

Вот как создается *метод* класса:

```
модификаторДоступа функция имяФункции ():ВозвращаемыйТипДанных {
```

```
    // Здесь располагаются действия.
}
```

Чтобы *вызвать* (т. е. выполнить) именованный метод, ссылаемся на него по имени вот так:

```
имяФункции();
```

Обсуждение

Группировка выражений в метод позволяет, определив последовательность действий всего один раз, выполнять ее сколько угодно раз. Это полезно, когда время от времени требуется осуществлять одинаковые действия и хочется избежать многократного дублирования одного и того же кода. Централизованное размещение кода в методах делает его более понятным (потому что метод пишется один раз и тот, кто встраивает его в свой код, может не задумываться о деталях его реализации) и упрощает его обслуживание (потому что изменения вносятся только в одном месте, а не во многих).

Как и переменные класса, методы могут объявляться с *модификаторами доступа*, которые определяют, какие классы могут вызывать методы. Здесь используются такие модификаторы доступа:

`private`

Доступ может осуществляться только внутри самого класса.

`protected`

Метод доступен классу или любому подклассу. Все основывается на экземплярах, т. е. экземпляр класса имеет доступ к собственным защищенным членам или защищенным членам своего надкласса, но не может обращаться к защищенным членам других экземпляров этого же класса.

`internal`

Метод доступен любому классу пакета.

`public`

Метод доступен любому классу.

В ActionScript 2.0 модификатор `private` делал метод доступным и подклассам. Если модификатор доступа не задается явно, методом по умолчанию будет `internal`.

Вот пример отрисовки десяти линий. Код создания линии (три строки) не повторяется для каждой отдельной линии, а заключен в метод `drawLine`, который вызывается 10 раз:

```
package {
    import flash.display.Sprite;
    public class ExampleApplication extends Sprite
    {
        public function ExampleApplication() {
```

```
            for(var i:int=0;i<10;i++) {
                drawLine();
            }
        }

        private function drawLine():void {
            graphics.lineStyle(1, Math.random() * 0xffffff, 1);
            graphics.moveTo(Math.random() * 400, Math.random() * 400);
            graphics.lineTo(Math.random() * 400, Math.random() * 400);
        }
    }
}
```

Другой важный тип методов – *статические* (*static*) методы. Они недоступны как члены экземпляра класса, но вызываются непосредственно из самого класса. Например, в классе `ExampleApplication` можно определить статический метод:

```
public static function showMessage():void {
    trace("Hello world");
}
```

Затем этот метод можно вызывать:

```
ExampleApplication.showMessage();
```

В некоторых классах нет ничего кроме статических методов, например класс *Math*. Заметьте, что для использования его методов не надо создавать новый объект *Math*; методы вызываются просто как свойства класса (*Math.random( )*, *Math.round( )* и т. д.).

1.14. Обобщение метода

Задача

Требуется повысить возможность выполнения многократного использования кода и при этом избежать дублирования множества строк кода для внесения в них лишь небольших изменений.

Решение

Чтобы сделать метод достаточно гибким для осуществления лишь немного отличающихся действий, введите в метод параметры.

Определите параметры, обеспечивающие необходимую изменчивость метода:

```
private function average (a:Number, b:Number, c:Number):void {
  trace("The average is " + (a + b + c)/3);
}
```

Если точное число параметров, передаваемых в метод, не известно, используется встроенный массив `arguments`, обеспечивающий возможность приема переменного количества параметров.

Обсуждение

Результат выполнения метода, не принимающего параметров, всегда один и тот же. Однако часто требуется осуществлять практически такие же действия, какие обеспечивает существующий метод, но с небольшими отличиями. Дублирование всего метода и внесенние в него изменений в большинстве случаев не годится. Обычно это приводит к усложнению обслуживания и понимания кода. Что еще важнее, обычно таких вариантов метода требуется не два, а намного больше. Обслуживание пяти-шести разновидностей того, что в идеале могло бы быть заключено в один метод, часто приводит к ненужным трудностям. Фокус в том, чтобы создать один метод, который может принимать разные значения и функционировать в зависимости от них.

Например, у нас есть метод *average( )* для вычисления среднего значения набора чисел. Но требуется находить среднее арифметическое не одних и тех же двух чисел, а произвольно задаваемых значений. Это можно реализовать с помощью параметров.

Самый обычный способ задания параметров – перечислить их в скобках в описании метода. Имена параметров указываются через запятую. При вызове метода в него передается список перечисленных через запятую аргументов, соответствующих определенным параметрам.

Переменные, определенные в описании метода, или значения, передаваемые в метод при его инициации, называют «параметрами» или «аргументами».

Вот простой пример описания метода с параметрами:

```
// Определяем функцию, ожидающую два параметра: a и b.
private function average(a:Number, b:Number):Number {
  return (a + b)/2;
}
```

А вот вызов метода с передачей аргументов:

```
// При вызове функции в нее передается два аргумента,
// например, 5 и 11, соответствующие параметрам a и b.
var averageValue:Number = average(5, 11);
```

В большинстве ситуаций лучше объявить параметры, которые ожидает метод. Однако иногда количество параметров неизвестно. Например, если требуется, чтобы метод *average( )* вычислял среднее арифметическое любого количества значений, можно использовать встроенный массив `arguments`, присутствующий в теле любой функции. Все параметры, передаваемые в функцию, автоматически помещаются в ее массив `arguments`.

```
// При использовании массива arguments описывать
// принимаемые параметры не нужно.
private function average():Number {
```

```
  var sum:Number = 0;
  // Перебираем все элементы массива arguments
  // и добавляем эти значения в sum.
  for (var i:int = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  // Затем делим на общее число аргументов
  return sum/arguments.length;
}
// Метод average() можно вызвать с любым числом параметров.
var average:Number = average (1, 2, 5, 10, 8, 20);
```

Формально `arguments` – это объект, обладающий дополнительными свойствами по сравнению с обычным массивом. Однако, хотя `arguments` представляет собой массив особого рода, с ним можно работать, как с обычным массивом.

1.15. Выход из метода

Задача

Необходимо покинуть метод до завершения его выполнения.

Решение

Методы завершаются автоматически после выполнения последнего оператора. Оператор *return* (возврат) предназначен для преждевременного выхода из метода.

Обсуждение

Оператор *return* осуществляет выход из метода, и интерпретатор ActionScript продолжает выполнение кода, изначально вызвавшего метод. Все операторы тела метода после оператора *return* игнорируются.

```
private function sampleFunction ():void {
  return;
  trace("Never called");
}
// Вызов из другого метода:
sampleFunction();
// После возвращения из метода sampleFuction() выполнение продолжается отсюда
```

В предыдущем примере оператор *return* прерывает выполнение метода до осуществления каких-либо действий, поэтому это не очень полезный метод. Чаще всего оператор *return* применяется для выхода из метода в случае невыполнения определенных условий. Вот пример завершения метода в случае неправильного введения пароля:

```
private function checkPassword (password:String):void {
  // Если password не "SimonSays", выйти из функции.
```

```
    if (password != "SimonSays") {
      return;
    }
    // В противном случае выполнить остальные действия.
    showForm ("TreasureMap");
}
// В этом вызове метода указан неверный пароль,
// поэтому выполнение функции прерывается.
checkPassword("MotherMayI");
// В этом вызове метода указан верный пароль,
// поэтому функция показывает форму TreasureMap.
checkPassword("SimonSays");
```

В предыдущем примере метод объявлен `void`, но оператор *return* может использоваться в методе, и это не приведет к ошибке компиляции. Если оператор *return* предназначен лишь для выхода из метода, он может использоваться в методе, объявленном как `void`.

В ActionScript 2.0 функции объявлялись с ключевым словом `Void`. В ActionScript 3.0 записывается с маленькой буквы: `void`.

Однако если попытаться возвратить значение из такого метода, компилятор сформирует ошибку.

```
private function sampleMethod ():void {
  return "some value";  // Компилятор формирует ошибку.
}
```

1.16. Получение результата метода

Задача

Необходимо выполнить некоторый метод и возвратить результаты в выражение, вызвавшее функцию.

Решение

Это делается при помощи оператора *return*, определяющего возвращаемое значение.

Обсуждение

Оператор *return* без параметров просто прерывает выполнение метода. Однако любое значение, заданное после ключевого слова `return`, возвращается в выражение, вызвавшее метод. Обычно возвращенное значение сохраняется в переменной, а его тип соответствует возвращаемому типу метода.

```
private function average (a:Number, b:Number):Number {
  return (a + b)/2;
}
```

Теперь можно вызвать метод *average( )* и сохранить результат в переменной.

```
var playerScore:Number = average(6, 10);
trace("The player's average score is " + playerScore);
```

Возвращаемое значение метода можно и не сохранять в переменной, а передавать как параметр в другую функцию, например:

```
trace("The player's average score is " + average(6, 10));
```

Однако обратите внимание, что если с возвращенным значением функции ничего не делать, оно будет просто утеряно. Например, данное выражение абсолютно бесполезно, потому что результат его выполнения нигде не отображается и никак не используется:

```
average(6, 10);
```

1.17. Обработка ошибок

Задача

Необходимо программно выявлять возникновение определенных ошибок и обрабатывать их.

Решение

Для формирования ошибок при их выявлении применяется выражение *throw*. Любой потенциально ошибочный код должен размещаться в блоке *try*, которому соответствует один или более блоков *catch* для обработки возможных ошибок.

Обсуждение

Flash Player 8.5 поддерживает методику *try/catch* для обработки ошибок ActionScript. Это означает, что можно создавать код, разумно обрабатывающий определенные типы ошибок в случае их появления. От синтаксических ошибок это не спасает (*.swf* не будет даже компилироваться в этом случае), но большинство других типов ошибок, таких как пропуск или применение недопустимых данных, вполне можно обрабатывать. А самое главное, можно попытаться решить все вопросы программно.

Разобраться с тем, когда и как следует применять методику *try/catch*, поможет пример. Рассмотрим приложение, отрисовывающее прямоугольник на основании вводимых пользователем размеров. Для создания изображения прямоугольника должен быть определен диапазон допустимых размеров. К примеру, заданные значения должны быть действительными числами больше 1 и меньше 200. Есть разные способы обеспечения достоверности данных еще даже до попытки отрисовки, но в дополнение к ним можно обратиться к методике *try/catch* для

повышения надежности. В этом случае, если значения размеров недопустимы или выходят за рамки заданного диапазона, можно будет сформировать ошибку, обрабатываемую программно. Здесь возможны разные варианты, начиная от простого невыполнения действия до подстановки данных по умолчанию и предложения пользователю ввести допустимые данные.

Работа с ошибками в ActionScript состоит из двух основных частей: формирования ошибки и ее перехвата. Есть несколько ошибок, формируемых автоматически проигрывателем, например *IllegalOperationError*, *MemoryError* и *ScriptTimeoutError*. Они располагаются в пакете *flash.errors*. Но также можно выявлять возникновение ошибки и формировать собственную. Ошибки формируются выражением *throw*, начинающимся с ключевого слова *throw*, за которым следует значение или ссылка, которые должны быть сформированы. Чаще всего приходится формировать объект *Error* (Ошибка) или экземпляр подкласса *Error*. Например:

```
throw new Error("A general error occurred.");
```

Как видите, конструктор *Error* принимает один параметр: сообщение, ассоциируемое с ошибкой. Параметр необязательный, и в зависимости от того, как обрабатывается ошибка, может присутствовать или отсутствовать. Однако в большинстве случаев имеет смысл определить сообщение об ошибке. Тогда их можно будет регистрировать в целях отладки.

Как только ошибка сформирована, Flash прерывает текущий процесс и ищет блок *catch* для ее обработки. Вот здесь в игру вступают блоки *try* и *catch*. Любой потенциально ошибочный код должен быть включен в блок *try*. Тогда, если формируется ошибка, прерывается только блок *try* и вызывается ассоциированный с ним блок *catch*. Вот самый простой сценарий:

```
try {
  trace("This code is about to throw an error.");
  throw new Error("A general error occurred.");
  trace("This line won't run");
}
catch (errObject:Error) {
  trace("The catch block has been called.");
  trace("The message is: " + errObject.message);
}
```

В результате выполнения этого кода в панели Output выводится следующее:

```
This code is about to throw an error.
The catch block has been called.
The message is: A general error occurred.
(Этот код формирует ошибку. Вызван блок catch.
Сообщение: Возникла общая ошибка.)
```

Конечно, приведенный пример очень упрощенный, и в реальных приложениях такой код вряд ли встретится, но он иллюстрирует основной процесс. Можно увидеть, что когда сформирована ошибка, выполнение блока *try* прерывается и выполняется блок *catch*, передающий ссылку на объект *Error*, который был сформирован.

Чаще всего ошибки формируются в функции или методе. Тогда Flash ищет в блоке *try* функции оператор *throw*. Если находит, то вызывает ассоциированный блок *catch*, как было показано выше. Однако если оператор *throw* в функции не включен в блок *try*, Flash прерывает выполнение функции и смотрит, не была ли она вызвана из блока *try*. Если это так, он прерывает выполнение блока *try* и выполняет ассоциированный с ним блок *catch*. Еще один, очень простой пример:

```
private function displayMessage(message:String):void {
  if(message == undefined) {
    throw new Error("No message was defined.");
  }
  trace(message);
}
try {
  trace("This code is about to throw an error.");
  displayMessage();
  trace("This line won't run");
}
catch (errObject:Error) {
  trace("The catch block has been called.");
  trace("The message is: " + errObject.message);
}
```

В результате выполнения предыдущего примера в панели Output выводится следующее:

```
This code is about to throw an error.
The catch block has been called.
The message is: No message was defined.
(Этот код формирует ошибку. Вызван блок catch.
Сообщение: Сообщение не определено.)
```

Как видите, код работает практически так же, как и в предыдущем примере, только оператор *throw* спрятан в функции, а не вызывается непосредственно в блоке *try*. Преимущество в том, что теперь можно создавать интеллектуальные функции и методы, которые знают, почему и когда формировать ошибки. А эти функции и методы можно просто заключать в блоки *try* и обрабатывать любые ошибки в случае их возникновения.

Следующий код иллюстрирует более реалистичный пример:

```
// Определяем функцию, которая отрисовывает прямоугольник в заданном спрайте
private function drawRectangle(sprite:Sprite, newWidth:Number,
newHeight:Number):void {
```

```
  // Проверяем заданные размеры. Если это не числа, формируем ошибку.
  if(isNaN(newWidth) || isNaN(newHeight)) {
    throw new Error("Invalid dimensions specified.");
  }
  // Если ошибка не была сформирована, отрисовываем прямоугольник.
  sprite.graphics.lineStyle(1, 0, 1);
  sprite.graphics.lineTo(nWidth, 0);
  sprite.graphics.lineTo(nWidth, nHeight);
  sprite.graphics.lineTo(0, nHeight);
  sprite.graphics.lineTo(0, 0);
}
```

Теперь можно вызвать метод *drawRectangle( )* посредством выражения *try/catch*.

```
try {
  // Делаем попытку отрисовать два прямоугольника в текущем спрайте.
  // В этом примере предполагается, что переменные для задания размеров
  // получают значения из ввода пользователя, базы данных,
  // XML-файла или другого источника данных.
  drawRectangle(this, widthA, heightA);
  drawRectangle(this, widthB, heightB);
}
catch(errObject:Error) {
  // Если возникает ошибка, удаляем из спрайта все отрисованные
  // прямоугольники. Затем выводим сообщение для пользователя.
  this.graphics.clear();
  tOutput.text = "An error occurred: " + errObject.message;
}
```

Кроме блоков *try* и *catch* можно определить блок *finally*. Блок *finally* содержит код, вызываемый независимо от того, была ли сформирована ошибка. Во многих случаях в блоке *finally* нет необходимости. Например, два следующих примера делают одно и то же:

```
// Без использования блока finally:
private function displayMessage(message:String):void {
  try {
    if(message == undefined) {
      throw new Error("The message is undefined.");
    }
    trace(message);
  }
  catch (errObject:Error) {
    trace(errObject.message);
  }
  trace("This is the last line displayed.");
}
// С блоком finally:
private function displayMessage(message:String):void {
  try {
    if(message == undefined) {
```

```
        throw new Error("The message is undefined.");
      }
      trace(message);
    }
    catch (errObject:Error) {
      trace(errObject.message);
    }
    finally {
      trace("This is the last line displayed.");
    }
  }
```

Однако блок *finally* выполняется независимо от того, что происходит в блоках *try* и *catch*, включая оператор *return*. Таким образом, следующие две функции не эквивалентны:

```
// Без блока finally:
private function displayMessage(message:String):void {
  try {
    if(message == undefined) {
      throw new Error("The message is undefined.");
    }
    trace(message);
  }
  catch (errObject:Error) {
    trace(errObject.message);
    return;
  }
  // Эта строка не выполняется, если ошибка перехвачена.
  trace("This is the last line displayed.");
}
// С блоком finally:
private function displayMessage(message:String):void {
  try {
    if(message == undefined) {
      throw new Error("The message is undefined.");
    }
    trace(message);
  }
  catch (errObject:Error) {
    trace(errObject.message);
    return;
  }
  finally {
    // Эта строка выполняется, даже если ошибка перехвачена.
    trace("This is the last line displayed.");
  }
}
```

Можно создавать намного более сложные системы обработки ошибок, чем показаны в этом рецепте. В данной книге можно найти такие обработчики в соответствующих контекстах.

2

Специальные классы

2.0. Введение

Классы – неотъемлемая часть ActionScript 3.0. Это справедливо для ActionScript 3.0 как ни для одной другой, более ранней версии этого языка. ActionScript 1.0 был преимущественно процедурным языком со скромными объектно-ориентированными возможностями. ActionScript 2.0 формализовал объектно-ориентированные черты и сделал большой шаг по направлению к истинно объектно-ориентированному языку. Однако именно ActionScript 3.0 меняет основной курс ActionScript и делает классы основными строительными блоками языка. Если ActionScript 3.0 используется с Flex, то весь код ActionScript должен находиться в классе; исключение составляет код, размещаемый в тегах `<mx:Script>`. В данной главе обсуждаются основы написания специальных классов в ActionScript 3.0.

2.1. Создание специального класса

Задача

Требуется написать специальный класс.

Решение

Создайте новый файл с расширением *.as* и именем, соответствующим имени класса. Затем добавьте в него описание класса, имеющее такую структуру:

```
package пакет {
    public class Класс {

    }
}
```

Обсуждение

Как уже говорилось, класс – это основной строительный блок всех приложений на ActionScript 3.0. Поэтому, чтобы создавать приложения, необходимо освоить азы написания классов. Для начинающих заметим, что все классы должны размещаться в обычных текстовых файлах, имеющих расширение *.as*. В одном файле может располагаться только одно описание открытого класса. Имена файла и класса должны быть одинаковыми. Например, если имя класса *Example*, тогда его описание должно храниться в файле под названием *Example.as*.

В ActionScript 3.0 все классы должны размещаться в пакетах. Пакет – это средство организации классов в группы. В ActionScript 3.0 пакет тождествен каталогу файловой системы. Пакеты соответствуют пути к классу (что подробно обсуждается в рецепте 2.2), но в этом вводном обсуждении будем считать, что путь к классу – это и путь к проекту (файлу *.fla* в случае Flash, основному классу или документу MXML в случае Flex). Следовательно, пакет верхнего уровня аналогичен корню проекта. Описание пакета всегда стоит самым первым в файле класса. Вот пример синтаксиса:

```
package имя {

}
```

Когда класс определяется как часть пакета верхнего уровня, имя пакета задавать не надо. В этих случаях описание пакета имеет следующий вид:

```
package {

}
```

Когда файл класса сохраняется в подкаталоге, имя пакета соответствует относительному пути подкаталога (имена каталогов разделяются точками (.)). Например, если файл сохраняется в подкаталог (относительно корня проекта) *example*, описание пакета имеет вид:

```
package example {

}
```

Если файл класса сохраняется в подкаталоге *example* под именем *subpackage* (подпакет), описание пакета будет таким:

```
package example.subpackage {

}
```

Пакеты являются важной частью работы с классами, потому что они позволяют предотвратить конфликты имен классов. Например, два разработчика вполне могут написать два разных класса и назвать их *MessageManager* (ДиспетчерСообщений). Эти два класса могут иметь одинаковые имена, но отвечать за очень разные задачи. Один может

управлять сообщениями электронной почты, а другой заниматься передачей двоичных сообщений через сетевые соединения. В одной области видимости не может быть двух классов с одинаковыми именами. Если это происходит, компилятор не знает, какой из классов использовать. Единственный выход – всегда давать классам уникальные имена.

В этом примере можно было бы назвать классы *EmailManager* и *BinarySocketMessageManager*. Однако есть веские причины, по которым такой подход не всегда возможен или приемлем. Проект потенциально может использовать сотни классов, поэтому очень сложно обеспечить уникальность имен всех классов. Более того, многие проекты могут задействовать уже существующие библиотеки кода, созданные другими разработчиками. А так как многие классы библиотеки могут быть взаимосвязаны с другими классами, изменение их имен вызвало бы большие сложности. Вот здесь на помощь приходят пакеты. Даже несмотря на то, что в одном каталоге не может быть двух одинаково названных классов, таких классов может быть сколько угодно, их просто нужно сохранять в разных каталогах. Один класс *MessageManager* можно поместить в пакет *net.messaging.email*, а другой *MessageManager* – в пакет *net.messaging.binarysocket*.

Приведенные имена пакетов на первый взгляд кажутся приемлемыми, однако можно обеспечить более высокую степень их уникальности. Как правило, лучше всего выбирать имя пакета, ориентируясь на его владельца и/или проект. Имена пакетов принято начинать с доменных имен, написанных в обратном порядке. Например, если корпорация Example Corp (*examplecorp.com*) создает классы ActionScript 3.0, она помещает их все в пакет *com.examplecorp* (или подпакет *com.examplecorp*). Тогда если другая Example Corp из Великобритании (*examplecorp.co.uk*) тоже пишет классы ActionScript 3.0, она может обеспечить уникальность, назвав пакет *uk.co.examplecorp*.

Данная рекомендация не относится к классам, составляющим продукт или библиотеку, область действия которых шире, чем компания/организация. К примеру, все собственные классы Flash Player располагаются в пакете Flash (например, *flash.net.URLRequest*) и все классы библиотеки «*Сборника рецептов ActionScript 3.0*» находятся в пакете *ascb* (смотрите *http://www.rightactionscript.com/ascb*).

Если классы представляют собой часть стандартной библиотеки, задействованной в нескольких проектах в пределах компании/организации, то их можно поместить в подпакеты основного пакета. Так, если вышеупомянутые классы *MessageManager* – это часть стандартной библиотеки, используемой многими приложениями Example Corp, то они могут размещаться в пакетах *com.examplecorp.net.messaging.email* и *com.examplecorp.net.messaging.binarysocket*. Если класс представляет собой часть конкретного приложения, то он должен находиться в специальном подпакете этого приложения. Например, у Example

Corp может быть приложение *WidgetStore*. Если приложение *WidgetStore* использует класс *ApplicationManager*, то он должен быть помещен в пакет *com.examplecorp.widgetstore* или его подпакет.

Имена пакетов начинаются со строчной буквы.

Следующий шаг – объявить сам класс:

```
public class Имя {

}
```

В соответствии с соглашением об именовании имя класса начинается с прописной буквы. Имя класса должно подчиняться тем же правилам именования, что и переменные, и функции (состоят из букв, цифр и символов подчеркивания, не могут начинаться с цифры). Описание класса располагается в описании пакета. Следующий код описывает класс *Example* в рамках пакета верхнего уровня:

```
package {
    public class Example {

    }
}
```

Тело класса в описании класса заключено в фигурные скобки. Оно состоит из свойств и методов. Свойства – это переменные, ассоциированные с классом. Они объявляются во многом так же, как переменные, с помощью ключевого слова var. Однако для свойств должны быть заданы еще и атрибуты, определяющие их область видимости. Вот список атрибутов, которые могут использоваться со свойствами:

`private`

Закрытые (`private`) свойства доступны только внутри класса.

`public`

Открытые (`public`) свойства доступны как в классе, так и в экземплярах класса (или по прямой ссылке на класс, когда объявлены как статические (`static`)).

`protected`

Защищенные (`protected`) свойства доступны только в классе и подклассах.

`internal`

Внутренние (`internal`) свойства доступны в рамках пакета.

По умолчанию, если не задан другой атрибут, все свойства внутренние (`internal`). В большинстве случаев свойства должны объявляться с модификаторами `private` или `protected`. Имена закрытых и защищенных

свойств полезно начинать с символа подчеркивания (_). В следующем примере объявляется новое закрытое свойство _id класса *Example*:

```
package {
    public class Example {
        private var _id:String;
    }
}
```

Методы – это функции, ассоциированные с классом. Они объявляются практически так же, как функции, с ключевым словом function. Однако, как и свойства, все методы должны принадлежать пространству имен, определенному одним из атрибутов из приведенного выше списка. Атрибуты public, private, protected и internal для методов и свойств имеют одинаковое действие. Методы должны объявляться с модификатором public, если предполагается вызывать их из экземпляров класса (или из самого класса, если он объявлен static). Если метод будет вызываться только из класса, он должен быть объявлен с модификатором private или protected. Метод должен быть защищенным, если необходима возможность ссылаться на него из подкласса. В следующем фрагменте кода объявляется метод *getId( )*:

```
package {
    public class Example {
        private var _id:String;
        public function getId():String {
            return _id;
        }
    }
}
```

Имена методов подчиняются тем же правилам, что и имена переменных и свойств. Это означает, что имена методов должны содержать только цифры, буквы, символы подчеркивания и знак доллара. Кроме того, несмотря на возможность включать цифры, начинаться с цифры имя метода не может. Имена методов принято начинать со строчных букв. Однако есть одно исключение. В каждом классе может быть особый метод, имя которого аналогично имени класса. Этот особый метод называется *конструктором* и, как следует из названия, он служит для создания новых экземпляров класса. В ActionScript 3.0 все конструкторы должны быть открытыми (public, и в этом отличие от ActionScript 2.0). В отличие от стандартных методов, конструкторы не могут возвращать значения и не должны объявлять возвращаемый тип. В следующем фрагменте кода объявляется метод-конструктор класса *Example*:

```
package {
    public class Example {
        private var _id:String;
        public function Example() {
            _id = ¨Example Class¨;
        }
```

```
        public function getId():String {
            return _id;
        }
    }
}
```

Ниже показано создание экземпляра класса *Example*:

```
var example:Example = new Example();
trace(example.getId()); // На экран выводится: Example Class
```

См. также

Рецепт 2.2.

2.2. Где сохранять класс

Задача

Необходимо определиться с тем, где сохранять файл класса.

Решение

Сохраните файл в каталоге, соответствующем названию пакета. Затем, в случае необходимости, добавьте в путь к классу каталог верхнего уровня.

Обсуждение

Файлы класса всегда должны сохраняться в каталоге, соответствующем пакету класса. Например, класс *com.examplecorp.net.messaging.email.MessageManager* должен храниться в файле *com/examplecorp/net/messaging/email/MessageManager.as*. Компилятор знает, что путь к классам соответствует их пакету. Однако компилятор также должен знать, где искать каталог верхнего уровня, в котором находятся эти подкаталоги. В данном примере компилятор должен знать, где в системе располагается каталог *com*. Где искать, компилятор знает благодаря *пути к классу*. Стандартный путь к классу любого проекта Flex или Flash включает каталог проекта. Например, если каталог сохранен в том же каталоге, что и файл *.fla* (Flash) или основной файл MXML или ActionScript (Flex), компилятор найдет классы. Однако может возникнуть необходимость разместить файлы в другом каталоге. Например, все файлы стандартной библиотеки, используемой во многих проектах, лучше сохранять в одном месте, а не делать ее копии для каждого проекта. Чтобы компилятор знал, где искать все специальные классы, путь к классу можно расширять и редактировать.

Для Flash путь к классу можно редактировать или на уровне проекта, или глобально. На уровне проекта в меню выбираем File→Publish Settings (Файл→Настройки публикации) и нажимаем кнопку ActionScript Settings (Настройки ActionScript) на вкладке Flash. Для гло-

бального пути к классу выбираем Edit→Preferences (Редактировать→Предпочтения) и щелкаем кнопку ActionScript Settings (Настройки ActionScript). В обоих случаях открывается одно и то же диалоговое окно, предоставляющее возможность редактировать путь к классу. Новый каталог в путь к классу добавляется нажатием кнопки +. Например, если каталог *com* для стандартной библиотеки хранится в *C:\libraries*, в путь к классу будет добавлено *C:\libraries*.

Для Flex путь к классу можно задавать только на уровне проекта. Во Flex Builder щелкните проект в обозревателе проекта, выберите Properties (Свойства) и затем выберите Flex/ActionScript Build Path (Путь к сборке Flex/ActionScript) в меню слева. На вкладке Source Path (Путь к исходным файлам) можно добавлять или редактировать путь к классу. Если используется SDK, а не Flex Builder, путь к классу необходимо задавать при компиляции проекта. В *mxmlc* (компилятор командной строки во Flex SDK) добавляем параметр *-source-path*, за которым следует список каталогов, образующих путь к классу, как показано здесь:

```
mxmlc -source-path . C:\libraries ExampleApplication.as
```

См. также

Рецепт 2.1.

2.3. Создание свойств, которые ведут себя как методы

Задача

Чтобы не нарушать инкапсуляцию, надо использовать открытые (`public`) свойства, которые ведут себя как методы.

Решение

Используйте неявные методы, возвращающие и устанавливающие значения свойства (геттер и сеттер соответственно[1]).

Обсуждение

Как упоминалось в рецепте 2.1, все свойства должны быть объявлены `private` или `protected`. Открытые (`public`) свойства не годятся согласно принципу, называемому *инкапсуляцией*. Хорошая инкапсуляция – это то, к чему надо стремиться. Она означает, что класс не позволяет нарушать свою внутреннюю структуру. Открытые свойства делают класс или его экземпляры уязвимыми. Рассмотрим следующий простой пример, в котором фигурирует `public` свойство:

---

[1] Getter (получатель) – метод, возвращающий значение переменной. Setter (установщик) – метод, устанавливающий значение переменной. – *Примеч. науч. ред.*

```
package {
    public class Counter {
        public var count:uint;
        public function Counter() {
            count = 0;
        }
    }
}
```

Можно создать экземпляр класса *Counter* (Счетчик) и изменить значение свойства `count`:

```
var counter:Counter = new Counter();
counter.count++;
```

Но что если согласно бизнес-правилам приложения значение *Counter* не может быть больше 100? Как видите, класс *Counter* с открытым свойством `count` дает возможность без труда нарушить это правило.

Один вариант – использовать геттер и сеттер явно:

```
package {
    public class Counter {
        private var _count:uint;
        public function Counter() {
            _count = 0;
        }
        public function getCount():uint {
            return _count;
        }
        public function setCount(value:uint):void {
            if(value < 100) {
                 _count = value;
            }
            else {
                throw Error();
            }
        }
    }
}
```

Другой вариант – использовать геттер и сеттер неявно. Неявные геттеры и сеттеры объявляются как методы, но выглядят как свойства. Метод, возвращающий значения (геттер), имеет такой синтаксис:

```
public function get имя():Типданных {

}
```

Вот синтаксис метода, устанавливающего значения (сеттер):

```
public function set имя(value: Типданных):void {

}
```

В следующем фрагменте кода описывается свойство `count` с неявными геттером и сеттером:

```
package {
    public class Counter {
        private var _count:uint;
        public function Counter() {
            _count = 0;
        }
        public function get count():uint {
            return _count;
        }
        public function set count(value:uint):void {
            if(value < 100) {
                _count = value;
            }
            else {
                throw Error();
            }
        }
    }
}
```

Теперь с `count` можно работать как с открытым свойством:

```
counter.count = 5;
trace(counter.count);
```

См. также

Рецепт 2.1.

2.4. Создание статических методов и свойств

Задача

Необходимо создать методы и свойства, доступные непосредственно из класса, а не из экземпляров класса.

Решение

Объявляя свойство или метод, указывайте атрибут `static`.

Обсуждение

По умолчанию свойства и методы являются свойствами и методами экземпляра, т. е. определяются в каждом экземпляре класса. Если класс *Example* описывает свойство `_id` и метод *getId( )*, то по умолчанию каждый экземпляр *Example* имеет собственные свойство `_id` и метод *getId( )*. Однако в некоторых классах свойства или методы должны быть ассоциированы с самими классами, а не их экземплярами. Это означает, что неважно, сколько экземпляров класса существует, дан-

ное свойство или метод всего одно. Такие свойства и методы называют *статическими* свойствами и методами.

Примеры статических свойств и методов можно найти в нескольких внутренних классах Flash Player. Например, класс *Math* определяет метод *round( )*. Метод *round( )* статический и, следовательно, доступен прямо из класса:

```
trace(Math.round(1.2345));
```

Весь класс *Math* состоит только из статических методов и констант. Однако класс может включать как статические методы и/или свойства, так и методы и/или свойства экземпляра. Например, класс *String* состоит преимущественно из свойств и методов экземпляра, но метод *fromCharCode( )* объявлен статическим. Метод *fromCharCode( )* возвращает строку, соответствующую передаваемым в него кодам символов. Поскольку метод не ассоциирован ни с одним экземпляром *String*, нет смысла делать его методом экземпляра, но разумно объявить его как статический.

Метод или свойство объявляется статическим с помощью атрибута `static`. Он всегда употребляется в сочетании с атрибутом `public`, `private`, `protected` или `internal`.

Например, здесь объявлено закрытое статическое свойство `example`:

```
static private var _example:String;
```

Порядок расположения атрибутов не имеет значения. Например, `static private` эквивалентно `private static`.

Статические свойства и методы задействуются в шаблоне проектирования Singleton, который позволяет создавать и использовать только один экземпляр данного класса. Singleton-классы имеют `private static` свойство, в котором хранится единственный экземпляр этого класса, а также `public static` метод, обеспечивающий возможность доступа к этому одному экземпляру.

См. также

Рецепт 2.1.

2.5. Создание подклассов

Задача

Необходимо создать класс, который наследуется от существующего класса.

Решение

Напишите подкласс, используя ключевое слово `extends` (расширяет).

Обсуждение

Новый класс может демонстрировать практически аналогичное существующему классу поведение, но являться его более специализированной версией. Вместо того чтобы переписывать всю общую функциональность, можно определить новый класс так, чтобы он наследовал все функциональные возможности существующего. По отношению друг к другу эти классы называются *подклассом* (новый класс) и *надклассом* (существующий класс).

Отношение наследования между классами можно определить с помощью ключевого слова `extends` в описании подкласса:

```
public class Подкласс extends Надкласс
```

Подкласс может использовать все `public` или `protected` свойства и методы надкласса. `Private` свойства и методы недоступны вне класса, даже для подкласса.

Наследование – мощный механизм. Однако, как и любой другой техникой, важно пользоваться им правильно. Прежде чем писать подкласс, необходимо выяснить, действительно ли новый класс должен быть подклассом существующего класса. Между классами может устанавливаться два типа отношений: наследование и композиция. Отношения между классами можно быстро определить посредством вопросов «является» или «имеет»:

- Отношения «является» – это часто отношения наследования. В качестве примера возьмем приложение, управляющее коллекцией библиотеки.
- Отношения «имеет» – это отношения композиции, при которых класс объявляет свойство. Большинство классов используют композицию. Часто композиция обеспечивает те же результаты, что и наследование, и, кроме того, бо**Ъ**шую гибкость (однако обычно для этого требуется большее количество кода). Например, книга не автор, но у нее есть автор (или авторы).

Коллекцию библиотеки образуют разные типы элементов (единицы хранения), к которым относятся и книги, и DVD. Очевидно, что с ними ассоциированы разные типы данных. У книг есть количество страниц и авторы, тогда как для DVD существенно время воспроизведения, исполнители, режиссеры и т. д. Однако есть и определенный общий набор данных, присутствующих как в книгах, так и в DVD. Например, все единицы хранения библиотеки могут быть организованы согласно десятичной системе классификации Дьюи (сокращенно DDC) или иметь уникальные идентификационные номера, присвоенные библиотекой. У каждой единицы хранения также есть название или наименование. В таком случае удобно определить класс, охватывающий общие черты всех элементов библиотеки:

```
package org.examplelibrary.collection {
```

```
    public class LibraryItem {
        protected var _ddc:String;
        protected var _id:String;
        protected var _name:String;

        public function LibraryItem() {}

        public function setDdc(value:String):void {
            _ddc = value;
        }
        public function getDdc():String {
            return _ddc;
        }

        public function setId(value:String):void {
            _id = value;
        }
        public function getId():String {
            return _id;
        }

        public function setName(value:String):void {
            _name = value;
        }
        public function getName():String {
            return _name;
        }
    }
}
```

Тогда можно сказать, что и книги, и DVD относятся к типу *LibraryItem* (ЕдиницаХранения). Затем уместно было бы описать класс *Book* (Книга) и класс *DVD*, представляющие собой подклассы *LibraryItem*. Класс *Book* мог бы выглядеть так:

```
package org.examplelibrary.collection {
    import org.examplelibrary.collection.LibraryItem;
    public class Book extends LibraryItem {
        private var _authors:Array;
        private var _pageCount:uint;

        public function Book() {}

        public function setAuthors(value:Array):void {
            _authors = value;
        }
        public function getAuthors():Array {
            return _authors;
        }

        public function setPageCount(value:uint):void {
            _pageCount = value;
        }
        public function getPageCount():uint {
```

```
                return _pageCount;
            }
        }
    }
```

Проверка на «Является» и «Имеет» полезна, но не всегда имеет решающее значение при выяснении характера отношений между классами. Часто там, где допустимо и уместно наследование, может применяться композиция. В подобных случаях выбор в пользу композиции может быть обусловлен предлагаемой ею гибкостью, которую не обеспечивает наследование. Более того, возможны ситуации, когда классы состоят в отношении «Является», но наследование не подходит. Например, приложение библиотеки обеспечивает пользователям возможность иметь учетные записи. Тогда для представления пользователя придется определить класс *User* (Пользователь). Приложение может различать типы пользователей, например администратор и обычные пользователи; определяем классы *Administrator* (Администратор) и *StandardUser* (Обычный пользователь). В данном примере может показаться, что эти классы устанавливают с классом *User* отношение «Является» (что *Administrator* – это *User*). Однако если разобраться в контексте, *Administrator* на самом деле не *User*, скорее *Administrator* – это роль *User*. Если возможно, было бы лучше описать *User* как класс, имеющий роль типа *Administrator* или *StandardUser*.

По умолчанию можно расширять любой класс. Однако иногда требуется обеспечить отсутствие подклассов некоторому классу. Для этого в описание класса вводится атрибут `final` (конечный)::

```
final public class Example
```

2.6. Реализация версий методов надкласса в подклассе

Задача

Требуется реализовать в подклассе метод, отличающийся от того, который реализован в надклассе.

Решение

Метод надкласса должен быть объявлен с ключевым словом `public` или `protected`. При объявлении реализации метода в подклассе указывается атрибут `override` (переопределить).

Обсуждение

Часто подкласс наследует все методы надкласса как есть, не внося каких-либо изменений в их реализацию. В этих случаях в подклассе метод не переопределяется. Однако иногда подкласс реализует метод не так, как надкласс, и тогда его необходимо переопределить. Переопределить можно только метод, который объявлен в надклассе как `public`

или protected. Тогда в подклассе этот метод можно объявить с атрибутом override. Для примера сначала определим класс *Superclass* (Надкласс):

```
package {
    public class Superclass {
        public function Superclass() {}
        public function toString():String {
            return "Superclass.toString()";
        }
    }
}
```

Далее наследуем *Subclass* (Подкласс) от *Superclass*:

```
package {
    public class Subclass extends Superclass {
        public function Subclass() {}
    }
}
```

По умолчанию *Subclass* наследует метод *toString( )*, поскольку он реализован в *Superclass*:

```
var example:Subclass = new Subclass();
trace(example.toString()); // На экран выводится: Superclass.toString()
```

Если требуется, чтобы метод *toString( )* в *Subclass* возвращал другое значение, необходимо переопределить его в подклассе так:

```
package {
    public class Subclass extends Superclass {
        public function Subclass() {}
        override public function toString():String {
            return "Subclass.toString()";
        }
    }
}
```

При переопределении метода его сигнатура должна строго соответствовать его сигнатуре в надклассе. Это означает, что количество и тип параметров и возвращаемый тип метода в подклассе должны быть точно такими же, как и в надклассе. Если они не идентичны, компилятор формирует ошибку.

Иногда при переопределении метода требуется, чтобы его реализация в подклассе совершенно отличалась от реализации в надклассе. Однако иногда необходимо дополнить реализацию метода в надклассе. В подобных случаях можно вызывать реализацию надкласса из реализации подкласса с помощью ключевого слова super для ссылки на надкласс:

```
super.methodName();
```

См. также

Рецепт 2.5.

2.7. Создание констант

Задача

Необходимо объявить константу.

Решение

Объявляются так же, как свойства, только вместо ключевого слова `var` указывается ключевое слово `const`.

Обсуждение

Как подразумевает термин *constant* (постоянный), значения констант не меняются. Константы удобны как простые идентификаторы при ссылке на сложные значения или когда необходима возможность проверять ошибки во время компиляции. `Math.PI` – пример константы, содержащей сложное значение (значение числа π или 3,14159). `MouseEvent.MOUSE_UP`, в котором хранится значение `mouseUp`, – пример константы, которая обеспечивает возможность выявлять ошибки. При добавлении слушателя события освобождения кнопки мыши можно использовать строковое значение `mouseUp`. Однако в случае опечатки уведомления об ошибке не последует, и код не будет работать, как ожидается:

```
// Это действительный код, но из-за опечатки (mousUp
// вместо mouseUp) он не будет работать правильно.
addEventListener("mousUp", onMouseUp);
```

Выручают константы. Если неправильно указать имя константы, возникнет ошибка компиляции, которая поможет отследить проблему.

```
// Это приводит к ошибке компиляции.
addEventListener(MouseEvent.MOUS_UP, onMouseUp);
```

Синтаксис объявления констант подобен объявлению стандартных свойств, но вместо ключевого слова `var` указывается `const`. Константы, как правило, также являются открытыми и статическими. Чтобы объявить такую константу, необходимо указать соответствующие атрибуты, `public` и `static`. Кроме того, при объявлении константы ей должно быть присвоено значение:

```
static public const EXAMPLE:String = "пример";
```

По соглашению имена констант пишутся прописными буквами. Это помогает выявлять и отличать константы от свойств.

См. также

Рецепт 2.4.

2.8. Передача событий

Задача

Необходимо передавать события.

Решение

Расширьте *flash.events.EventDispatcher* и вызовите метод *dispatchEvent( )*.

Обсуждение

События – это важное средство обмена информацией между объектами. Они неотъемлемая часть при создании гибких систем. Flash Player 9, например, имеет встроенный механизм рассылки событий в классе *flash.events.EventDispatcher*. Все классы, рассылающие события, наследуются от *EventDispatcher* (например, *NetStream* и *Sprite*). Если необходимо определить такой класс, можно расширить *EventDispatcher* так:

```
package {
    import flash.events.EventDispatcher;
    public class Example extends EventDispatcher {

    }
}
```

В классе *EventDispatcher* есть открытые методы *addEventListener( )* и *removeEventListener( )*, которые можно вызывать из любого экземпляра подкласса *EventDispatcher* для регистрации слушателей событий. *EventDispatcher* также определяет защищенный метод *dispatchEvent( )*, который можно вызвать в подклассе для передачи события. Метод *dispatchEvent( )* требует по крайней мере один параметр: объект *flash.events.Event* или подкласс *Event*.

3

Среда времени выполнения

3.0. Введение

Flash Player 9 предлагает достаточно сведений о среде времени выполнения и средств управления ею. Класс *flash.system.Capabilities* имеет множество статических методов, возвращающих информацию о проигрывателе и компьютере, на котором он запущен (например, данные об операционной системе, языке, аудио- и видеовозможностях). Есть другие классы, такие как *flash.display.Stage* и *flash.system.Security*, которые обеспечивают возможность управления другими элементами проигрывателя, такими как меню, выводимое в Windows по щелчку правой кнопкой мыши (по щелчку клавиши Control на Макинтошах), и диалоговое окно Settings (Настройки). Класс *flash.display.Stage* также управляет масштабированием и выравниванием ролика в проигрывателе.

Наверное, самое важное новшество Flash Player 7, относящееся к данной главе, состоит в расширении возможностей работы с контекстным меню по сравнению с предыдущими версиями проигрывателя. Во Flash Player 7 с помощью класса *ContextMenu* (Контекстное меню) можно программно удалять и, что еще важнее, добавлять пункты контекстного меню. И, как следует из имени, контекстное меню можно сделать действительно контекстным, чтобы его пункты соответствовали объекту, на котором меню отображается.

3.1. Определение версии проигрывателя

Задача

Необходимо убедиться, что пользователь работает с требуемой версией Flash Player.

Решение

Для проверки версии проигрывателя и, в случае необходимости, запуска его обновления используйте Flash Player Detection Kit, доступный на сайте Adobe (*http://www.adobe.com/software/flashplayer/download/detection_kit*).

Обсуждение

Определение установленной версии Flash Player в броузере пользователя представляло проблему в течение многих лет. Разработчики применяли множество разных решений, которые можно разделить на три категории:

- Определение версии с помощью сценария на стороне клиента
- Определение версии на стороне сервера
- Определение версии средствами ActionScript

Первый метод для определения версии установленного у пользователя Flash Player использует JavaScript или VBScript. Со многими этими сценариями было связано большое количество ошибок из-за различий платформ и типов броузеров.

С определением версии на стороне сервера могут возникнуть сложности, если нельзя создавать серверные сценарии.

Большинство основанных на ActionScript методик определения версии проигрывателя не будут напрямую работать в *.swf* на ActionScript 3.0. В ActionScript 1.0 и 2.0 существовали различные методы, переменные и функции, возвращающие версию проигрывателя, но теперь в классах ActionScript 3.0 все они недействительны. В ActionScript 3.0 есть собственное средство определения версии проигрывателя – свойство *flash.system.Capabilities.version*. Этот способ, конечно, неприменим ни к одной версии Flash Player до 8.5, поэтому, скорее всего, он бесполезен для определения версии Flash.

Компания Adobe тщательно изучила все эти проблемы и разработала Flash Player Detection Kit, который предлагает на рассмотрение все рекомендуемые процедуры и обеспечивает выбор наилучшего метода определения версии проигрывателя.

Комплект включает документацию по различным вопросам и потенциальные решения, включая VBScript и JavaScript, для определения версии в броузере; файлы *.fla*, *.as* и *.swf* для определения версии с помощью ActionScript; а также сценарии ColdFusion и PHP для определения версии на стороне сервера.

Определение версии средствами ActionScript превосходно работает для Flash Player от версии 4 и выше. По существу сценарий для определения текущей версии проигрывателя выполняет *.swf* Flash 4. От вас требуется лишь задать в сценарии минимальную версию содержимого как переменную. Если версия проигрывателя хотя бы такая же, как

версия содержимого, он загружает указанное содержимое. Если нет, он перенаправляет броузер на страницу с альтернативным содержимым. Эта страница может содержать более раннюю версию *.swf*, не-Flash версию содержимого или уведомление пользователя о необходимости обновить его Flash Player со ссылкой на страницу установки проигрывателя на сайте Adobe.

Более того, комплект содержит *.swf* и шаблон HTML, который инициализирует экспресс-установку (Express Install) последней версии Flash Player. Если версия проигрывателя пользователя не отвечает требованиям, броузер перенаправляется к этому *.swf*, который скачивает последнюю версию Flash Player с сайта Adobe, автоматически устанавливает ее и наконец возвращается к заданному содержимому Flash, и при этом пользователь даже не покидает ваш сайт. Реализация такого варианта требует наличия установленной версии Flash Player 6.0.65.0.

Методы, включенные во Flash Player Detection Kit, обеспечивают очень четкий контроль над версией Flash Player и поставляемым пользователям содержимым.

Для целей тестирования на сайте Macromedia можно найти более ранние версии Flash Player (*http://www.adobe.com/cfusion/knowledgebase/index.cfm?id=tn_14266*).

3.2. Определение операционной системы

Задача

Требуется знать, в какой операционной системе воспроизводится ролик Flash, например для того, чтобы указать, какие операционные системы не поддерживаются, или реализовать ту или иную функциональность, специфическую для данной платформы.

Решение

Задействуйте свойство `flash.system.Capabilities.os`.

Обсуждение

Свойство `flash.system.Capabilities.os` в ActionScript 3.0 возвращает строку с названием операционной системы и именем версии. Возможные значения: Windows XP, Windows 2000, Windows NT, Windows 98/Me, Windows 95 и Windows CE. На Макинтошах строка включает номер версии, например Mac OS 9.2.1 или Mac OS X 10.4.4.

Можно выбирать разные варианты проектных решений исходя из операционной системы. Например, ролик может загружать разные ресурсы в зависимости от операционной системы пользователя. Или можно просто записывать операционные системы пользователей, которые просматривают ролики, для статистического анализа.

Если необходимо выяснить лишь тип платформы, а не конкретную версию, достаточно проверить только первые три буквы строки:

```
var os:String = System.capabilities.os.substr(0, 3);
if (os == "Win") {
  // Здесь располагается код для Windows
} else if (os == "Mac") {
  // Здесь располагается код для Mac
} else {
  // Здесь для UNIX или Linux
}
```

3.3. Проверка типа проигрывателя

Задача

Необходимо знать, на каком типе проигрывателя выполняется файл *.swf*.

Решение

Обратитесь к свойству `flash.system.Capabilities.playerType`.

Обсуждение

Разные типы Flash Player включают:

- Подключаемый модуль, выполняющийся в таких веб-броузерах, как Mozilla или Firefox
- Элемент управления ActiveX для Internet Explorer
- Автономный проигрыватель, который воспроизводит файлы *.swf* вне броузера
- Внешний проигрыватель – это проигрыватель, интегрированный во Flash IDE

Часто информация о проигрывателе, на котором выполняется файл *.swf* в данный момент, крайне важна. Например, при любой интеграции со сценариями броузера (например, JavaScript, VBScript) важно знать, выполняется ли приложение в Internet Explorer или каком-то другом броузере, поскольку при выполнении сценариев эти броузеры могут вести себя по-разному. А если *.swf* выполняется на автономном проигрывателе, сценарии JavaScript и другие вообще недоступны.

Для проверки типа проигрывателя служит свойство `flash.system.Capabilities.playerType`. Оно может возвращать значения `PlugIn` (подключаемый), `ActiveX`, `StandAlone` (автономный) и `External` (внешний). Проверка выполняется с помощью выражения *if*:

```
if(flash.system.Capabilities.playerType == "Plugin") {
  // выполнять действия для броузеров Mozilla и др.
}
```

```
else if(flash.system.Capabilities.playerType == "ActiveX") {
  // выполнять действия для IE
}
else {
  // выполнять действия, предусмотренные на случай отсутствия броузера
}
```

3.4. Проверка языка системы

Задача

Требуется узнать, во-первых, какой язык выбран в качестве системного на компьютере, на котором просматривается ролик, и во-вторых, как пользователь будет вводить текст.

Решение

Возьмите свойство `flash.system.Capabilities.language` и класс `flash.system.IME`.

Обсуждение

Язык компьютера, на котором воспроизводится ролик, можно проверить посредством свойства `flash.system.Capabilities.language`. Оно возвращает двухбуквенный код языка (например, «fr» для французского) в соответствии со стандартом ISO-639-1. Там, где это необходимо, к коду языка через дефис добавляется двухбуквенный код страны (например, «zh-CN» для упрощенного китайского (Simplified Chinese) и «zh-TW» для традиционного китайского (Traditional Chinese)).

Перечень кодов языков можно найти по адресу *http://lcweb.loc.gov/standards/iso639-2/englangn.html* и *http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html*.

Вот пример применения свойства `language` (язык):

```
// Вывод примера: en-US
trace(flash.system.Capabilities.language);
```

Оно позволяет организовать динамическую загрузку содержимого на соответствующем языке:

```
// Создаем ассоциативный массив кодов языков
// для ключей и приветствий для значений.
var greetings:Array = new Array();
greetings["en"] = "Hello";
greetings["es"] = "Hola";
greetings["fr"] = "Bonjour";

// Извлекаем первые два символа кода языка.
var lang:String = flash.system.Capabilities.language.substr(0, 2);
```

```
// Если данного языка нет в списке, используем язык по умолчанию.
if (greetings[lang] == undefined) {
  lang = "en";
}

// Выводим на экран приветствие на соответствующем языке.
trace(greetings[lang]);
```

Существуют разные подходы к обеспечению возможности выбора языка в ролике. Один из них, как показано в предыдущем примере, состоит в том, чтобы создать ассоциативные массивы всех надписей, появляющихся в ролике. А другой – в том, чтобы создать статическое содержимое в нескольких роликах (по одному для каждого языка) и загружать соответствующий ролик исходя из полученного кода языка. В этом случае имя каждого файла *.swf* должно включать код языка, например *myMovie_en.swf*, *myMovie_es.swf*, *myMovie_fr.swf* и т. д.

```
// Получаем язык из объекта capabilities.
var lang:String = flash.system.Capabilities.language.substr(0, 2);

// Создаем массив поддерживаемых языков (т. е. языков,
// которые предусмотрены в созданных роликах).
var supportedLanguages:Array = ["en", "es", "fr"];

// Задаем язык по умолчанию на случай,
// если язык пользователя не поддерживается.
var useLang:String = "en";

// Перебираем поддерживаемые языки в поисках соответствия
// языку пользователя. Если такой язык обнаруживается,
// присваиваем useLang это значение и выходим из оператора for.
for (var i:int = 0; i < supportedLanguages.length; i++) {
  if (supportedLanguages[i] == lang) {
    useLang = lang;
    break;
  }
}

// Загружаем соответствующий ролик.
var movieURL:String =  "myMovie_" + useLang + ".swf");
```

Также нередко требуется знать, как пользователь будет вводить текст. В таких языках, как китайский, японский и корейский, тысячи возможных символов. Чтобы ввести их с клавиатуры, требуется специальная программа, редактор методов ввода (Input Method Editor, IME). Обычно она является частью локализованной операционной системы.

Чтобы выяснить, имеется ли IME в ситеме пользователя, проверяем свойство `flash.system.Capabilities.hasIME`, которое возвращает значение `true` или `false`. Получение дальнейшей информации и взаимодействие с IME осуществляется с помощью класса `flash.system.IME`. Свойство `flash.system.IME.enabled` сообщит, задействует ли пользователь IME или вводит текст прямо с клавиатуры. Это свойство доступно для записи, т. е. может служить для активации IME. На некоторых платформах

и версиях ОС можно взаимодействовать с IME, посылая в него строку для преобразования в корректные символы и принимая результат в выбранное текстовое поле. Поскольку это возможно не во всех системах, лучше сначала проверить ОС (см. рецепт 3.2).

См. также

Рецепт 3.2.

3.5. Определение настроек экрана

Задача

Требуется знать настройки экрана устройства, на котором будет воспроизводиться ролик.

Решение

В этом вам помогут свойства `screenResolutionX` и `screenResolutionY` объекта `flash.system.Capabilities`.

Обсуждение

Для определения настроек экрана устройства, на котором воспроизводится ролик, предназначен объект `flash.system.Capabilities`. Свойства `screenResolutionX` и `screenResolutionY` возвращают разрешение экрана в пикселах.

```
// Вывод примера:
// 1024
// 768
trace(flash.system.Capabilities.screenResolutionX);
trace(flash.system.Capabilities.screenResolutionY);
```

По этим-то значениям мы и определим, как отображать ролик или даже какой ролик загружать. Это приобретает все большее значение по мере увеличения количества портативных устройств, поддерживающих Flash Player. Так, размеры экрана мобильного телефона и обычного настольного компьютера различны, поэтому и загружаться должно разное содержимое, в зависимости от воспроизводящего устройства.

```
var resX:int = flash.system.Capabilities.screenResolutionX;
var resY:int = flash.system.Capabilities.screenResolutionY;

// Если разрешение 240×320 или менее, загружаем версию ролика
// для PocketPC. В противном случае принимаем, что устройство
// является настольным компьютером и загружаем обычное содержимое.
if ( (resX <= 240) && (resY <= 320) ) {
  var url:String = "main_pocketPC.swf";
}
else {
```

```
    var url:String = "main_desktop.swf";
}
loader.load(new URLRequest(url));
```

Также показатели разрешения экрана могут служить для центрирования раскрывающегося окна броузера:

```
var resX:int = flash.system.Capabilities.screenResolutionX;
var resY:int = flash.system.Capabilities.screenResolutionY;

// Задаем переменные для ширины и высоты нового окна броузера.
var winW:int = 200;
var winH:int = 200;

// Определеяем значения X и Y для центрирования окна.
var winX:int = (resX / 2) - (winW / 2);
var winY:int = (resY / 2) - (winH / 2);

// Создаем код, который при передаче в URLLoader.load()
// открывает новое окно броузера.
var jsCode:String = "javascript:void(
         newWin=window.open('http://www.person13.com/'," +
         "'newWindow', 'width=" + winW +
         ", height=" +  winH + "," +
         "left=" + winX + ",top=" + winY + "'));";

// С помощью объекта URLLoader вызываем функцию JavaScript
urlLoader.load(new URLRequest(jsCode));
```

Кроме того, при принятии решения о масштабировании ролика рекомендуется учитывать разрешение экрана. Так, если разрешение экрана слишком велико, скажем, 1600×1200, то некоторые шрифты могут отображаться слишком мелко, и текст станет нечитаемым.

3.6. Масштабирование ролика

Задача

Организовать управление размещением ролика в проигрывателе, включая его масштабирование.

Решение

Обратитесь к свойству `stage.scaleMode`.

Обсуждение

Возможны несколько режимов масштабирования ролика при изменении размеров проигрывателя. Режимы определяются строковыми значениями *exactFit* (точное соответствие), *noBorder* (нет рамки), *noScale* (не масштабируется) и *showAll* (показать все). Однако во избежание опечаток они также определены как статические свойства класса `flash.display.StageScaleMode`: `EXACT_FIT`, `NO_BORDER`, `NO_SCALE` и `SHOW_ALL`.

По умолчанию во Flash Player установлен режим *showAll*. В этом режиме ролик Flash масштабируется соответственно размеру проигрывателя, но сохраняются его пропорции. В результате по краям ролика, если пропорции проигрывателя не соответствуют пропорциям ролика, может появляться рамка. Режим *showAll* можно задать из основного класса приложения так (не забывайте импортировать класс `flash.display.StageScaleMode`):

```
stage.scaleMode = StageScaleMode.SHOW_ALL;
```

Обратите внимание, что `stage` не глобальный объект, а свойство отображаемого объекта, и потому данное выражение работает только в спрайте или другом классе, расширяющем класс `DisplayObject`.

В режиме *noBorder* ролик масштабируется так, чтобы он заполнял область проигрывателя, сохраняя при этом пропорции. Однако этот режим заставляет проигрыватель не отображать рамку вокруг Сцены (Stage). Если пропорции проигрывателя и ролика не совпадают, то ролик будет обрезан. Режим *noBorder* задается так:

```
stage.scaleMode = StageScaleMode.NO_BORDER;
```

Режим *exactFit* масштабирует ролик так, чтобы он помещался в проигрыватель, и в случае необходимости меняет пропорции ролика в соответствии с пропорциями проигрывателя. В результате ролик всегда заполняет всю Сцену, но его элементы могут быть искажены. Например:

```
stage.scaleMode = StageScaleMode.EXACT_FIT;
```

В режиме *noScale* ролик не масштабируется и сохраняет исходные размер и пропорции независимо от размера Сцены. В режиме *noScale* нельзя забывать задавать выравнивание ролика (в рецепте 3.7 есть пример кода, демонстрирующий доступные варианты выравнивания). Например:

```
stage.scaleMode = StageScaleMode.NO_SCALE;
```

Значения свойства `scaleMode` не лишают пользователя возможности менять размеры ролика при помощи контекстного меню. Однако в меню эти опции можно отключить, как показано в рецепте 3.8.

См. также

Рецепты 3.7 и 3.8.

3.7. Изменение выравнивания

Задача

Требуется изменить выравнивание ролика в проигрывателе.

Решение

Задействуйте свойство `stage.align`.

Обсуждение

По умолчанию ролики Flash размещаются в центре проигрывателя. Выравниванием ролика можно управлять, задавая свойство `stage.align`, присутствующее в любом классе, расширяющем `DisplayObject`. Различные режимы выравнивания реализованы в виде строк, например «`T`» для «top» (по верху), «`L`» для «left» (влево) и т. д. Однако во избежание ошибок при написании эти значения также были введены как свойства класса `flash.display.StageAlign`. Они перечислены в табл. 3.1.

Таблица 3.1. Варианты выравнивания, обеспечиваемые классом stage.align

Значение	Выравнивание по вертикали	По горизонтали
`StageAlign.TOP`	По верху	По центру
`StageAlign.BOTTOM`	По низу	По центру
`StageAlign.LEFT`	По центру	Слева
`StageAlign.RIGHT`	По центру	Справа
`StageAlign.TOP_LEFT`	По верху	Слева
`StageAlign.TOP_RIGHT`	По верху	Справа
`StageAlign.BOTTOM_LEFT`	По низу	Слева
`StageAlign.BOTTOM_RIGHT`	По низу	Справа

Не предусмотрено «официальное» значение для центрирования Сцены в проигрывателе и по вертикали, и по горизонтали. Конечно, если такой режим необходим, то не требуется ничего делать, поскольку это режим по умолчанию. Но если режим выравнивания был изменен и надо вернуться к размещению по центру, то любая строка, не соответствующая ни одному из значений свойства `StageAlign`, центрирует Сцену. Проще и безопаснее всего использовать пустую строку `""`.

Следующий класс демонстрирует эффекты от применения различных режимов масштабирования и выравнивания ролика в проигрывателе. Поэкспериментируйте, изменяя значения свойств `stage.scaleMode` и `stage.align` и размеры броузера.

```
package {
  import flash.display.Sprite;
  import flash.display.StageScaleMode;
  import flash.display.StageAlign;

  public class ExampleApplication extends Sprite {
    public function ExampleApplication() {
```

```
            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_RIGHT;

            graphics.beginFill(0xff0000);
            graphics.drawRect(0, 0, stage.stageWidth, stage.stageHeight);
            graphics.endFill();
        }
    }
}
```

3.8. Сокрытие пунктов меню Flash Player

Задача

Требуется скрыть меню, выводимое по нажатию правой кнопки мыши в Windows (или по щелчку кнопкой мыши при нажатой клавише Control на Макинтоше).

Решение

Полностью отключить контекстное меню Flash Player нельзя, но можно сократить до минимума количество отображаемых в нем пунктов, задав для свойства `stage.showDefaultContextMenu` значение `false`.

Обсуждение

По умолчанию контекстное меню Flash Player, появляющееся по щелчку правой кнопкой мыши в Windows (или нажатию Control-Click на Маке), включает следующие пункты:

- Zoom In (Увеличить масштаб)
- Zoom Out (Уменьшить масштаб)
- Show All (Показать все)
- Quality (Low, Medium or High) (Качество (низкое, среднее или высокое))
- Settings (Настройки)
- Print (Печать)
- Show Redraw Regions (Показать перерисованные области) (если используется отладочная версия проигрывателя)
- Debugger (Отладчик) (если используется отладочная версия проигрывателя)
- About Adobe Flash Player 9 (О программе Adobe Flash Player 9)

Многие из этих пунктов, кроме Settings (Настройки), About (О программе) и возможностей отладочной версии Flash Player, можно убрать с помощью такой строки кода на ActionScript:

```
stage.showDefaultContextMenu = false;
```

К сожалению, Flash не позволяет полностью отключить меню. К тому же пользователи Windows привыкли вызывать всплывающее меню броузера, которое, например, позволяет открывать ссылку в новом окне при помощи правой кнопки мыши. Эти возможности недоступны из-за присутствия контекстного меню Flash.

См. также

В рецепте 3.11 показано, как отображать диалоговое окно Flash Settings (Настройки) без щелчка правой кнопкой мыши (в Windows) или нажатия клавиши Control (в Mac).

3.9. Определение аудиовозможностей устройства

Задача

Требуется выяснить, какими аудиовозможностями обладает устройство, на котором выполняется Flash Player.

Решение

Задача решается при помощи свойства `hasAudio` и `hasMP3` класса *flash.system.Capabilities*.

Обсуждение

Свойство `flash.system.Capabilities.hasAudio` возвращает значение `true`, если система пользователя обладает возможностью воспроизведения аудиоданных, и `false` в противном случае. Это исключительно важно при воспроизведении роликов на разных устройствах. Если устройство не имеет аудиоподдержки, пользователю незачем загружать содержимое, которое он не сможет прослушать (особенно если учесть, что аудиоданные бывают достаточно объемными).

```
// Загрузить .swf, содержащий звук, только если
// Проигрыватель может воспроизводить аудиоданные
if (flash.system.Capabilities.hasAudio) {
  content = "sound.swf";
} else {
  content = "silent.swf";
}
// код для загрузки упомянутого .swf
```

Однако то, что система имеет возможность воспроизведения аудиоданных, не означает, что она способна воспроизводить звуки в формате MP3. Проверить это позволяет свойство `flash.system.Capabilities.hasMP3`. Звук в формате MP3 характеризуется лучшим, по сравнению со звуком ADSP, соотношением качества звука и размера файла и потому предпочтительнее ADCP.

```
// Если проигрыватель может воспроизводить MP3,
// загрузить MP3, используя объект Sound.
// В противном случае загрузить во вложенный спрайт
// .swf, содержащий звук ADCP.
if (flash.system.Capabilities.hasMP3) {
  var url:URLRequest = new URLRequest("sound.mp3»);
  sound = new Sound(url);
  sound.play();
} else {
  // код для загрузки внешнего .swf, содержащего звук ADCP
}
```

Важно понимать, что значения свойств `hasAudio` и `hasMP3` зависят от возможностей проигрывателя, а не системы, в которой он выполняется. Проигрыватели, работающие на настольной ОС (Windows, Mac OS и Linux), всегда возвращают `true` для обоих свойств, независимо от того имеется ли в системе оборудование (т. е. звуковая карта и колонки) для воспроизведения звуков. Но проигрыватели, выполняющиеся на других устройствах, могут возвращать `false`, если устройство не поддерживает возможность воспроизведения или формат MP3.

3.10. Определение видеовозможностей устройства

Задача

Требуется выяснить, какими возможностями воспроизведения видеоданных обладает устройство, на котором выполняется Flash Player.

Решение

Обратитесь к свойствам `hasEmbeddedVideo`, `hasStreamingVideo` и `hasVideoEncoder` класса `flash.system.Capabilities`.

Обсуждение

Прежде чем делать попытку доставить видеосодержимое пользователю, необходимо проверить, может ли его система воспроизвести это содержимое, и выбрать способ его доставки. Для Flash-видео это потоковая передача. При этом пользователь может просматривать видеоизображение по мере поступления, а не ожидать загрузки всего (обычно довольно объемного) файла. Способность системы пользователя принимать потоковое видео проверяется при помощи свойства `flash.system.Capabilities.hasStreamingVideo`. Если оно возвращает `false`, то один из вариантов состоит в том, чтобы предоставить проигрывателю для загрузки другой файл *.swf*, содержащий встроенное видео. Однако прежде чем запустить загрузку, необходимо проверить свойство `flash.system.`

Capabilities.hasEbeddedVideo, дабы убедиться, что пользователь может просматривать такое содержимое. Код будет выглядеть примерно так:

```
if(flash.system.Capabilities.hasStreamingVideo) {
  // Код настройки видеопотока и начала потоковой
  // передачи указанных видеоданных
}
else if(flash.system.Capabilities.hasEmbeddedVideo) {
  // Код загрузки внешнего .swf,
  // содержащего встроенные видеоданные
}
else {
  // Альтернативное содержимое без видеоданных
}
```

Аналогично, если приложение требует кодирования видеопотока, как при передаче «живого» изображения с системы пользователя с помощью веб-камеры, требуется убедиться, что система может осуществлять такое кодирование. Проверяется это с помощью свойства flash.system.Capabilities.hasVideoEncoder. Как и в предыдущем примере, проверку лучше осуществлять через выражение *if* и начинать потоковую передачу, только если возвращено true. В противном случае пользователю можно вывести сообщение, объясняющее ситуацию, или перенаправить его на другую страницу.

3.11. Как подсказать пользователю изменить настройки проигрывателя

Задача

Требуется открыть диалоговое окно Settings (Настройки) во Flash Player пользователя, чтобы подсказать пользователю предоставить больший доступ к локальной системе.

Решение

Это можно сделать посредством метода *flash.system.Security.showSettings( )*.

Обсуждение

Метод *flash.system.Security.showSettings( )* открывает диалоговое окно Настройки (Settings) Flash Player, включающее несколько вкладок. Требуемая вкладка передается в метод как строковый параметр. Эти строки сделаны статическими свойствами класса *flash.system.SecurityPanel*, чтобы избежать ошибок из-за опечаток. Возможные значения:

SecurityPanel.CAMERA

Обеспечивает пользователю возможность выбирать камеру.

SecurityPanel.DEFAULT

Открывает вкладку, которая была открыта при последнем просмотре Панели безопасности (Security Panel).

SecurityPanel.LOCAL_STORAGE

Обеспечивает пользователю возможность определить, как хранятся локальные объекты, используемые роликами совместно, включая максимальный размер отведенного для них дискового пространства.

SecurityPanel.MICROPHONE

Обеспечивает пользователю возможность выбрать микрофон и настроить громкость.

SecurityPanel.PRIVACY

Обеспечивает пользователю возможность разрешить или запретить доступ Flash к камере и микрофону.

SecurityPanel.SETTINGS_MANAGER

Открывает новое окно броузера и загружает страницу Settings Manager (Диспетчер настроек), которая предоставляет пользователю дополнительные варианты выбора и возможность вносить коррективы не только для домена данного ролика, но и глобальные изменения.

Если не передать параметр в метод *showSettings( )*, будет взято значение по умолчанию *SecurityPanel.DEFAULT*. Здесь явно задается значение `Local Storage` (Локальное хранилище), поэтому диалоговое окно Settings (Настройки) открывается на вкладке Local Storage.

```
// Открываем диалоговое окно Settings на вкладке Local Storage.
flash.system.Security.showSettings(SecurityPanel.LOCAL_STORAGE);
```

С вашей стороны лучше вежливо предложить пользователю посетить диалог Settings (Настройки), а не просто открывать его без предупреждения. Следует также заранее сообщить, какие настройки следует изменить.

3.12. Безопасность системы

Задача

Требуется загрузить *.swf* в приложение из другого домена и обеспечить ему возможность доступа к коду ActionScript приложения.

Решение

Это можно сделать при помощи методов *flash.system.Security.allowDomain( )*, *flash.system.Security.allowInsecureDomain( )* или файла политики.

Обсуждение

В большинстве случаев все файлы *.swf* приложения располагаются на одном сервере (и в одном домене). Однако иногда приложению требуется

загрузить внешний *.swf* из другого домена. В этом случае ни *.swf*, ни загружающее его приложение не будут иметь доступ к коду друг друга. Возможность доступа обеспечивается посредством методов *flash.system.Security.allowDomain( )*, *flash.system.Security.allowInsecureDomain( )* или файла политики.

Вызываемый (см. ниже) файл *.swf* должен явно разрешить доступ *.swf*-файлам другого домена. Неважно, который из *.swf* загружает или загружается. Чтобы было понятнее, назовем *.swf*, к которому осуществляется доступ, *вызываемым.swf*, а *.swf*, осуществляющий доступ, *вызывающим.swf*. Скажем, *вызывающий.swf* находится в домене *mydomain.com* и загружает *вызываемый.swf* из *otherdomain.com* в объект *content* (содержимое) (рис. 3.1).

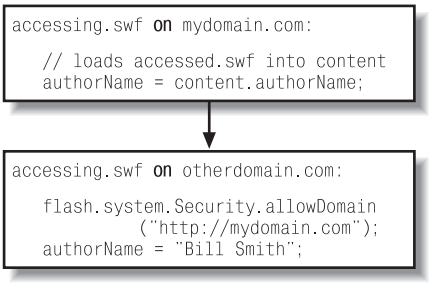


Рис. 3.1. *Применение Security.allowDomain*

Теперь *вызывающий.swf* пытается осуществить доступ к переменной `authorName` (имяАвтора) из загруженного *вызываемого.swf*. Здесь *вызываемый.swf* жалуется и не дает разрешение на доступ файлу *.swf* из другого домена.

Чтобы этого не произошло, в *вызываемом.swf* должна присутствовать следующая строка:

```
flash.system.Security.allowDomain("http://mydomain.com");
```

Благодаря этому он знает, что можно предоставить доступ любому *.swf* из этого домена.

Следует отметить, что разрешение является односторонним. Если теперь загруженному *.swf* понадобиться обратиться к какому-то коду в *.swf*, загрузившем его, он не сможет этого сделать. Для этого загружающему *.swf* надо было бы явно разрешить доступ файлам из *otherdomain.com*.

Домен может быть задан строкой, как в предыдущем примере, или IP-адресом. Также поддерживаются групповые символы. Если по какой-то причине требуется предоставить доступ любому *.swf*, находящемуся где угодно, то можно передать строку `"*"`. Однако это полностью анну-

лирует действие всех средств обеспечения междоменной безопасности, встроенных в проигрыватель, и потому не может быть рекомендовано.

Если *.swf* расположен на защищенном сервере, доступ к которому осуществляется посредством *https://*, по умолчанию он не допустит, чтобы к нему осуществлял доступ любой *.swf*, загруженный с небезопасного домена (*http://*), даже если разработчик разрешил это посредством *flash.system.Security.allowDomain( )*. Возможность доступа файлам из небезопасного домена предоставляется посредством метода *flash.system.Security.allowInsecureDomain( )*.

Приведенный здесь метод требует, чтобы имя или имена доменов были жестко прописаны в коде *.swf*. Для этого надо заранее точно знать, каким доменам будет разрешен доступ, и что их список не изменится. Однако если позже потребуется добавить или убрать домены, то придется менять код, перекомпилировать и повторно размещать *.swf*. Если это не единичный случай, эффективнее создать *файл политики*.

Файл политики – это XML-файл, в котором перечислены все домены, которым разрешен доступ к коду *.swf*. Он имеет такой формат:

```
<?xml version="1.0"?>
<!-- http://www.mydomain.com/crossdomain.xml -->
<cross-domain-policy>
  <allow-access-from domain="www.otherdomain.com" />
  <allow-access-from domain="*.adobe.com" />
  <allow-access-from domain="123.45.67.89" />
</cross-domain-policy>
```

Как видите, здесь просто перечислены домены, которым разрешен доступ. Файл должен называться *crossdomain.xml*. Во Flash до версии 8 требовалось, чтобы этот файл располагался в корневом каталоге домена *.swf*. Теперь файл политики можно задавать и загружать из любого другого каталога посредством метода *flash.system.Security.loadPolicyFile( )*. Он принимает строку, определяющую URL необходимого *crossdomain.xml*. Этот файл должен быть загружен в самом начале работы приложения, до загрузки содержимого из другого домена. Данный метод позволяет добавлять, удалять или изменять список предусмотренных доменов, просто переписывая XML-файл.

Этот метод также поддерживает групповые символы. Например, разрешить доступ любому домену можно так:

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

Если же требуется явно запретить доступ всем доменам кроме текущего, то можно создать пустой файл политики:

```
<cross-domain-policy>
</cross-domain-policy>
```

4

Числа и математические операции

4.0. Введение

Числа не всегда находятся в центре внимания, но их роль в коде нельзя недооценивать. Они могут быть и двоичными, и десятичными, и шестнадцатеричными. Каждое представление занимает собственную нишу, в которой его возможности находят наибольшее применение. Например, шестнадцатеричные числа часто служат для записи RGB-значений цвета, потому что такое представление позволяет легко выделить каждую из трех его составляющих. (Преобразования между системами счисления рассматриваются в рецепте 4.2.)

С числами тесно связана математика. Без математических операций ролики Flash были бы довольно скучными. Такие простые действия, как сложение и вычитание, можно найти в большинстве базовых приложений на ActionScript, а более сложные вычисления, например генерирование случайных чисел и тригонометрические преобразования, являются неотъемлемой частью сложных программ.

В ActionScript 3.0 три основных числовых типа данных: *number*, *int* и *uint*. *number* предназначен для всех чисел с плавающей точкой, тогда как *int* и *uint* представляют целые числа. Разница между *int* и *uint* в том, что *int* – это множество отрицательных и неотрицательных целых, тогда как *uint* – множество неотрицательных целых (целых без знака).

4.1. Представление чисел в разных системах счисления

Задача

Требуется представить значение в двоичной, восьмеричной или шестнадцатеричной системе счисления.

Решение

Шестнадцатеричные литералы начинаются префиксом `0x` (где первый символ – это нуль, а не «о»), а восьмеричные – префиксом `0` (это тоже нуль, а не «о»). Для двоичных чисел нет формы представления, но их можно задавать восьмеричными или шестнадцатеричными эквивалентами или с помощью функции *parseInt( )*, которая преобразует строку в число.

Обсуждение

В ActionScript можно представить число в любом удобном формате. Например, задать значение свойства `Sprite.rotation` лучше всего десятичным числом:

```
rectangleSprite.rotation = 180;
```

Шестнадцатеричные числа удобны для представления RGB-цветов. Например, значение свойства `rgb` объекта *ColorTransform* (преобразование цвета) можно задать (в данном примере) числом `0xF612AB`, представляющим оттенок розового):

```
var pink:ColorTransform = new ColorTransform();
pink.rgb = 0xF612AB;
```

Любой числовой литерал, начинающийся с `0x` или `0X` (где первый символ нуль, а не «о»), считается числом в *шестнадцатеричной системе счисления* (т. е. *шестнадцатеричным*, или по основанию 16). Шестнадцатеричные числа записываются десятичными цифрами от 0 до 9 и буквами от A до F (буквы в верхнем и нижнем регистрах эквивалентны, т. е. `0xFF` равнозначно `0xff`).

Любой числовой литерал, начинающийся с `0` (опять же, нуль, а не «о»), но не с `0x` или `0X`, считается восьмеричным числом (т. е. числом по основанию 8). В восьмеричной системе для представления числа служат десятичные цифры от 0 до 7. Например, `0777` – это восьмеричное число. Восьмеричная система счисления практически не применяется в ActionScript, потому что десятичная система намного удобнее, кроме тех случаев, когда надо записать значение цвета, для чего обычно прибегают к шестнадцатеричным числам. Примеров, когда восьмеричные числа удобнее, чем десятичные или шестнадцатеричные, немного.

В *двоичных числах* (т. е. по основанию 2) допустимы только цифры 0 и 1. Напрямую задать двоичное число нельзя, но можно определить его шестнадцатеричный эквивалент. Четыре двоичных разряда (*бита*) эквивалентны одному шестнадцатеричному. Например, двоичная запись `1111` эквивалентна `F` в шестнадцатеричной записи (15 в десятичной). Число `11111111` в двоичной записи эквивалентно `FF` в шестнадцатеричной (255 в десятичной). Двоичные числа (или их шестнадцатеричные эквиваленты) чаще всего фигурируют в побитовых операциях (`&`, `|`, `^`, `>>`, `<<`, and `>>>`).

См. также

Рецепт 4.2.

4.2. Преобразование из одной системы счисления в другую

Задача

Требуется преобразовать число из одной системы счисления в другую (например, десятичной, двоичной, шестнадцатеричной и т. д.).

Решение

Для преобразования строки в десятичное представление предназначена функция *parseInt( )* с параметром *radix* (*radix* – основание числа). Для преобразования десятичного числа в строковое представление значения по другому основанию служит метод *toString( )* объекта *Number*, *uint* или *int* с параметром *radix*.

Обсуждение

Неважно, как задано числовое значение в ActionScript, результат всегда возвращается в десятичном виде (по основанию 10):

```
// Создаем объект Color
var pink:ColorTransform = new ColorTransform();

// Задаем RGB-значение в шестнадцатеричном виде
pink.rgb = 0xF612AB;

// Здесь значение отображается в десятичном виде: 16126635
trace(pink.rgb);
```

Для вывода значения в другой системе счисления можно применить метод *toString(radix)* объекта *Number*, *uint* или *int*. Он преобразовывает любое число в строку, представляющую это число по заданному основанию.

Следующие два примера преобразовывают числовые литералы в объекты *uint* и выводят строковое представление в формате по основанию 2 (двоичный) и основанию 16 (шестнадцатеричный).

```
// radix - 2, поэтому выводим в двоичной системе
trace(new uint(51).toString(2));  // На экран выводится: 110011
// radix - 16, поэтому выводим как шестнадцатеричное число
trace(new uint(25).toString(16)); // На экран выводится: 19
```

При использовании метода *toString( )* с переменной, содержащей значение числового литерала, Flash перед вызовом метода автоматически создает новый объект *Number*, *uint* или *int*. Хотя это не лучшая практика, с технической точки зрения все правильно и в большинстве при-

ложений разница несущественна. В данном примере переменной присваивается простое число и вызывается метод *toString( )*, чтобы вывести это значение в шестнадцатеричном виде:

```
var quantity:Number = 164;
trace(quantity.toString(16)); // На экран выводится: a4
```

Результатами данных примеров являются не числовые литералы, а строки, такие как `110011`, `19` и `A4`.

В следующем примере задается значение свойства *rgb* объекта *ColorTransform* и вызывается метод *toString( )* для вывода на экран этого значения в шестнадцатеричном виде (в том, в котором оно было введено, хотя прописные символы преобразуются в строчные и результат представляет собой строку, а не число):

```
// Создаем объект Color
var pink:Color = new ColorTransform();

// Задаем RGB-значение в шестнадцатеричном виде
pink.rgb = 0xF612AB;

trace(pink.rgb.toString(16));  // На экран выводится: f612ab
```

Диапазон допустимых значений параметра *radix* метода *toString( )* от 2 до 36. Если метод *toString( )* вызывается без параметра *radix* или для него задано недействительное значение, принимается десятичный (по основанию 10) формат.

Обратное методу *toString( )* действие осуществляет функция *parseInt( )* с параметром *radix*. Она принимает строковое значение и возвращает число. Это требуется при работе с недесятичными числами.

В данных примерах выполняется синтаксический разбор и преобразование строк в двоичные (по основанию 2), шестнадцатеричные (по основанию 16) и десятичные (по основанию 10) числа (обратите внимание, результат всегда десятичный):

```
trace(parseInt("110011", 2));  // На экран выводится: 51
trace(parseInt("19", 16));     // На экран выводится: 25
trace(parseInt("17", 10));     // На экран выводится: 17
```

Если параметр *radix* опущен, то он считается равным 10. За исключением тех случаев, когда строка начинается с `0x`, `0X` или `0`, и он считается равным 16 или 8:

```
trace(parseInt("0x12"));     // radix неявно задан равным 16.
                             // На экран выводится: 18
trace(parseInt("017"));      // radix неявно задан равным 8.
                             // На экран выводится: 15
```

Если значение *radix* указано явно, то неявное не действует. В следующем примере результат равен 0, а не 12. Если число интерпретируется

как десятичное, преобразование прекращается, как только в строке встречается нечисловой символ (*x*):

```
// Число интерпретируется не как шестнадцатеричное, а как десятичное
trace(parseInt("0x12", 10));   // На экран выводится: 0 (не 12 или 18)
```

В следующем примере нет буквенных символов, препятствующих преобразованию, и первым символом является нуль, но число интерпретируется как десятичное, а не восьмеричное:

```
// Число интерпретируется не как восьмеричное, а как десятичное
trace(parseInt("017", 10));   // На экран выводится: 17 (не 15)
```

Не забывайте включать символы `0`, `0x` или явно задавать *radix*. Следующее выражение интерпретирует строку как десятичное число и возвращает `NaN` (нечисло), потому что «A» не может быть преобразовано в целое:

```
trace(parseInt("A9FC9C"));    // NaN
```

4.3. Округление чисел

Задача

Требуется округлить число до ближайшего целого, десятичного разряда или интервала (например, до ближайшего, кратного пяти).

Решение

Для округления числа до ближайшего целого предназначен метод *Math.round( )*. Для округления числа в меньшую или большую стороны – методы *Math.floor( )* и *Math.ceil( )*. Округление числа до заданного количества десятичных разрядов или заданного кратного осуществляется с помощью специального метода *NumberUtilities.round( )*.

Обсуждение

Существует множество причин округления чисел, например вывод на экран результатов вычислений с заданной точностью. Поскольку все арифметические операции в ActionScript осуществляются над числами с плавающей точкой, результатом некоторых вычислений могут быть числа с огромной десятичной частью, которая должна быть округлена. Например, реальным результатом вычисления может быть число 3,9999999, даже если теоретически это 4,0.

Метод *Math.round( )* возвращает ближайшее целое значение любого передаваемого в него параметра:

```
trace(Math.round(204.499));  // На экран выводится: 204
trace(Math.round(401.5));    // На экран выводится: 402
```

Метод *Math.floor( )* округляет в меньшую, а метод *Math.ceil( )* – в большую сторону:

```
trace(Math.floor(204.99));    // На экран выводится: 204
trace(Math.ceil(401.01));     // На экран выводится: 402
```

Чтобы округлить число до ближайшего десятичного разряда:

1. Необходимо определиться, до какого десятичного разряда будет проводиться округление. Например, если требуется округлить 90,337 до 90,34, значит, округление проводится до двух десятичных знаков, т. е. до ближайшей сотой (,01).
2. Делим входную величину на число, выбранное на шаге 1 (в данном случае на ,01).
3. Вызываем *Math.round( )* для округления вычисленного на шаге 2 значения до ближайшего целого.
4. Умножаем результат шага 3 на то же значение, которое использовалось для деления на шаге 2.

Например, число 90,337 округляется до двух десятичных знаков так:

```
trace (Math.round(90.337 / .01) * .01);   // На экран выводится: 9.34
```

Аналогично выполняется округление до ближайшего кратного.

Следующее выражение округляет 92,5 до ближайшего кратного 5:

```
trace (Math.round(92.5 / 5)  *  5);       // На экран выводится: 95
```

Вот другой пример. Округление 92,5 до ближайшего кратного 10:

```
trace (Math.round(92.5 / 10) * 10);       // На экран выводится: 90
```

На практике, наверное, проще иметь дело со специальным методом *NumberUtilities.round( )*, инкапсулирующим эту функциональность. Метод принимает два параметра:

number

Число, которое требуется округлить.

roundToInterval

Интервал, до которого необходимо округлить *number*. Например, если требуется округлить до десятых, интервал равен 0,1. А если требуется округлить число до ближайшего кратного шести, то 6.

Класс *NumberUtilities* находится в пакете *ascb.util*, поэтому прежде всего надо импортировать этот класс. Вот пример применения метода *NumberUtilities.round( )* (следующий код предполагает, что класс *ascb.util.NumberUtilities* импортирован):

```
trace(NumberUtilities.round(Math.PI));        // На экран выводится: 3
trace(NumberUtilities.round(Math.PI, .01));   // На экран выводится: 3.14
trace(NumberUtilities.round(Math.PI, .0001)); // На экран выводится: 3.1416
trace(NumberUtilities.round(123.456, 1));     // На экран выводится: 123
trace(NumberUtilities.round(123.456, 6));     // На экран выводится: 126
trace(NumberUtilities.round(123.456, .01));   // На экран выводится: 123.46
```

4.4. Как добавить ведущие или завершающие нули или пробелы

Задача

Требуется добавить нули или пробелы в начало или конец числа при его отображении в строковом виде.

Решение

Обратитесь к специальному классу *NumberFormat*, примените маску и вызовите метод *format( )*.

Обсуждение

При отображении чисел, скажем времени или дат, может понадобиться форматировать их с помощью подстановки нулей или пробелов в начало или конец числа. Например, время 6 часов 3 минуты требуется представить в формате 6:03 или 06:03, а не 6:3. Кроме того, иногда приходится добавлять пробелы в начало/конец, чтобы выровнять числа в столбце:

```
123456789
  1234567
    12345
```

Конечно же, можно создавать собственные алгоритмы добавления символов в начало/конец строки, но лучше делать это при помощи объекта *NumberFormat*. Это намного быстрее, проще и обеспечивает большую гибкость. Класс *NumberFormat* – это специальный класс, включенный в пакет файлов для скачивания к этой книге по адресу *http://www.rightactionscript.com/ascb*. Он представляет собой часть пакета *ascb.util*, поэтому прежде всего необходимо включить в программу директиву `import`:

```
import ascb.util.NumberFormat;
```

Далее понадобится определить маску, которая будет служить для форматирования числа. Маска может состоять из нулей (`0`), знаков фунта (`#`), точек (`.`) и запятых (`,`); все остальные символы игнорируются.

Нули (0)

Поля подстановки, заполняемые соответствующей цифрой или нулем.

Знаки фунта (#)

Поля подстановки, заполняемые соответствующей цифрой или пробелом.

Точки (.)

Поле подстановки десятичной точки; может быть заменено локальным символом десятичной точки.

Запятые (,)

Поля подстановки для группирующих символов; заменяются локальным группирующим символом.

Чтобы разобраться в этом, полезно рассмотреть несколько примеров; возьмем такую маску:

```
##,###.0000
```

После применения этой маски к числам 1,2345; 12,345; 123,45; 1234,5 и 12345 получены следующие результаты (предполагается, что в локальных настройках в качестве группирующих символов указаны запятые, и десятичных точек – точки):

```
     1.2345
    12.3450
   123.4500
 1,234.5000
12,345.0000
```

Маска для объекта *NumberFormat* задается по-разному. Можно определить маску как параметр при создании объекта:

```
var styler:NumberFormat = new NumberFormat("##,###.0000");
```

Кроме того, для изменения маски в любой момент можно прибегнуть к свойству `mask` объекта:

```
styler.mask = "##.00";
```

Свойство `mask` доступно и для записи, и для чтения, поэтому текущую маску можно получить, считав строковое значение этого свойства.

После применения маски к объекту *NumberFormat* можно форматировать любое числовое значение, вызывая метод *format( )* и передавая в него число как параметр:

```
trace(styler.format(12345);
```

Следующий фрагмент кода представляет собою законченный рабочий пример, иллюстрирующий обсуждаемые возможности класса *NumberFormat*:

```
var styler:NumberFormat = new NumberFormat("#,###,###,###");

trace(styler.format(1));
trace(styler.format(12));
trace(styler.format(123));
trace(styler.format(1234));

styler.mask = "#,###,###,###.0000";

trace(styler.format(12345));
trace(styler.format(123456));
```

```
trace(styler.format(1234567));
trace(styler.format(12345678));
trace(styler.format(123456789));
```

Результат выполнения предыдущего примера (предполагаются локальные настройки США):

```
          1
         12
        123
      1,234
     12,345.0000
    123,456.0000
  1,234,567.0000
 12,345,678.0000
123,456,789.0000
```

По умолчанию объекты *NumberFormat* пытаются автоматически локализовать возвращаемые значения. Если Flash Player выполняется на операционной системе с настройками «U.S. English» (английский США), то запятые выступают в качестве группирующих символов, а точки – в качестве десятичных точек. Если же ОС локализована для Франции, то все наоборот: точки играют роль группирующих символов, а запятые – роль десятичных точек). Есть несколько причин, по которым может потребоваться переопределить автоматическую локализацию:

- Неоходимо, чтобы числа форматировались стандартно, независимо от операционной системы, на которой выполняется Flash-приложение.
- Автоматическая локализация работает некорректно. Это может происходить минимум по двум причинам:
  - Класс *Locale* (служащий *NumberFormat* для определения корректных настроек локализации) может не включать некоторые языки/страны.
  - Flash Player не описывает очень специфические настройки. Он сообщает только код языка. Из-за этого может быть сложно (практически невозможно) правильно определить настройки.

Существует множество способов переопределения настроек автоматической локализации:

- Объект *Locale* передается в метод *format( )* как второй параметр. Тогда метод *format( )* берет настройки объекта *Locale*, а не автоматические. Этот вариант хорош, если требуется применять разные специальные настройки локализации при каждом вызове метода *format( )*. Объект *Locale* можно создать с помощью конструктора. Без параметров объект *Locale* определяет локализацию автоматически, поэтому в него надо передать один или два параметра. Первый параметр – *код языка* (например, *en*), второй – код страны (напри-

мер, *US*); его еще называют *вариантом* (*variant*). Вариант необходимо задавать только для таких регионов, для которых определен один язык, но разное форматирование. Например, код языка *es* (испанский) потенциально применим ко многим странам, включая Мексику (*MX*) и Испанию (*ES*) – в обеих странах для форматирования чисел используются разные символы.

- Задаются свойства *Locale.slanguage* и/или *Locale.svariant* для глобального определения свойств локализации. Далее при вызове метода *format( )* не надо задавать никаких дополнительных параметров. Настройки, заданные при помощи статических свойств *Locale.slanguage* и/или *Locale.svariant* распространяются на все последующие вызовы *format( )*.
- При вызове метода *format( )* в качестве второго параметра выступает объект, свойства которого определяют символы форматирования. У этого объекта должно быть два свойства: *group* и *decimal*. Задавая значения этим свойствам, можно определять символы, служащие для форматирования чисел. Этот вариант лучше всего подходит, когда у объекта *Locale* нет настроек для того или иного региона и/или если используются специальные символы форматирования.

Класс *Locale* находится в пакете *ascb.util*, поэтому его необходимо импортировать.

Следующий пример иллюстрирует некоторые способы переопределения настроек автоматической локализации:

```
var styler:NumberFormat = new NumberFormat("#,###,###,###.00");

Locale.slanguage = "fr";
trace(styler.format(1234));
trace(styler.format(12345, {group: ",", decimal: "."}));
trace(styler.format(123456));
Locale.slanguage = "en";
trace(styler.format(1234567));
trace(styler.format(12345678, new Locale("es", "ES")));
trace(styler.format(123456789, {group: "|", decimal: ","}));
```

Показанный выше код выводит на экран следующее:

```
     1.234,00
    12,345.00
   123.456,00
 1,234,567.00
12.345.678,00
123|456|789,00
```

См. также

Рецепты 4.2 и 4.6.

4.5. Форматирование чисел для отображения без применения маски

Задача

Требуется отформатировать число для отображения без применения маски.

Решение

Создайте объект *NumberFormat* без задания маски, затем вызовите метод *format( )*.

Обсуждение

В рецепте 4.4 обсуждаются разные способы форматирования строкового представления чисел, включая применение масок и нулей и пробелов в начале/конце строки. Однако иногда требуется отформатировать число без всех этих сложностей. Класс *NumberFormat* обеспечивает и такой вариант. Если не указывать маску в объекте *NumberFormat*, метод *format( )* применяет к числу базовое локальное форматирование, как показано в следующем примере:

```
var styler:NumberFormat = new NumberFormat();

trace(styler.format(12.3));
trace(styler.format(123.4));
trace(styler.format(1234.5));
trace(styler.format(12345.6));
```

Обратите внимание, что маска к объекту *NumberFormat* нигде не применялась. Для настроек локализации США результат выполнения предыдущего кода будет таким:

```
12.3
123.4
1,234.5
12,345.6
```

Как и в других случаях (обсуждаемых в рецепте 4.4), метод *format( )* пытается применить автоматическую локализацию. Здесь справедливы те же замечания. Разработчик может предпочесть переопределить настройки автоматической локализации и сделать это теми же способами, которые обсуждались в рецепте 4.4. Это иллюстрирует следующий пример:

```
var styler:NumberFormat = new NumberFormat();

Locale.slanguage = "fr";
trace(styler.format(1234, new Locale("en")));
trace(styler.format(12345, {group: ":", decimal: "|"}));
trace(styler.format(123456));
```

Этот код выведет:

```
1,234
12:345
123.456
```

См. также

В рецептах 4.3 и 4.4 рассказывается, как обеспечить отображение числа с заданным количеством знаков после запятой. После этого для выравнивания чисел достаточно правильно задать выравнивание в текстовом поле при помощи свойства *TextFormat*.`align`. См. также рецепт 4.6.

4.6. Форматирование денежных сумм

Задача

Требуется отформатировать число как денежную сумму в некоторой валюте, например в долларах.

Решение

Обратитесь к методу *NumberFormat.currencyFormat( )*.

Обсуждение

В отличие от некоторых других языков, таких как ColdFusion, в ActionScript нет встроенной функции для форматирования чисел как денежных сумм. Однако специальный класс *NumberFormat* включает метод *currencyFormat( )*, обеспечивающий возможность форматирования в основных валютах.

Методу *currencyFormat( )* необходим хотя бы один параметр, а именно число, которое должно быть отформатировано. Следующий пример иллюстрирует самый простой вариант применения *currencyFormat( )*:

```
var styler:NumberFormat = new NumberFormat();
trace(styler.currencyFormat(123456));
```

Если предыдущий код выполняется на компьютере с настройкой «U.S. English», результат будет таким:

```
$123,456.00
```

Как и метод *format( )* класса *NumberFormat*, обсуждаемый в рецептах 4.4 и 4.5, метод *currencyFormat( )* определяет настройки автоматически. Поэтому если предыдущий код выполняется в операционной системе, локализованной для Испании, то результат будет таким:

```
123.456,00
```

Однако не всегда класс *Locale* (отвечающий за определение местной специфики системы, на которой выполняется приложение) может правильно определить локализацию. Более того, автор программы может просто переопределить автоматическую локализацию, чтобы получать одно и то же значение независимо от того, где выполняется приложение. Существует несколько вариантов переопределения автоматической локализации. Они аналогичны способам, которые применяются с методом *format( )*:

- Указать объект *Locale* в качестве второго параметра при вызове *currencyFormat( )*.
- Присвоить глобальные значения свойствам *Locale.slanguage* и/или *Locale.svariant*.
- Указать объект, явно задающий символы форматирования, в качестве второго параметра при вызове *currencyFormat( )*.

Объект, определяющий символы форматирования для *currencyFormat( )*, немного сложнее, чем для *format( )*. В этом случае он включает следующие четыре свойства: `group`, `decimal`, `currency` и `before`. Свойства `group` и `decimal` играют ту же роль, что и в методе *format( )*. Свойство `currency` (валюта) задает символ валюты. Свойство `before` (перед) – это Булево значение, для которого `true` означает, что символ валюты должен располагаться перед суммой, а `false` – что после суммы.

Вот пример различных вариантов переопределения настроек локализации при помощи метода *currencyFormat( )*:

```
var styler:NumberFormat = new NumberFormat();

trace(styler.currencyFormat(123456));
Locale.slanguage = "nl";
trace(styler.currencyFormat(123456));
trace(styler.currencyFormat(123456, new Locale("sv")));
trace(styler.currencyFormat(123456, {group: ",", decimal: ".", currency: "@",
before: false}));
```

Результат выполнения этого кода:

```
$123,456.00
123.456,00
123,456.00kr
123,456.00@
```

См. также

Создание специальных символов, включая евро (€), йену (Ґ) и британский фунт (J), описывается в рецептах 4.3, 4.5 и в приложении А. Выравнивание денежных сумм в текстовых полях осуществляется посредством задания соответствующего выравнивания поля с помощью свойства *TextFormat*`.align`.

4.7. Генерирование случайных чисел

Задача

Требуется сгенерировать случайное число с помощью ActionScript.

Решение

Для генерирования случайных чисел в диапазоне от 0 до 0,999999999 предназначен метод *Math.random( )*. Случайное число в заданном диапазоне генерируется методом *NumberUtilities.random( )*.

Обсуждение

Метод *Math.random( )* может служить для генерирования случайного числа с плавающей точкой от 0 до 0,999999999. Однако в большинстве случаев в программах возникает необходимость в случайном целом числе. Более того, может потребоваться число в заданном диапазоне. Если все-таки генерируется случайное число с плавающей точкой, может понадобиться задать точность (число десятичных знаков).

Проще всего сгенерировать случайное число в заданном диапазоне и с заданной точностью с помощью метода *NumberUtilities.random( )*. Этот метод принимает три параметра:

minimum

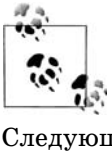
Нижняя граница диапазона, заданная как число типа *Number*.

maximum

Верхняя граница диапазона, заданная как число типа *Number*.

roundToInterval

Необязательный интервал, служащий для округления. Если опущен, числа округляются до ближайшего целого. Можно задавать целые интервалы для округления до целого кратного, а также числа меньше 1 для округления до заданного числа десятичных знаков.



Класс *NumberUtilities* располагается в пакете *ascb.util*, поэтому обязательно включите директиву `import`.

Следующий фрагмент кода иллюстрирует некоторые примеры применения метода *round( )*:

```
// Генерируем случайное целое в диапазоне от 0 до 100.
trace(NumberUtilities.random(0, 100));

// Генерируем случайное кратное 5 в диапазоне от 0 до 100.
trace(NumberUtilities.random(0, 100, 5));

// Генерируем случайное число в диапазоне от -10 до 10,
// округленное до ближайшей десятой.
trace(NumberUtilities.random(-10, 10, .1));
```

```
// Генерируем случайное число в диапазоне от -1 до 1,
// округленное до ближайших пяти сотых.
trace(NumberUtilities.random(-1, 1, .05));
```

Следующий сценарий позволяет убедиться, что генерируемые методом *NumberUtilities.random( )* случайные числа распределены равномерно:

```
package {

  import flash.display.Sprite;
  import ascb.util.NumberUtilities;
  import flash.utils.Timer;
  import flash.events.TimerEvent;

  public class RandomNumberTest extends Sprite {

    private var _total:uint;
    private var _numbers:Object

    public function RandomNumberTest() {
      var timer:Timer = new Timer(10);
      timer.addEventListener(TimerEvent.TIMER, randomizer);
      timer.start();
      _total = 0;
      _numbers = new Object();
    }

    private function randomizer(event:TimerEvent):void {
      var randomNumber:Number = NumberUtilities.random(1, 10, 1);
      _total++;
      if(_numbers[randomNumber] == undefined) {
        _numbers[randomNumber] = 0;
      }
      _numbers[randomNumber]++;
      trace("random number: " + randomNumber);
      var item:String;
      for(item in _numbers) {
        trace("\t" + item + ": " + Math.round(100 * _numbers[item]/_total));
      }
    }

  }
}
```

См. также

Рецепты 4.3 и 4.11.

4.8. Моделирование подбрасывания монеты

Задача

Смоделировать подбрасывание монеты или другое Булево событие (`true/false`), в котором вероятность любого исхода составляет 50%.

Решение

Обратитесь к методу *NumberUtilities.random( )* для генерирования 0 или 1 и затем соотнесите каждый возможный ответ с одним из требуемых результатов.

Обсуждение

Сгенерировать целое число в заданном диапазоне можно посредством метода *random( )* из рецепта 4.7. Чтобы соотнести его результат с событием, имеющим два возможных состояния, как при подбрасывании монеты (`heads` или `tails` (орел или решка)) или при логическом условии (`true` или `false`), связываем каждое случайное целое с одним из состояний. Обычно программисты обозначают одно состояние (например, «выкл») нулем, а противоположное (например, «вкл») единицей, хотя пара чисел может быть любая. Например, вот как можно смоделировать подбрасывание монеты:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.MouseEvent;
  import ascb.util.NumberUtilities;

  public class CoinExample extends Sprite {

    private var _field:TextField;

  public function CoinExample() {
    _field = new TextField();
    _field.autoSize = "left";
    addChild(_field);
    var circle:Sprite = new Sprite();
    circle.graphics.beginFill(0, 100);
    circle.graphics.drawCircle(100, 100, 100);
    circle.graphics.endFill();
      circle.addEventListener(MouseEvent.CLICK, onClick);
      addChild(circle);
  }

  private function onClick(event:MouseEvent):void {
      var randomNumber:Number = NumberUtilities.random(0, 1);

      _field.text = (randomNumber == 0) ? "heads" : "tails";
    }

  }
}
```

В следующем примере проверяется, обеспечивает ли алгоритм *coinFlip( )* равномерное распределение. Стоит ли ожидать идеального распределения 50/50 независимо от количества подбрасываний монеты? Посмотрим.

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import ascb.util.NumberUtilities;

  public class CoinTest extends Sprite {

    private var _field:TextField;

  public function CoinTest() {
    _field = new TextField();
    _field.autoSize = "left";
    addChild(_field);
    var heads:Number = 0;
    var tails:Number = 0;
    var randomNumber:Number;
    for(var i:Number = 0; i < 10000; i++) {
        randomNumber = NumberUtilities.random(0, 1);
        if(randomNumber == 0) {
          heads++;
        }
        else {
          tails++;
        }
    }
    _field.text = "heads: " + heads + ", tails: " + tails;
  }

  }
}
```

Если предполагается, что случайное число понадобится не один раз, то его следует сохранить в переменной (с которой и ведется работа!), а не генерировать каждый раз вновь.

Следующий пример содержит ошибку, потому что в нем независимые случайные числа генерируются в зависимых блоках *else...if*. В некоторых случаях ни одно из условий не выполняется, и метод возвращает пустую строку:

```
package {

  import flash.display.Sprite;
  import ascb.util.NumberUtilities;

  public class RandomLetter extends Sprite {

    public function RandomLetter() {
      for(var i:Number = 0; i < 10000; i++) {
        trace(getRandomLetter());
      }
    }

    private function getRandomLetter():String {
      if(NumberUtilities.random(0, 2) == 0) {
```

```
        return "A";
      }
      else if(NumberUtilities.random(0, 2) == 1) {
        return "B";
      }
      else if(NumberUtilities.random(0, 2) == 2) {
        return "C";
      }
      // Ни одно из предыдущих условий может не выполниться,
      // и метод придет в эту точку без действительной строки.
      return "";
    }
  }
}
```

Вот как это должно выглядеть:

```
package {

  import flash.display.Sprite;
  import ascb.util.NumberUtilities;

  public class RandomLetter extends Sprite {

    public function RandomLetter() {
      for(var i:uint = 0; i < 10000; i++) {
        trace(getRandomLetter());
      }
    }

    private function getRandomLetter():String {
      // Перед проверкой возвращаемого из random()
      // значения присваиваем его переменной.
      var randomInteger:uint = NumberUtilities.random(0, 2);
      if(randomInteger == 0) {
        return "A";
      }
      else if(randomInteger == 1) {
        return "B";
      }
      else if(randomInteger == 2) {
        return "C";
      }
      return "";
    }
  }
}
```

См. также

Рецепт 4.7.

4.9. Моделирование игры в кости

Задача

Требуется смоделировать игру в кости.

Решение

Нам нужен метод *NumberUtilities.random( )* для генерирования случайных чисел в заданном диапазоне.

Обсуждение

Выбрасывание кости (или костей) моделируется путем генерирования случайных целых чисел. В роликах Flash для этого можно вызывать метод *random( )*, рассмотренный в рецепте 4.7. Имитация выбрасывания костей – это важная часть многих игр, создаваемых с помощью ActionScript, и метод *random( )* упрощает эту задачу.

NumberUtilities.random(1, 12) неверно моделирует пару шестигранных игральных костей, потому что возможные значения лежат в диапазоне между 2 и 12, а не 1 и 12. А разве *NumberUtilities.random(2, 12)* обеспечивает требуемый результат? Нет. *NumberUtilities.random(2, 12)* дает равномерное распределение чисел в диапазоне от 2 до 12, тогда как при игре с двумя костями результат 7 имеет большую вероятность, чем 2 или 12. Следовательно, каждая кость должна моделироваться отдельно, и результаты суммироваться. Более того, во многих играх, таких как нарды, ход игры зависит от результата каждой кости в отдельности, а не их суммы, поэтому потребуется сохранять их отдельно.

Нередко случайное число сохраняют в переменной, вместо того чтобы генерировать каждый раз заново. Обратите внимание на отличие этих двух сценариев. В первом `dice` (кости) всегда представляет собой сумму `die1` и `die2`:

```
var die1:uint = NumberUtilities.random(1, 6);
var die2:uint = NumberUtilities.random(1, 6);
var dice:uint = die1 + die2;
```

В следующем сценарии между значением `dice` и случайными значениями, сохраненными в `die1` и `die2`, нет никакой связи. Иначе говоря, даже если сумма `die1` и `die2` составляет 7, в `dice` сохраняется совершенно другое значение в диапазоне от 2 до 12:

```
var die1:uint = NumberUtilities.random(1, 6);
var die2:uint = NumberUtilities.random(1, 6);
var dice:uint = NumberUtilities.random(1, 6) + NumberUtilities.random(1, 6);
```

Метод *NumberUtilities.random( )* позволяет моделировать многогранную кость, задавая любой диапазон значений. Вот пример с диапазоном

от 1 до 15, который моделирует ситуацию выбрасывания 15-гранной кости, что можно использовать в ролевой игре:

```
var die1:uint = NumberUtilities.random(1, 15);
```

В следующем коде метод *NumberUtilities.random( )* применяется в сочетании с программным созданием изображения для визуализации одной кости:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.MouseEvent;
  import ascb.util.NumberUtilities;

  public class NumbersAndMath extends Sprite {

    var _die:Sprite;
    var _value:uint;

    public function NumbersAndMath() {
      _die = new Sprite();
      addChild(_die);
      _die.addEventListener(MouseEvent.CLICK, rollDie);
      rollDie(null);
    }

    private function rollDie(event:MouseEvent):void {
      _value = NumberUtilities.random(1, 6);
      _die.graphics.clear();
      _die.graphics.lineStyle();
      _die.graphics.beginFill(0xFFFFFF);
      _die.graphics.drawRect(0, 0, 50, 50);
      _die.graphics.endFill();
      _die.graphics.beginFill(0x000000);
      if(_value == 1 || _value == 3 || _value == 5) {
        _die.graphics.drawCircle(25, 25, 4);
      }
      if(_value == 2 || _value == 3 || _value == 4 ||
         _value == 5 || _value == 6)
      {
        _die.graphics.drawCircle(11, 11, 4);
        _die.graphics.drawCircle(39, 39, 4);
      }
      if(_value == 4 || _value == 5 || _value == 6) {
        _die.graphics.drawCircle(11, 39, 4);
        _die.graphics.drawCircle(39, 11, 4);
      }
      if(_value == 6) {
        _die.graphics.drawCircle(11, 25, 4);
        _die.graphics.drawCircle(39, 25, 4);
      }
    }

  }
}
```

В результате выполнения этого кода на сцене отображается одна кость. При каждом щелчке по кости, значение на ней изменяется.

См. также

Рецепт 4.7.

4.10. Моделирование карточной игры

Задача

Требуется с помощью ActionScript сдать карты в карточной игре с колодой из 52 карт (без джокеров).

Решение

Задача решается при помощи специального класса *Cards* (карты).

Обсуждение

Моделировать карточную игру сложнее, чем, скажем, выбрасывание пары костей. Поэтому работу с игральными картами в приложениях Flash обеспечивает специальный класс *Cards*.

Класс *Cards* находится в пакете *ascb.play*, поэтому его надо импортировать, прежде чем задействовать в коде.

```
import ascb.play.Cards;
```

Посредством конструктора можно создать новый объект *Cards*:

```
var cards:Cards = new Cards();
```

По умолчанию объект *Cards* создает стандартную колоду из 52 игральных карт. Дальнейшая работа с картами обеспечивается посредством метода *deal( )*. Этот метод возвращает массив объектов *CardHand* (рука с картами). При вызове метода *deal( )* необходимо задать хотя бы один параметр, количество игроков.

```
// Участвует четыре игрока.
var hands:Array = cards.deal(4);
```

По умолчанию метод раздает все карты колоды (за исключением случаев, когда 52 не делится нацело на число игроков). В некоторых карточных играх сдаются не все карты колоды. Поэтому можно задать второй необязательный параметр, определяющий количество карт, раздаваемых каждому игроку:

```
// Карты раздаются четырем игрокам по пять карт каждому.
var hands:Array = cards.deal(4, 5);
```

Каждый объект *CardHand* – это массив объектов *Card*. Класс также предоставляет интерфейс для вытягивания и сбрасывания карт, которые были сданы игроку. Для этого применяется метод *discard( )* (сбросить), в котором в качестве параметров задаются индексы карт. Указанные карты удаляются из руки и возвращаются в низ колоды:

```
// Сбросить карты с индексами 0 и 4 из объекта CardHand,
// хранящегося в первом элементе массива hands.
hands[0].discard(0, 4);
```

И наоборот, метод *draw( )* (вытянуть) может применяться для взятия карт из колоды. Метод *draw( )* по умолчанию, если не задан ни один параметр, снимает одну верхнюю карту колоды. Если за один раз надо вытянуть несколько карт, их число можно задать как параметр:

```
// Снимаем одну верхнюю карту колоды и добавляем ее в объект,
// хранящийся в первом элементе массива hands.
hands[0].draw();

// Снимаем четыре верхних карты колоды и добавляем их в объект,
// хранящийся в четвертом элементе массива hands.
hands[3].draw(4);
```

Количество карт у игрока можно узнать с помощью свойства `length` (длина) объекта *CardHand*, а методом *getCardAt( )* можно извлечь карту под заданным индексом.

Как было сказано, *CardHand* образован объектами *Card*. Объекты *Card*, в свою очередь, обладают четырьмя свойствами: `value` (значение), `name` (имя), `suit` (масть) и `display` (представление). Свойство `value` возвращает числовое значение от 0 до 12, где 0 – это двойка, а 12 – туз. Свойство `name` возвращает название карты, например 2, 10, Q или A. Свойство `suit` возвращает трефы (clubs), бубны (diamonds), черви (hearts) или пики (spades). Свойство `display` возвращает имя карты и масть через пробел. Следующий пример кода иллюстрирует применение класса *Cards*:

```
package {

  import flash.display.Sprite;
  import ascb.play.Cards;
  import flash.utils.trace;

  public class CardExample extends Sprite {

    public function CardExample() {
      var cards:Cards = new Cards();
      var hands:Array = cards.deal(4, 10);
      var i:uint;
      var j:uint;
      for(i = 0; i < hands.length; i++) {
        trace("hand " + i);
        for(j = 0; j < hands[i].length; j++) {
```

```
            trace(hands[i].getCardAt(j));
          }
        }
      }

    }
  }
```

4.11. Генерирование уникального числа

Задача

Требуется сгенерировать уникальное число, например добавляемое к URL для предотвращения кэширования URL.

Решение

Это делается при помощи метода *NumberUtilities.getUnique( )*.

Обсуждение

Чаще всего уникальные числа генерируются для составления уникальных URL (чтобы предотвратить их кэширование). Добавлением уникального номера в конец URL обеспечивается его отличие от любого другого URL; следовательно, броузер получает данные с удаленного сервера, а не из локального кэша.

Метод *NumberUtilities.getUnique( )* возвращает число исходя из текущего времени в миллисекундах. (В следующем примере предполагается, что класс *ascb.util.NumberUtilities* импортирован.)

```
// Выводим на экран уникальное число.
trace(NumberUtilities.getUnique());
```

В большинстве случаев этот код возвращает текущее время в миллисекундах. Однако может возникнуть необходимость генерировать ряд уникальных чисел чаще, чем каждую миллисекунду. В этом случае метод *getUnique( )* добавляет случайное число ко времени в миллисекундах, чтобы гарантировать уникальность числа. Следующий код генерирует несколько чисел за одну миллисекунду:

```
for(var i:Number = 0; i < 100; i++) {
  trace(NumberUtilities.getUnique());
}
```

4.12. Преобразование единиц измерений углов

Задача

Для работы с угловыми значениями в ActionScript требуется преобразовывать их в соответствующие единицы.

Решение

Задача решается при помощи классов *Unit* и *Converter*.

Обсуждение

Значения свойства `_rotation` клипа задаются в градусах, а во всех остальных случаях углы в ActionScript задаются в радианах. Это может быть плохо по двум причинам. Во-первых, если требуется задать свойство на основании результата выполнения одного из тригонометрических методов ActionScript, приходится преобразовывать значения из радиан в градусы. Во-вторых, люди, как правило, предпочитают работать с градусами, которые необходимо преобразовывать в радианы перед передачей в любой тригонометрический метод. К счастью, преобразовать радианы в градусы и наоборот очень просто. Чтобы преобразовать радианы в градусы, достаточно умножить значение на 180/Math.PI. Обратное преобразование, из градусов в радианы, выполняется посредством умножения на Math.PI/180. Однако намного удобнее работать со специальными классами *Unit* и *Converter*.

Unit и *Converter* – это два класса пакета *ascb.unit*, облегчающие преобразования между различными единицами измерения, включая градусы, радианы и грады (окружность составляет 400 град). Первый шаг – создание экземпляра класса *Unit*, описывающего тип единицы измерения, которую требуется преобразовать. Класс *Unit* предоставляет большую группу констант, делающих работу с ним очень удобной. Константы `Unit.DEGREE`, `Unit.RADIAN` и `Unit.GRADIAN` возвращают новые объекты *Unit*, которые представляют градусы, радианы и грады, соответственно. Объекты *Unit* имеют целый ряд свойств, включая `name`, `category`, `label` (надпись) и `labelPlural` (надпись во множественном числе):

```
var degree:Unit = Unit.DEGREE;
trace(degree.name);        // На экран выводится: degree
trace(degree.category);    // На экран выводится: angle
trace(degree.label);       // На экран выводится: degree
trace(degree.labelPlural); // На экран выводится: degrees
```

Когда создан экземпляр *Unit*, представляющий единицы, которые необходимо преобразовать, можно извлекать экземпляр *Converter*, который может осуществить преобразование в определенные единицы измерения. Для этого предназначен метод *getConverterTo( )*, в который передается ссылка на объект *Unit*, представляющий единицы измерения, в которые требуется провести преобразование. Например, следующий фрагмент кода создает объект *Converter*, преобразующий градусы в радианы:

```
var converter:Converter = Unit.DEGREE.getConverterTo(Unit.RADIAN);
```

Создав экземпляр *Converter*, можно вызвать метод *convert( )* (преобразовать), задавая преобразуемое значение. Например:

```
trace(converter.convert(90));
```

Метод *convertWithLabel( )* преобразовывает значение в строку, включающую соответствующую надпись на случай, если потребуется вывести значение на экран:

```
var converterToRadians:Converter = Unit.DEGREE.getConverterTo(Unit.RADIAN);
var converterToDegrees:Converter = Unit.RADIAN.getConverterTo(Unit.DEGREE);
trace(converterToRadians.convertWithLabel(1));
trace(converterToRadians.convertWithLabel(57.2957795130823));
trace(converterToDegrees.convertWithLabel(1));
trace(converterToDegrees.convertWithLabel(0.0174532925199433));

/*
   На экран выводится:
   0.0174532925199433 radians
   1 radian
   57.2957795130823 degrees
   1 degree
*/
```

Если понадобится выполнить обратное преобразование, то посредством метода *getConverterFrom( )* можно также создать экземпляр *Converter*, который преобразовывает одни единицы измерения в другие. Например:

```
var converter:Converter = Unit.DEGREE.getConverterFrom(Unit.GRADIAN);
trace(converter.convert(100));
trace(converter.convert(23));
```

4.13. Вычисление расстояния между двумя точками

Задача

Требуется вычислить расстояние между двумя точками.

Решение

Обратитесь к методам *Math.pow( )* и *Math.sqrt( )* в сочетании с теоремой Пифагора.

Обсуждение

Расстояние (по прямой) между любыми двумя точками можно вычислить с помощью теоремы Пифагора. *Теорема Пифагора* гласит, что в любом прямоугольном треугольнике (треугольнике, один из углов которого составляет 90 градусов) длина *гипотенузы* (самой длинной стороны) равна корню квадратному из суммы квадратов двух других сторон (называемых в прямоугольном треугольнике *катетами*). Теорема записывается так:

```
a2 + b2 = c2
```

По этой формуле можно вычислить расстояние между любыми двумя точками, где *a* – разность координат точек по оси X, *b* – разность координат точек по оси Y и *c* (искомое расстояние) равняется корню квадратному из (*a2* + *b2*). В ActionScript это записывается так:

```
var c:Number = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
```

Как вычислить расстояние между двумя точками с помощью прямоугольного треугольника? Сразу этого можно и не заметить, но для любых двух точек в системе координат Flash можно построить воображаемый прямоугольный треугольник, как показано на рис. 4.1.

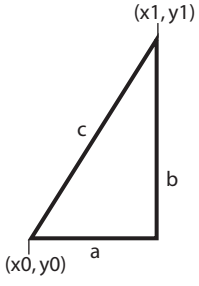


Рис. 4.1. *Чтобы вычислить расстояние между двумя точками, соединяем их гипотенузой прямоугольного треугольника*

Линия, соединяющая две точки, является гипотенузой воображаемого треугольника. Катеты образованы горизонтальной и вертикальной линиями, выходящими из этих точек. Длины катетов можно найти, найдя разности между координатами *X* и *Y*. Длина катета *a* определяется разностью координат точек по оси *X*, а длина катета *b* – разностью координат точек по оси *Y*. Зная длины катетов *a* и *b*, можно по теореме Пифагора вычислить длину гипотенузы *c*, т. е. расстояние между точками (что и требовалось найти).

4.14. Определение координат точки окружности

Задача

Требуется найти координаты точки окружности по заданным радиусу окружности и углу поворота.

Решение

Задача решается при помощи методов *Math.sin( )* и *Math.cos( )*.

Обсуждение

Зная тригонометрию, найти координаты точки окружности просто. Рассмотрим формулы, которые можно использовать в коде на ActionScript, и связанную с ними теорию.

Пусть на Сцене задана точка – назовем ее `p0` – с координатами (`x0`, `y0`), а также некоторое расстояние и угол к горизонтали. По этим данным и некоторым тригонометрическим формулам можно найти координаты другой точки – назовем ее `p1` – с координатами (`x1`, `y1`). Угол образован воображаемой линией, соединяющей `p0` и `p1`, и линией, параллельной оси X, как показано на рис. 4.2. *Противолежащий катет* – это сторона треугольника, дальше всех отстоящая от угла. *Прилежащий катет* – это сторона, которая вместе с гипотенузой образует угол.

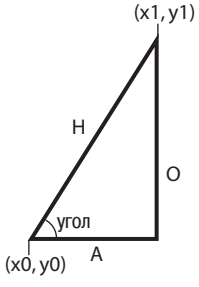


Рис. 4.2. *Угол, прилежащий катет, противолежащий катет и гипотенуза прямоугольного треугольника*

Если известно расстояние между двумя точками и угол к горизонтали, то с помощью тригонометрических функций можно вычислить координаты X и Y соответствующей точки. Синус угла равен отношению противолежащего катета к гипотенузе:

```
sine(угол) = противолежащий/гипотенуза
```

Находим длину противолежащего катета:

```
противолежащий = sine(угол) * гипотенуза
```

Как видите, противолежащий катет представляет приращение координаты Y.

Косинус угла равен отношению прилежащего катета к гипотенузе:

```
cosine(угол) = прилежащий/гипотенуза
```

Выражаем длину прилежащего катета:

```
прилежащий = cosine(угол) * гипотенуза
```

Как видно из рис. 4.2, прилежащий катет представляет приращение по оси X.

Поскольку длины противолежащего и прилежащего катетов показывают изменения координат X и Y, суммируя эти значения с исходными координатами X и Y, можно вычислить координаты новой точки.

Но как все это поможет найти точки окружности? Все станет ясным, если взглянуть на рис. 4.3, где показан треугольник, вписанный в окруж-

ность. Гипотенуза треугольника – это радиус окружности, а угол треугольника – это угол поворота (центральный угол, образованный осью X и радиусом, проведенным к искомой точке) интересующей точки окружности.

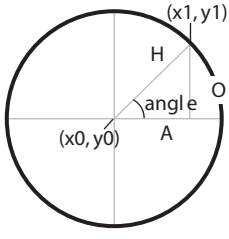


***Рис. 4.3.** Определение координат точки окружности*

Следовательно, координата X точки окружности определяется произведением радиуса и косинуса угла. Координата Y определяется произведением радиуса и синуса угла. Вот код на ActionScript для определения координат точки `p1` по заданным радиусу и центральной точке (`p0`):

```
x1 = x0 + (Math.cos(angle) * radius);
y1 = y0 + (Math.sin(angle) * radius);
```

По этим формулам можно определить координаты любой точки окружности по заданным координатам центра и радиусу. Изменяя угол во времени, можно моделировать движение по окружности.

В следующем примере тригонометрические уравнения служат для перемещения спрайта по окружности:

```
package {

  import flash.display.Sprite;
  import ascb.units.Converter;
  import ascb.units.Unit;
  import flash.events.Event;

  public class NumbersAndMath extends Sprite {

    private var _square:Sprite;
    private var _angle:uint;

    public function NumbersAndMath() {
      _square = new Sprite();
      _square.graphics.lineStyle(0);
      _square.graphics.drawCircle(0, 0, 20);
      addChild(_square);
      _angle = 0;
      addEventListener(Event.ENTER_FRAME, move);
    }

    private function move(event:Event):void {
      var converter:Converter = Unit.DEGREE.getConverterTo(Unit.RADIAN);
```

```
            var angleRadians:Number = converter.convert(_angle);
            _square.x = Math.cos(angleRadians) * 100 + 200;
            _square.y = Math.sin(angleRadians) * 100 + 200;
            _angle++;
        }

    }
}
```

См. также

Рецепты 7.7 и 7.8.

4.15. Преобразование единиц измерения

Задача

Требуется осуществлять преобразования между градусами по шкале Фаренгейта и градусами Цельсия, фунтами и килограммами или другими единицами измерения.

Решение

Это делается при помощи классов *Unit* и *Converter*.

Обсуждение

В мире используется множество различных систем измерения. Например, температура измеряется в градусах Фаренгейта и градусах Цельсия. Массу можно измерять в фунтах, но широко распространена метрическая система, где в качестве единицы массы приняты килограммы. Расстояние может измеряться в милях, а не километрах, или дюймах, а не сантиметрах. Поэтому может возникнуть необходимость преобразования одних единиц измерения в другие.

Формально фунты – единицы измерения веса, а килограммы – массы. Вес – это сила, с которой тело вследствие его притяжения к планете действует на опору или подвес, а масса – постоянная характеристика тела, не зависящая от эффектов гравитации. Таким образом, вес объекта на Луне и Земле различен, поскольку действуют разные силы притяжения, а масса объекта остается неизменной. Однако в применении к объектам на поверхности Земли понятия массы и веса обычно используются взаимозаменяемо.

Каждое из преобразований осуществляется по определенному алгоритму. Например, чтобы перевести градусы Цельсия в градусы Фаренгейта, требуется умножить значение на 9, разделить на 5 и добавить 32 (чтобы осуществить обратное преобразование, из числа вычитается 32, результат умножается на 5 и делится на 9). Аналогично, чтобы перевес-

ти фунты в килограммы, надо умножить значение на 2,2, или разделить на 2,2, чтобы выполнить обратное преобразование. (В рецепте 4.12 было показано, как переводить градусы в радианы и наоборот.)

Как и в случае с преобразованиями единиц измерения углов, для преобразования других типов единиц измерения можно использовать классы *Unit* и *Converter*. Класс *Unit* поддерживает множество разных видов измерений (углов, температуры, объема и т.д.). Массив поддерживаемых категорий измерений можно получить с помощью метода *Unit.getCategories( )*:

```
// Выводим на экран поддерживаемые категории измерений.
trace(Unit.getCategories());
```

Для каждой категории поддерживается множество разных единиц измерения. Например, в категории измерения углов поддерживаются градусы, радианы и грады. Список поддерживаемых единиц измерения для данной категории можно получить с помощью статического метода *Unit.getUnits( )*. В метод передается имя интересующей категории, и он возвращает массив поддерживаемых единиц измерения для этой группы. Если параметр опущен, будет возвращен список всех поддерживаемых единиц измерения:

```
// Выводим на экран единицы измерения,
// поддерживаемые в категории температура.
trace(Unit.getUnits("temperature"));
```

Для задания единиц, подлежащих преобразованию, используются встроенные константы класса *Unit*, как в рецепте 4.12. Затем с помощью методов *getConverterTo( )* или *getConverterFrom( )* извлекается объект *Converter*. В следующем примере создается объект *Converter* для вычисления значений по шкале Фаренгейта, эквивалентных заданным по шкале Цельсия:

```
var converter:Converter = Unit.CELCIUS.getConverterTo(Unit.FAHRENHEIT);
```

Затем, конечно же, можно применить метод *convert( )* (и/или *convertWithLabel( )*) для преобразования значений:

```
trace(converter.convert(0));  // На экран выводится: 32
```

См. также

Пример применения описанной в коде функции для осуществления одного типа преобразования единиц измерения приведен в рецепте 4.12, который включает функцию для преобразования градусов в радианы и функцию, осуществляющую обратное преобразование.

5

Массивы

5.0. Введение

Успешное программирование на ActionScript немыслимо без массивов.

Они обеспечивают возможность группировать взаимосвязанные данные и затем организовывать и обрабатывать их. Идея массива близка каждому. Мы постоянно пользуемся ими в повседневной жизни. Обычный список покупок или текущих дел можно назвать массивами. Адресная книга – это массив, содержащий имена людей, адреса, даты рождения и другую информацию. Библиотеки отслеживают книги с помощью системы индексации, посредством чего каждая книга становится, по существу, элементом массива библиотеки.

В ActionScript существует два типа массивов: массивы с целочисленными индексами и ассоциативные. Массивы обоих типов группируют взаимосвязанные данные, но используют разные средства доступа к ним.

Массив с целочисленными индексами

Использует целые (числа) в качестве уникальных идентификаторов элементов массива. Такие массивы упорядочиваются *индексом* (т. е. номером элемента), начиная с 0. Каждый элемент занимает нумерованную ячейку массива. Массивы с целочисленными индексами идеальны для наборов данных, с которыми предполагается работать последовательно.

Ассоциативный массив

Для организации доступа к каждому значению использует строковые *ключи*. Более подробно об ассоциативных массивах – в рецепте 5.15.

Основная масса рецептов данной главы посвящена массивам с целочисленными индексами, и, если не указано обратное, термин «массив» подразумевает именно такой массив.

Конечно, прежде чем работать с массивом, надо знать, как его создать. ActionScript позволяет создавать массивы двумя способами: с помощью функции-конструктора и как литерал массива. Все массивы представляют собой члены класса *Array* (массив). Возможны три варианта создания нового экземпляра массива с помощью функции-конструктора *Array( )*:

```
// Создаем пустой массив.
var array:Array = new Array();

// Создаем массив, состоящий из elements неопределенных элементов.
var array:Array = new Array(elements);

// Создаем массив заданных элементов.
var array:Array = new Array(element0,...elementN);
```

Литеральная нотация также создает массив, но без функции-конструктора. Посредством литеральной нотации удобно задавать элементы в момент создания:

```
var letters:Array = ["a", "b", "c"];
```

Некоторые методы класса *Array* изменяют существующий массив, для которого были вызваны, другие возвращают новый массив (предлагая косвенный способ создания массивов).

Элементы массива задаются и извлекаются с помощью оператора доступа к массиву (квадратные скобки) и индекса интересующего элемента, например:

```
// Присвоим пятому элементу массива значение "яблоки"
// (индексация массива начинается с 0).
items[4] = "яблоки";

// Выводим в окне Output пятый элемент.
trace(items[4]);   // На экран выводится: яблоки
```

ActionScript не интересует, элементы какого типа хранятся в массиве. Это могут быть строки, числа, Булевы значения или ссылки на любые объекты. И, в отличие от более строгих языков программирования, в одном массиве даже можно сохранять разные типы данных. Например, в этом массиве находится строка, целое, Булево значение и объект:

```
var data:Array = ["a", 2, true, new Object()];
```

В отличие от многих языков программирования, в ActionScript не обязательно указывать число элементов массива при его объявлении.

5.1. Добавление элементов в начало или конец массива

Задача

Требуется добавить элементы в существующий массив.

Решение

Метод *push( )* предназначен для добавления элементов в конец массива, а метод *unshift( )* – в начало массива.

Обсуждение

Добавление элементов в конец массива осуществляется методом *Array.push( )*. В него передается одно или более вставляемых значений:

```
var array:Array = new Array();
array.push("val 1", "val 2");
```

Можно добавить один элемент с помощью свойства `length` (длина) в качестве индекса. В ActionScript нумерация элементов массива начинается с нуля, поэтому последнему элементу соответствует индекс *Array*.`length` – 1. Следовательно, размещение элемента в ячейке с индексом *Array*.`length` создает новый элемент сразу после существующего последнего элемента. Например:

```
array[array.length] = "val 3";
```

Если задается элемент с несуществующим индексом, массив автоматически расширяется, чтобы включить необходимое количество элементов. Если есть промежуточные элементы, они инициализируются как `undefined` (неопределенный). Например, в массиве `letters` (буквы) после выполнения следующих выражений будут располагаться элементы `["a", "b", "c", undefined, undefined, "f"]`:

```
var letters:Array = ["a", "b", "c"];
letters[5] = "f";
```

Добавление элементов в массив применяется очень часто, например, если требуется компоновать массив постепенно или сохранять историю действий пользователя с целью реализации возможности кнопки возврата или предыстории.

Для добавления элементов в начало массива применяется метод *unshift( )*. Он сдвигает все элементы на один индекс вправо и вставляет новый элемент в ячейку с индексом 0:

```
// Создаем массив с четырьмя элементами:
// "a", "b", "c" и "d".
var letters:Array = new Array();
letters.push("a", "b", "c", "d");

// Добавляем "z" в начало массива. При этом сдвигаются
// все остальные элементы, так что индекс значения "a"
// меняется с 0 на 1 и т. д.
letters.unshift("z");

// Выводим на экран результаты, перебирая все
// элементы массива. См. рецепт 5.2.
for (var i:int = 0; i < letters.length; i++) {
```

```
    trace(letters[i]);
}
```

Куда добавлять элементы: в начало или конец массива? Это зависит главным образом от того, как предполагается осуществлять доступ к ним и их удаление. Например, если доступ к элементам должен выполняться по схеме «последним пришел, первым вышел» («last in, first out», LIFO), то элементы добавляются посредством метода *Array.push( )*, а для удаления элементов в обратном порядке применяется метод *Array.pop( )*.

См. также

Рецепт 5.2.

5.2. Перебор всех членов массива

Задача

Требуется последовательно осуществить доступ к каждому элементу массива.

Решение

Организуйте цикл *for*, в котором приращение индексной переменной осуществляется от 0 до достижения конца массива (*Array*.`length`). Используйте эту переменную для получения доступа к элементам массива по очереди.

Обсуждение

Для организации доступа к значениям, хранящимся в элементах массива, осуществляется перебор элементов массива с помощью цикла *for*. Поскольку первый индекс массива – `0`, отсчет значений индексной переменной оператора *for* должен начинаться с `0`. Последний индекс массива всегда на `1` меньше значения свойства `length` этого массива. В теле цикла *for* индексная переменная для доступа к элементам массива указывается в квадратных скобках. Например:

```
var letters:Array = ["a", "b", "c"];
for (var i:int = 0; i < letters.length; i++) {
  // Выводим элементы в панели Output.
  trace("Элемент " + i + ": " + letters[i]);
}
```

Индексная переменная цикла (в примере это *i*) должна меняться в диапазоне от `0` до значения, на единицу меньшего значения свойства `length`. Запомните, последний индекс массива всегда на единицу меньше его длины (`length`).

Также цикл *for* можно применять для перебора элементов в обратном порядке от *Array*.length – 1 до 0, уменьшая индексную переменную на единицу при каждом проходе. Обратный перебор нужен, если требуется найти последний соответствующий определенному критерию элемент, а не первый (см. рецепт 5.3), например:

```
var letters:Array = ["a", "b", "c"];
for (var i:int = letters.length - 1; i >= 0; i--){
  // Выводим на экран элементы в обратном порядке.
  trace("Элемент " + i + ": " + letters[i]);
}
```

Перебор всех элементов массива применяется часто. Например, перебирая элементы массива, содержащие ссылки на спрайты, можно выполнить определенное действие над каждым спрайтом:

```
for (var i:int = 0; i < sprites.length; i++){
  // Перемещаем каждый спрайт на один пиксел вправо.
  sprites[i].x++;
}
```

Длину (length) массива можно сохранить в переменную, а не вычислять при каждом проходе цикла. Например:

```
var length:int = sprites.length;
for (var i:int = 0; i < length; i++){
  // Перемещаем каждый спрайт на один пиксел вправо.
  sprites[i].x++;
}
```

В результате получаем небольшое повышение производительности, потому что Flash теперь не приходится вычислять длину при каждой итерации. Однако этот вариант годится, только если в цикле не происходит добавления или удаления элементов, что меняет длину массива. В таком случае лучше вычислять длину при каждом проходе.

См. также

Рецепт 12.8, в котором описываются способы перебора символов строки. Рецепт 5.16, в котором подробно рассматривается перечисление элементов ассоциативного массива. См. также рецепт 5.3.

5.3. Поиск элементов массива, соответствующих некоторому критерию

Задача

Требуется найти первый элемент массива, соответствующий заданному значению.

Решение

Чтобы решить задачу, надо применить выражение *for* для перебора элементов массива и выражение *break* для выхода из цикла при выявлении соответствия. Применяйте на свой выбор методы *ArrayUtilities.findMatchIndex( )*, *ArrayUtilities.findLastMatchIndex( )* и/или *ArrayUtilities.findMatchIndices( )*.

Обсуждение

При поиске первого элемента массива, соответствующего заданному значению, задействуется выражение *for*, как показано в рецепте 5.2. В него добавляется выражение *break* (для выхода из цикла при выявлении соответствия.

Выражение *break* прерывает выполнение цикла. *break* следует размещать в выражении *if*, чтобы оно выполнялось только при соблюдении определенного условия.

В случае поиска только первого соответствующего элемента выражение *break* важно вдвойне. Во-первых, оно избавляет от необходимости перебирать оставшиеся элементы массива после выявления соответствия, что привело бы к пустой трате машинного времени. В следующем примере выражение *break* обеспечивает выход из цикла после второй итерации, избавляя от еще шести ненужных проходов. (А если бы элементов было на тысячу больше!)

Более того, выражение *break* жизненно важно при поиске первого соответствия, поскольку гарантирует, что будет выбран только первый элемент, а все последующие соответствия будут проигнорированы. Если в следующем примере опустить *break*, то на экран будут выведены все соответствующие элементы, а не только первый из них.

```
// Создаем массив из восьми элементов.
var letters:Array = ["a", "b", "c", "d", "a", "b", "c", "d"];

// Определяем, что ищем.
var match:String = "b";

// Используем выражение for для перебора, потенциально,
// всех элементов массива.
for (var i:int = 0; i < letters.length; i++) {

    // Проверяем, соответствует ли текущий
    // элемент заданному значению.
    if (letters[i] == match) {

        // Осуществляем некоторое действие над соответствующим
        // элементом. В данном примере выводим на экран
        // сообщение для целей тестирования.
        trace("Элемент с индексом " + i +
              " соответствует " + match);
```

```
        // Включаем выражение break, чтобы выйти из цикла
        // for, как только будет найдено соответствие.
        break;
    }
}
```

Поиск последнего соответствующего элемента массива осуществляется путем перебора его элементов в обратном порядке. Индексная переменная инициализируется значением *Array*.length – 1. При каждой итерации ее значение уменьшается на 1, и цикл выполняется до тех пор, пока она не станет равной 0:

```
var letters:Array = ["a", "b", "c", "d", "a", "b", "c", "d"];

var match:String = "b";

// Перебираем элементы массива в обратном порядке.
// В данном примере "b" под индексом 5.
for (var i:int = letters.length - 1; i >= 0; i--) {
    if (letters[i] == match) {
        trace("Элемент с индексом " + i +
              " соответствует " + match);
        break;
    }
}
```

Поиск соответствующих элементов можно упростить, применив некоторые статические методы класса *ArrayUtilities*. Этот класс располагается в пакете *ascb.util*, поэтому прежде всего его надо импортировать:

```
import ascb.util.ArrayUtilities;
```

В классе *ArrayUtilities* имеется три метода для поиска элементов, соответствующих определенному критерию – *findMatchIndex( )*, *findLastMatchIndex( )* и *findMatchIndices( )*. Метод *findMatchIndex( )* (найти индекс соответствия) требует минимум два параметра: ссылку на массив, в котором осуществляется поиск, и значение, которое требуется сопоставить. Метод возвращает или индекс первого соответствующего заданному критерию элемента, или -1, если соответствие не найдено. Например:

```
var letters:Array = ["a", "b", "c", "d"];

trace(ArrayUtilities.findMatchIndex(letters, "b"));
// На экран выводится: 1

trace(ArrayUtilities.findMatchIndex(letters, "r"));
// На экран выводится: -1
```

Можно задать начальный индекс, с которого начинается поиск. Так можно выбрать индекс соответствия, следующий за первым найденным. Начальный индекс задается как третий параметр:

```
var letters:Array = ["a", "b", "c", "d", "a", "b", "c", "d"];
```

```
trace(ArrayUtilities.findMatchIndex(letters, "a", 1));
// На экран выводится: 4
```

Этот метод позволяет также выявлять неполные соответствия. По умолчанию выбираются только точные соответствия. Однако если в качестве третьего параметра задать значение `true`, метод находит все элементы, содержащие заданную подстроку:

```
var words:Array = ["bicycle", "baseball", "mat", "board"];

trace(ArrayUtilities.findMatchIndex(words, "s", true));
// На экран выводится: 1
```

Если требуется выбирать частичные соответствия, причем не с начала массива, начальный индекс просто передается как четвертый параметр.

Метод *findLastMatchIndex( )* (найти индекс последнего соответствия) работает аналогично методу *findMatchIndex( )*, за исключением того, что он начинает поиск с конца массива.

Метод *findMatchIndices( )* (найти индексы соответствий) возвращает массив индексов для всех элементов, соответствующих переданному значению. Два параметра метода обязательны: массив и элемент, для которого идет поиск соответствий. Например:

```
var letters:Array = ["a", "b", "c", "d", "a", "b", "c", "d"];

trace(ArrayUtilities.findMatchIndices(letters, "b"));
// На экран выводится: 1,5
```

Метод *findMatchIndices( )* может выбирать и частичные соответствия. Для этого достаточно в качестве третьего параметра задать Булево значение `true`:

```
var words:Array = ["bicycle", "baseball", "mat", "board"];

trace(ArrayUtilities.findMatchIndices(words, "b", true));
// На экран выводится: 0,1,3
```

Все описанные методы *ArrayUtilities* задействуют одни и те же базовые приемы с выражением *for*. Рассмотрим код этих методов. Метод *findMatchIndex( )* довольно прост, комментарии дают необходимые пояснения. Однако хотелось бы обратить внимание, что в этом методе в цикле *for* место выражений *break* занимают выражения *return*. В контексте функции или метода выражение *return* осуществляет выход из выражения *for*, поэтому *break* не нужен.

```
public static function findMatchIndex(array:Array, element:Object):int {
    // Используем переменную для определения
    // индекса, с которого начинается цикл.
    // Используем значение по умолчанию 0.
    var startingIndex:int = 0;

    // По умолчанию частичные соответствия не учитываются.
```

```
    var partialMatch:Boolean = false;

    // Если третий параметр - число, присваиваем его startingIndex.
    // В противном случае, если четвертый параметр - число,
    // присваиваем startingIndex его значение.
    if(typeof arguments[2] == "number") {
        startingIndex = arguments[2];
    }
    else if(typeof arguments[3] == "number") {
        startingIndex = arguments[3];
    }

    // Если третий параметр - Булево значение,
    // присваиваем его partialMatch.
    if(typeof arguments[2] == "boolean") {
        partialMatch = arguments[2];
    }

    // Если соответствие не найдено.
    var match:Boolean = false;

    // Перебираем все элементы массива, начиная
    // с заданного начального индекса.
    for(var i:int = startingIndex;
            i < array.length; i++) {

        // Проверяем, соответствует ли элемент заданному
        // критерию полностью или частично.
        if(partialMatch) {
            match = (array[i].indexOf(element) != -1);
        }
        else {
            match = (array[i] == element);
        }

        // Если элемент соответствует критерию,
        // возвращаем его индекс.
        if(match) {
          return i;
        }
    }

    // Данное выражение return будет достигнуто,
    // только если соответствия не выявлены.
    // В этом случае возвращается -1.
    return -1;
}
```

Метод *findLastMatchIndex( )* практически идентичен методу *findMatchIndex( )*, за исключением того, что осуществляет перебор в обратном направлении. Метод *findMatchedIndices( )* перебирает элементы массива в поисках всех соответствующих критерию элементов. Из их индексов он формирует массив, добавляя каждый последующий индекс в конец, и после выполнения возвращает этот массив. Он

использует метод *findMatchIndex( )*, как показано в следующем фрагменте кода:

```
public static function findMatchIndices(array:Array,
element:Object, partialMatch:Boolean = false):Array {
    var indices:Array = new Array();
    var index:int = findMatchIndex(array,
                                   element,
                                   partialMatch);
    while(index != -1) {
        indices.push(index);
        index = findMatchIndex(array,
                               element,
                               partialMatch,
                               index + 1);
    }
    return indices;
}
```

См. также

Рецепты 5.2 и 5.10.

5.4. Удаление элементов

Задача

Требуется удалить из массива один или более элементов и сместить все оставшиеся элементы, чтобы заполнить освободившиеся ячейки.

Решение

Элемент из середины массива удаляется посредством метода *splice( )* последний элемент – метода *pop( )* и первый элемент – метода *shift( )*.

Обсуждение

Для уничтожения элементов массива начиная с определенного индекса применяется метод *splice( )* (срастить). В этот метод передаются два параметра:

`start`

Индекс элемента массива, с которого следует начинать уничтожение элементов.

`deleteCount`

Количество подлежащих уничтожению элементов. Если это значение `undefined`, удалены будут все элементы массива, начиная с индекса, заданного значением *start*:

```
var letters:Array = ["a", "b", "c", "d"];
```

```
// Удаляем один элемент из letters, начиная с индекса 1.
letters.splice(1, 1);

// Выводим на экран результаты. Теперь в массиве три
// элемента: "a", "c" и "d".
for (var i:int = 0; i < letters.length; i++) {
    trace(letters [i]);
}
```

Также метод *splice( )* возвращает новый массив, содержащий удаленные элементы, например:

```
var letters:Array = ["a", "b", "c", "d"];

// Удаляем два элемента из letters, начиная с индекса 0.
var deleted:Array = letters.splice(0, 2);

// Выводим на экран удаленные элементы: "a" и "b".
for (var i:int = 0; i < deleted.length; i++) {
    trace(deleted[i]);
}
```

Один элемент в начале или в конце массива удаляется методами *shift( )* (сдвинуть) и *pop( )* (выбросить). Метод *shift( )* удаляет первый элемент массива и возвращает его значение. Метод *pop( )* удаляет последний элемент массива и возвращает его значение:

```
var letters:Array = ["a", "b", "c", "d"];

// Удаляем первый элемент и выводим его значение.
trace(letters.shift());

// Удаляем последний элемент и выводим его значение.
trace(letters.pop());

// Выводим на экран оставшиеся элементы.
// В массиве осталось два элемента: "b" и "c".
for (var i = 0; i < letters.length; i++) {
    trace(letters[i]);
}
```

При удалении элементов массива в выражении *for* необходимо соответствующим образом изменить значение индексной переменной. Следующий пример показывает, что может произойти, если не обновить значение индексной переменной:

```
var numbers:Array = new Array(4, 10);
numbers[4] = 1;
trace(numbers);  // На экран выводится: 4,10,undefined,undefined,1
for(var i:int = 0; i < numbers.length; i++) {
    if(numbers[i] == undefined) {
        numbers.splice(i, 1);
    }
}
trace(numbers);  // На экран выводится: 4,10,undefined,1
```

В предыдущем фрагменте кода предполагалось, что элементы, показанные в результате как неопределенные, будут удалены из массива. Однако последнее выражение `trace` свидетельствует о том, что удален только один элемент. Если поэтапно разобрать последовательность выполнения выражения *for*, можно понять, почему это произошло:

1. В первых двух итерациях ничего не происходит, потому что первые два элемента не `undefined`.
2. При третьей итерации обнаруживается `undefined` элемент и удаляется, а четвертый и пятый элементы сдвигаются на один индекс влево, превращаясь в третий и четвертый.
3. Следующая итерация проверяет новый четвертый элемент, который теперь является последним. Получается, что другой `undefined` элемент (ставший теперь третьим) оказывается пропущенным. Чтобы этого не произошло, необходимо уменьшать значение индексной переменной после удаления элемента. Следующий фрагмент кода показывает, как это можно сделать:

```
var numbers:Array = new Array(4, 10);
numbers[4] = 1;
trace(numbers);  // На экран выводится: 4,10,undefined,undefined,1
for(var i:int = 0; i < numbers.length; i++) {
  if(numbers[i] == undefined) {
    numbers.splice(i, 1);
    i--;
  }
}
trace(numbers);  // На экран выводится: 4,10,1
```

5.5. Как вставить элементы в середину массива

Задача

Требуется вставить элементы в середину массива.

Решение

Используйте метод *splice( )*.

Обсуждение

Метод *splice( )* может использоваться как для удаления, так и для вставки элементов массива. Значения, передаваемые в метод *splice( )* после первого и второго параметров, вставляются в ячейки массива, начиная с индекса, заданного параметром *start*. Все существующие элементы, находящиеся правее заданного индекса, сдвигаются вправо, чтобы вместить в массив вставляемые значения. Если в метод *splice( )* в качестве параметра *deleteCount* (счетчик удаления) передается 0, элементы не удаляются, а новые значения вставляются:

```
var letters:Array = ["a", "b", "c", "d"];

// Вставляем три строковых значения ("r", "s"
// и "t"),начиная с индекса 1.
letters.splice(1, 0, "r", "s", "t");

// теперь в letters семь элементов:
// "a", "r", "s", "t", "b", "c" и "d".
for (var i:int = 0; i < letters.length; i++) {
    trace(letters[i]);
}
```

Также можно одновременно удалить и вставить новые элементы:

```
var letters:Array = ["a", "b", "c", "d"];

// Удаляем из letters два элемента
// и вставляем три,начиная с индекса 1.
letters.splice(1, 2, "r", "s", "t");

// теперь в myArray пять элементов:
// "a", "r", "s", "t" и "d".
for (var i:int = 0; i < letters.length; i++) {
    trace(letters[i]);
}
```

5.6. Преобразование строки в массив

Задача

Имеется список значений в виде строки. Требуется преобразовать ее в массив отдельных элементов.

Решение

Используйте метод *String.split( )*.

Обсуждение

Метод *split( )* (разделить) класса *String* (строка) разбивает строку, содержащую список значений, в массив. В списке должны использоваться унифицированные разделители. Например, список *Susan,Robert,Paula* разделяется запятыми.

Метод *split( )* принимает два параметра:

delimiter (разделитель)

Подстрока, используемая для разделения элементов списка. Если `undefined`, весь список помещается в первый элемент нового массива.

limit (предел)

Максимальное число элементов в новом массиве. Если `undefined`, в массив помещаются все элементы списка.

При разделении строки в массив слов в качестве разделителя можно использовать пробел:

```
var list:String = "Peter Piper picked a peck of pickled peppers";
// Разделить строку, используя пробел как разделитель.
// Тогда каждое слово будет помещено в отдельный
// элемент нового массива words.
var words:Array = list.split(" ");
```

Метод *split( )* исключительно полезен, когда значения во Flash загружаются с помощью объекта *URLLoader* или другой подобной техники загрузки данных. Например, с сервера можно получить список имен в виде строки:

```
names=Michael,Peter,Linda,Gerome,Catherine
```

Можно упростить работу с именами, преобразовав этот список в массив с помощью метода *split( )*:

```
// Предположим, для загрузки данных
// используется URLLoader.
var namesData:String = _loader.data;
var names:Array = namesData.split(",");
```

См. также

Рецепт 5.7.

5.7. Преобразование массива в строку

Задача

Требуется преобразовать массив в строку.

Решение

Используйте метод *join( )*.

Обсуждение

ActionScript предоставляет встроенную возможность быстрого преобразования массивов в строки (предполагается, конечно, что элементы массива сами являются или строками, или данными другого типа, который может быть автоматически приведен к строке) с помощью метода *join( )*. В метод *join( )* передается строка, указывающая Flash, какой разделитель должен использоваться между элементами:

```
var letters:Array = ["a", "b", "c"];
trace(letters.join("|"));   // На экран выводится: a|b|c
```

Если не указан разделитель, по умолчанию Flash использует запятую:

```
var letters:Array = ["a", "b", "c"];
trace(letters.join());      // На экран выводится: a,b,c
```

Метод *toString( )* делает то же самое, что и *join( )*, без параметров или с запятой в качестве параметра. Кстати, если использовать массив там, где требуется строка, Flash автоматически вызывает метод *toString( )* следующим образом:

```
var letters:Array = ["a", "b", "c"];
trace(letters);  // На экран выводится: a,b,c
```

См. также

Рецепт 5.6.

5.8. Создание отдельной копии массива

Задача

Требуется создать точную копию (дубликат) массива – массив, содержащий все элементы исходного, но не являющийся очередной ссылкой на оригинал.

Решение

Используйте метод *concat( )* или метод *slice( )*. По выбору можно использовать метод *ArrayUtilities.duplicate( )*. Метод *duplicate( )* может создавать рекурсивные дубликаты.

Обсуждение

Поскольку массивы являются составным типом данных, сравниваются и копируются они не так, как элементарные типы. Переменная, содержащая массив, на самом деле не хранит данные массива. Она просто указывает на область памяти в компьютере, где эти данные находятся. Это имеет смысл с точки зрения оптимизации. Элементарные данные обычно имеют небольшой размер, например отдельное число или короткая строка. Но составные данные, такие как массив, могут быть очень объемными. Было бы неэффективно копировать весь массив каждый раз, когда требуется произвести над ним некоторое действие или передать его в функцию. Поэтому при копировании массива ActionScript не делает отдельной копии данных массива. Проиллюстрируем это простым примером.

Сначала рассмотрим копирование элементарных данных из переменной `quantity` в другую переменную, `newQuantity`:

```
// Присвоим переменной число 5.
var quantity:int = 5;

// Скопируем значение quantity в другую переменную, newQuantity.
var newQuantity:int = quantity;
```

```
// Изменяем значение quantity.
quantity = 29;

trace(quantity);        // На экран выводится: 29
trace(newQuantity);     // На экран выводится: 5
```

При копировании содержимое quantity копируется в newQuantity. После копирования последующие изменения quantity не оказывают никакого влияния на newQuantity (и наоборот), потому что при копировании элементарных типов *копируются значения*.

Теперь рассмотрим аналогичную операцию с массивами и обратим внимание на различия. Переменная letters присваивается переменной newLetters, но обе переменные просто ссылаются на один и тот же массив в памяти. При изменении letters изменяется и значение newLetters:

```
// Задаем элементы массива.
var letters:Array = ["a", "b", "c"];

// Копируем letters в другую переменную, newLetters.
var newLetters:Array = letters;

// Оба массива, как и ожидалось,
// содержат одинаковые значения.
trace(letters);         // На экран выводится: "a,b,c"
trace(newLetters);      // На экран выводится: "a,b,c"

// Меняем значение letters.
letters[0] = "d";
letters[1] = "e";
letters[2] = "f";

// Сюрприз! Новые значения в обоих массивах.
// Старые значения утеряны!
trace(letters);         // На экран выводится: "d,e,f"
trace(newLetters);      // На экран выводится: "d,e,f" (не "a,b,c")
```

Взаимосвязь двух копий массива – это хорошо или плохо? Ответ зависит от того, что ожидается и чего требуется достичь. Сначала разберемся, что происходит, а потом научимся выходить из ситуации.

В следующей строке предыдущего примера содержимое letters не копируется, как это происходило бы, если бы в letters хранились элементарные данные:

```
var newLetters:Array = letters;
```

Вместо этого данная строка указывает Flash: «Создать указатель newLetters, указывающий на то, на что указывает letters, даже если содержимое в будущем будет изменяться». Таким образом, обе переменные, letters и newLetters, всегда указывают на одни и те же данные, хранящиеся в памяти. Если это поможет, можно провести аналогию с *ярлыком* файла в Windows (*псевдоним* в Mac OS). Ярлык просто ука-

зывает на файл, размещенный где-то в памяти. Независимо от того, открывается ли файл напрямую или посредством ярлыка, физически существует только один файл с интересующим содержимым. Если файл меняется, ярлык по-прежнему предлагает доступ к текущему содержимому файла. Если требуются два независимых файла, необходимо дублировать оригинальный файл, а не создавать его ярлык.

Итак, хорошо ли это, если две переменные ссылаются на один и тот же массив? Как описывалось ранее, при нормальном ходе вещей это повышает производительность, предотвращая ненужное копирование содержимого массива. Однако может возникнуть необходимость работать с копией массива без воздействия на оригинал. В этом случае, используя метод *concat( )*, можно создать дубликат массива:

```
// Задаем элементы массива.
var letters:Array = ["a", "b", "c"];

// Создаем независимую копию letters с помощью
// метода concat(), возвращающего новый массив.
var newLetters:Array = letters.concat();

// Оба массива содержат одинаковые значения, как ожидалось.
trace(letters);        // На экран выводится: "a,b,c"
trace(newLetters);     // На экран выводится: "a,b,c"

// Меняем значение letters.
letters = ["d", "e", "f"];

// В отличие от предыдущих примеров, массивы независимы.
trace(letters);        // На экран выводится: "d,e,f"
trace(newLetters);     // На экран выводится: "a,b,c"
```

В строке 6 предыдущего примера вместо метода *concat( )* можно было бы использовать метод *slice( )*:

```
var newLetters:Array = letters.slice(0);
```

Методы *concat( )* или *slice( )* эффективны при создании дубликатов одномерных массивов с целочисленными индексами. Однако все не так хорошо в случае с *многомерным* (массив, содержащий другие массивы) или ассоциативным массивом. (Более подробно о многомерных и ассоциативных массивах см. в рецептах 5.9 и 5.15 соответственно.) С ассоциативными массивами методы *concat( )* или *slice( )* не применяются. А с многомерными массивами при использовании методов *concat( )* или *slice( )* дублируется только массив верхнего уровня, а данные вложенных массивов не дублируются. Следующий фрагмент кода иллюстрирует это:

```
var coordinates:Array = new Array();
coordinates.push([0,1,2,3]);
coordinates.push([4,5,6,7]);
coordinates.push([8,9,10,11]);
coordinates.push([12,13,14,15]);
```

```
// Делаем дубликат.
var coordinatesDuplicate:Array = coordinates.concat();

// Заменяем один из элементов одного из вложенных массивов в дубликате.
coordinatesDuplicate[0][0] = 20;
trace(coordinates[0][0]);  // На экран выводится: 20

// Заменяем один из элементов верхнего уровня.
coordinatesDuplicate[1] = [21,22,23,24];
trace(coordinates[1]);     // На экран выводится: 4,5,6,7
```

В предыдущем коде `coordinates` (координаты) – это массив массивов. Такие массивы еще называют *двумерными массивами*. `coordinatesDuplicate` – дубликат `coordinates`. Однако, даже несмотря на то, что это – дубликат, его элементы (тоже являющиеся массивами) по-прежнему остаются ссылками на оригинальные элементы, а не их копиями. Это означает, что если в одном из вложенных в `coordinatesDuplicate` массивов одному из элементов присвоить новое значение, аналогично изменится `coordinates`. Но последние две строки кода подтверждают, что `coordinatesDuplicate` на самом деле дублирует элементы верхнего уровня: замена одного из элементов не приводит к изменению `coordinates`.

Чтобы гарантированно продублировать весь массив, включая все вложенные элементы, необходимо использовать *рекурсию*. Метод *ArrayUtilities.duplicate( )* как раз этим и занимается, упрощая задачу по рекурсивному дублированию массивов. Методу *duplicate( )* необходим всего один параметр, ссылка на массив или ассоциативный массив, и он возвращает дубликат данного объекта. Однако по умолчанию *duplicate( )* возвращает копию только элементов верхнего уровня, аналогично *concat( )* или *slice( )*. Чтобы дублировать экземпляр рекурсивно, необходимо задать Булево значение `true` в качестве второго параметра, как показано в следующем примере:

```
// Создаем двумерный массив.
var coordinates:Array = new Array();
for(var i:int = 0; i < 4; i++) {
  coordinates[i] = new Array();
  for(var j:int = 0; j < 4; j++) {
    coordinates[i].push(String(i) + "," + String(j));
  }
}

// Копируем весь массив координат.
var newCoordinates:Array = ArrayUtilities.duplicate(coordinates, true)
                           as Array;

// Заменяем элемент во вложенном массиве.
newCoordinates[0][0] = "a";

// Используем метод toString() класса ArrayUtilities
// для быстрого вывода содержимого массивов.
trace(ArrayUtilities.toString(coordinates));
trace(ArrayUtilities.toString(newCoordinates));
```

Следующий пример иллюстрирует применение того же метода *duplicate( )* с ассоциативным массивом:

```
var coordinatesMap:Object = new Object();
coordinatesMap.a = [{a: 1},{b: 2}, {c: 3}, {d: 4}];
coordinatesMap.b = [{a: 1},{b: 2}, {c: 3}, {d: 4}];
coordinatesMap.c = [{a: 1},{b: 2}, {c: 3}, {d: 4}];
coordinatesMap.d = [{a: 1},{b: 2}, {c: 3}, {d: 4}];
var newCoordinatesMap:Object = ArrayUtilities.duplicate(coordinatesMap, true);
newCoordinatesMap.a[0] = {r: 5};
trace(ArrayUtilities.toString(coordinatesMap));
trace(ArrayUtilities.toString(newCoordinatesMap));
```

В обоих примерах изменения дубликатов не оказывают никакого влияния на исходный массив (или ассоциативный массив).

См. также

Рецепты 5.9 и 5.15.

5.9. Хранение сложных или многомерных данных

Задача

Имеется два или более наборов взаимосвязанных данных и требуется отслеживать отношения между их элементами.

Решение

Используйте *параллельные массивы*, массив массивов (*многомерный массив*) или *массив объектов*.

Обсуждение

Можно создать два или более параллельных массива, в которых элементы с одинаковыми индексами взаимосвязаны. Например, метод *beginGradientFill( )* (начать градиентное заполнение), обсуждаемый в главе 7, использует три параллельных массива: массив цветов, массив прозрачности соответствующих цветов и массив коэффициентов распределения цветов, используемых в градиенте. В каждом массиве элементы с одинаковыми индексами связаны друг с другом.

Для создания параллельных массивов взаимосвязанные элементы располагаются в разных массивах под одинаковыми индексами. Использование параллельных массивов позволяет без труда извлекать связанные данные, поскольку их индексы во всех массивах одинаковые; например:

```
var colors:Array = ["maroon", "beige",    "blue",     "gray"];
```

```
var years:Array  = [1997,     2000,       1985,       1983];
var makes:Array  = ["Honda",  "Chrysler", "Mercedes", "Fiat"];

// Перебираем элементы массивов. Поскольку длина всех
// массивов одинаковая, в выражении for может
// использоваться свойство length любого из них.
// В данном случае используется makes.length.
for (var i:int = 0; i < makes.length; i++) {
    // На экран выводится:
    // A maroon 1997 Honda
    // A beige 2000 Chrysler
    // A blue 1985 Mercedes
    // A gray 1983 Fiat

    // Выводим на экран элементы массивов
    // с соответствующими индексами.
    trace("A " + colors[i] + " " +
                 years[i] + " " +
                 makes[i]);
}
```

При работе с параллельными массивами необходимо быть крайне аккуратными. Удаляя или добавляя элементы одного массива, нельзя забывать об изменении всех остальных массивов. В противном случае произойдет рассинхронизация, и массивы станут бесполезными.

Другой вариант работы с несколькими наборами данных – создание многомерного массива, т.е. массива массивов (массива, каждый элемент которого является массивом):

```
// Создаем массив, cars, и заполняем его элементами,
// которые сами являются массивами. Каждый массив-элемент
// представляет машину и содержит три элемента
// (цвет, год и производитель).
var cars:Array = new Array();
cars.push(["maroon", 1997, "Honda"]);
cars.push(["beige", 2000, "Chrysler"]);
cars.push(["blue", 1985, "Mercedes"]);
cars.push(["gray", 1983, "Fiat"]);

// Перебираем элементы массива cars.
for (var i:int = 0; i < cars.length; i++) {
    // Результат аналогичен результату,
    // полученному в предыдущем примере
    // с использованием параллельных массивов:
    // A maroon 1997 Honda
    // A beige 2000 Chrysler
    // A blue 1985 Mercedes
    // A gray 1983 Fiat

    // Выводим все элементы каждого подмассива cars[i].
    // Обратите внимание, что используется два индекса
```

```
    // последовательно, например cars[i][0].
    trace("A " + cars[i][0] + " " +
                 cars[i][1] + " " +
                 cars[i][2]);
}
```

Далее представлен другой способ просмотра содержимого двумерного массива `cars` (автомобили). В этом случае выводится длинный список элементов (форматирование не такое красивое, как в предыдущем примере, но более четко видна структура массива):

```
// Перебираем элементы массива cars.
for (var i:int = 0; i < cars.length; i++) {
    // Перебираем элементы каждого подмассива, cars[i].
    for (var j:int = 0; j < cars[i].length; j++) {
        // Обратите внимание на использование
        // двух индексов последовательно, cars[i][j].
        trace("Элемент [" + i + "][" + j + "] содержит: "
              + cars[i][j]);
    }
}
```

В предыдущем примере (массив массивов) значение выражений `cars[i][0]` или `cars[i][j]` малопонятно. Более того, при изменении порядка элементов подмассива приходится менять код (или будет ошибочно выведено «A Honda maroon 1997» вместо «A maroon 1997 Honda»).

Один из вариантов — использовать массив объектов (ассоциативные массивы). Эта методика подобна работе с массивом массивов, но предлагает преимущество именованных свойств. В массиве массивов каждое значение определяется его числовым индексом. Однако, когда используется массив объектов, ссылаться на данные можно по имени свойства, а не по номеру индекса. Задавать свойства объекта можно в любом порядке, поскольку затем они вызываются по имени, не по номеру:

```
// Создаем массив, cars, и заполняем его объектами.
// У каждого объекта есть свойство make, свойство year
// и свойство color.
var cars:Array = new Array();

// Здесь для определения трех свойств каждой машины
// используются объектные литералы; объектные литералы
// добавляются в основной массив.
cars.push({make: "Honda",    year: 1997, color: "maroon"});
cars.push({make: "Chrysler", year: 2000, color: "beige"});
cars.push({make: "Mercedes", year: 1985, color: "blue"});
cars.push({make: "Fiat",     year: 1983, color: "gray"});

// Перебираем элементы массива cars.
for (var i:int = 0; i < cars.length; i++) {
    // Результат аналогичен предыдущим примерам, но обращение
    // к каждому значению осуществляется по имени свойства,
    // что удобнее и понятнее для программиста.
```

```
        trace("A " + cars[i].color + " " +
                     cars[i].year + " " +
                     cars[i].make);
    }
```

См. также

В рецепте 5.15 рассматриваются ассоциативные массивы, доступ к элементам которых осуществляется по имени, а не по номеру.

5.10. Сортировка или обращение массива

Задача

Требуется сортировать элементы массива.

Решение

Это делается при помощи метода *sort( )*. Для массивов объектов годится и метод *sortOn( )*.

Обсуждение

Простую сортировку массива можно осуществить с помощью метода *sort( )*. Без параметров этот метод сортирует элементы массива в порядке по возрастанию. Элементы сортируются соответственно таблице символов Unicode (строго по алфавиту для западноевропейских языков).

```
var words:Array = ["tricycle", "relative", "aardvark", "jargon"];
words.sort();
trace(words); // На экран выводится: aardvark,jargon,relative,tricycle
```

Метод *sort( )*, в его форме по умолчанию, очень полезен, если требуется сортировать элементы массива в возрастающем, алфавитном порядке. Однако есть несколько оговорок. А именно, при сортировке учитывается регистр и числа располагаются в «алфавитном» порядке, а не цифровом. К счастью, ActionScript обеспечивает возможность конкретизировать способ сортировки, передавая в метод *sort( )* одну из зарезервированных констант.

Сортировка массива в порядке по убыванию осуществляется через использование константы *Array.DESCENDING*:

```
var words:Array = ["tricycle", "relative", "aardvark", "jargon"];
words.sort(Array.DESCENDING);
trace(words); // На экран выводится: tricycle,relative,jargon,aardvark
```

Как говорилось ранее, метод *sort( )* выполняет сортировку с учетом регистра. Он помещает элементы, начинающиеся с большой буквы, перед элементами, написанными строчными символами. Это иллюстрирует следующий фрагмент кода:

```
var words:Array = ["Tricycle", "relative", "aardvark", "jargon"];
words.sort();
trace(words); // На экран выводится: Tricycle,aardvark,jargon,relative
```

Для проведения сортировки без учета регистра можно использовать константу *Array.CASEINSENSITIVE*:

```
var words:Array = ["Tricycle", "relative", "aardvark", "jargon"];
words.sort(Array.CASEINSENSITIVE);
trace(words); // На экран выводится: aardvark,jargon,relative,Tricycle
```

При сортировке массива чисел значения располагаются соответственно ASCII-эквивалентам цифр, а не в цифровой последовательности. Это иллюстрирует следующий фрагмент кода:

```
var scores:Array = [10, 2, 14, 5, 8, 20, 19, 6];
scores.sort();
trace(scores);   // На экран выводится: 10,14,19,2,20,5,6,8
```

Чтобы сортировать массив в цифровой последовательности, в методе *sort( )* используется константа *Array.NUMERIC*:

```
var scores:Array = [10, 2, 14, 5, 8, 20, 19, 6];
scores.sort(Array.NUMERIC);
trace(scores);   // На экран выводится: 2,5,6,8,10,14,19,20
```

Есть еще две константы, которые могут использоваться в методе *sort( )*: *Array.UNIQUESORT* и *Array.RETURNINDEXEDARRAY*. В некоторых случаях требуется сортировать массив, только если он содержит уникальные элементы. Тогда используется константа *Array.UNIQUESORT*, и Flash сортирует массив, только если все его элементы уникальны. В противном случае метод *sort( )* возвращает 0, и массив не сортируется:

```
var ranking:Array = [2,5,6,3,1,1,4,8,7,10,9];
var sortedRanking:Object = ranking.sort(Array.UNIQUESORT);
trace(sortedRanking);   // На экран выводится: 0
trace(ranking);  // На экран выводится: 2,5,6,3,1,1,4,8,7,10,9
```

Часто требуется сортировать элементы массива, но при этом сохранить и исходный массив, потому что другие части приложения зависят от существующего порядка. Например, если есть параллельные массивы и сортировке подвергается один из них, его отношения с другими массивами становятся недействительными. В подобных сценариях очень полезна константа *Array.RETURNINDEXEDARRAY*. Она позволяет возвращать новый массив, содержащий индексы элементов исходного массива в сортированном порядке, как иллюстрирует следующий фрагмент кода:

```
var words:Array = ["tricycle", "relative", "aardvark", "jargon"];
var indices:Array = words.sort(Array.RETURNINDEXEDARRAY);
trace(words);   // На экран выводится: tricycle,relative,aardvark,jargon
trace(indices); // На экран выводится: 2,3,1,0
```

```
for(var i:int = 0; i < words.length; i++) {
  /* На экран выводится:
     aardvark
     jargon
     relative
     tricycle
  */
  trace(words[indices[i]]);
}
```

Выбор вида сортировки не ограничен использованием всего одного модификатора. Константы можно комбинировать, используя побитовую операцию *ИЛИ* (|). Следующий фрагмент кода иллюстрирует сортировку без учета регистра в порядке убывания:

```
var words:Array = ["Tricycle", "relative", "aardvark", "jargon"];
words.sort(Array.CASEINSENSITIVE | Array.DESCENDING);
trace(words);   // На экран выводится: Tricycle,relative,jargon,aardvark
```

Иногда требуется изменить порядок элементов массива на обратный. Метод *sort( )* обеспечивает возможность выполнять сортировки по убыванию, по возрастанию, с учетом регистра, без учета регистра и в цифровой последовательности, но не позволяет просто расположить элементы в обратном порядке. Для этого используется метод *reverse( )*. Он делает именно то, что предполагает его имя: изменяет порядок элементов на обратный:

```
var words:Array = ["tricycle", "relative", "aardvark", "jargon"];
words.reverse();
trace(words);   // На экран выводится: jargon,aardvark,relative,tricycle
```

Предыдущая часть рецепта описывала, как сортировать массивы, элементами которых являются строки или числа. Но с помощью метода *sortOn( )* можно сортировать массивы объектов любого типа. В метод *sortOn( )* необходимо передать строковый параметр, определяющий имя свойства, по которому будет проводиться сортировка элементов:

```
var cars:Array = new Array();
cars.push({make: "Honda",    year: 1997, color: "maroon"});
cars.push({make: "Chrysler", year: 2000, color: "beige"});
cars.push({make: "Mercedes", year: 1985, color: "blue"});
cars.push({make: "Fiat",     year: 1983, color: "gray"});
// Сортируем массив cars по свойству year
// каждого элемента element.
cars.sortOn("year");
for (var i:int = 0; i < cars.length; i++) {
  /* На экран выводится:
     gray    1983  Fiat
     blue    1985  Mercedes
     maroon  1997  Honda
     beige   2000  Chrysler
  */
```

```
  trace(cars[i].color + "\t" +
        cars[i].year + "\t" +
        cars[i].make);
}
```

Метод *sortOn( )* также может проводить сортировку по нескольким полям. Для этого необходимо задать массив полей сортировки. Тогда элементы сортируются по этим полям в заданном порядке. Чтобы понять, как это происходит, рассмотрим следующий пример:

```
var cars:Array = new Array();
cars.push({make: "Honda",    year: 1997, color: "maroon"});
cars.push({make: "Chrysler", year: 2000, color: "beige"});
cars.push({make: "Mercedes", year: 1985, color: "blue"});
cars.push({make: "Fiat",     year: 1983, color: "gray"});
cars.push({make: "Honda",    year: 1992, color: "silver"});
cars.push({make: "Chrysler", year: 1968, color: "gold"});
cars.push({make: "Mercedes", year: 1975, color: "green"});
cars.push({make: "Fiat",     year: 1983, color: "black"});
cars.push({make: "Honda",    year: 2001, color: "blue"});
cars.push({make: "Chrysler", year: 2004, color: "orange"});
cars.push({make: "Mercedes", year: 2000, color: "white"});
cars.push({make: "Fiat",     year: 1975, color: "yellow"});

// Сортируем элементы массива cars сначала
// по свойству year, а затем по свойству make.
cars.sortOn(["year", "make"]);

for (var i:int = 0; i < cars.length; i++) {
  /* На экран выводится:
     gold     1968    Chrysler
     yellow   1975    Fiat
     green    1975    Mercedes
     black    1983    Fiat
     gray     1983    Fiat
     blue     1985    Mercedes
     silver   1992    Honda
     maroon   1997    Honda
     beige    2000    Chrysler
     white    2000    Mercedes
     blue     2001    Honda
     orange   2004    Chrysler
  */
  trace(cars[i].color + "\t" +
        cars[i].year + "\t" +
        cars[i].make);
}
```

В следующем примере тот же массив сортируется сначала по производителю, а затем по году выпуска (обратите внимание на результат):

```
cars.sortOn(["make", "year"]);
```

```
for (var i:int = 0; i < cars.length; i++) {
  /* На экран выводится:
     gold    1968    Chrysler
     beige   2000    Chrysler
     orange  2004    Chrysler
     yellow  1975    Fiat
     black   1983    Fiat
     gray    1983    Fiat
     silver  1992    Honda
     maroon  1997    Honda
     blue    2001    Honda
     green   1975    Mercedes
     blue    1985    Mercedes
     white   2000    Mercedes
  */
  trace(cars[i].color + "\t" +
        cars[i].year + "\t" +
        cars[i].make);
}
```

Как и *sort( )*, метод *sortOn( )* поддерживает модификаторы сортировки. Константы класса *Array* могут использоваться для сортировки по убыванию, без учета регистра и в цифровой последовательности. Как и метод *sort( )*, *sortOn( )* может выполнять сортировку уникальных элементов и возвращать массив сортированных индексов без изменения исходного массива. В следующем примере массив `cars` сортируется в порядке по убыванию:

```
cars.sortOn("year", Array.DESCENDING);

for (var i:int = 0; i < cars.length; i++) {
  /* На экран выводится:
     beige   2000  Chrysler
     maroon  1997  Honda
     blue    1985  Mercedes
     gray    1983  Fiat
  */
  trace(cars[i].color + "\t" +
        cars[i].year + "\t" +
        cars[i].make);
}
```

Сортированные массивы используются во многих сценариях. Например, если требуется вывести элементы массива в компонент пользовательского интерфейса или текстовое поле, предпочтительнее представлять их в алфавитном порядке.

Во всех случаях, кроме применения константы *Array.RETURNINDEXEDARRAY*, методы *sort( )* и *sortOn( )* изменяют порядок элементов исходного массива; они не возвращают новый массив.

См. также

Рецепт 5.8, чтобы создать отдельную копию массива, с которой можно делать все, что угодно. В рецепте 5.11 подробно описывается специальная сортировка.

5.11. Реализация специальной сортировки

Задача

Требуется сортировать массив, используя более сложную логику, чем сортировка в алфавитном порядке или цифровой последовательности.

Решение

Используйте метод *sort( )* и передайте в него ссылку на *функцию сравнения*.

Обсуждение

Если требуется провести управляемую сортировку, применяется метод *sort( )* в сочетании со специальной *функцией сравнения* (ее еще называют *функцией сортировки*). Метод *sort( )* многократно вызывает функцию сравнения. При каждом вызове можно поменять местами два элемента массива. Метод передает в функцию сравнения два параметра (назовем их *a* и *b*). Функция сравнения определяет порядок расположения элементов и в зависимости от этого возвращает положительное число, отрицательное число или 0. Если функция возвращает отрицательное значение, *a* должен стоять перед *b*. Если возвращается 0, сохраняется текущее расположение. Если возвращается положительное значение, *a* располагается после *b*. Метод *sort( )* вызывает функцию сравнения для каждого рассматриваемого сочетания элементов до тех пор, пока весь массив не будет упорядочен соответствующим образом. Использовать специальную функцию сравнения намного проще, чем это может показаться. Не надо вдаваться в детали сортировки всего массива, требуется просто задать критерий для сравнения любых двух его элементов.

Один из примеров, когда следует прибегнуть к специальной сортировке: необходимо сортировать список строк, но перед этим строки должны быть некоторым образом обработаны. Скажем, создается программа для воспроизведения музыки, которая должна отображать список групп. Если проводить сортировку просто в алфавитном порядке, все названия групп, начинающиеся с «The», появятся в разделе на букву «T», а это, вероятно, не совсем то, что требуется. Можно определить функцию сравнения, которая будет отбрасывать начальный «The» при сравнении названий. Вот код, в котором задается массив, выполняется простая сортировка и выводятся результаты:

```
var bands:Array = ["The Clash",
```

```
                    "The Who",
                    "Led Zeppelin",
                    "The Beatles",
                    "Aerosmith",
                    "Cream"];
bands.sort();
for(var i:int = 0; i < bands.length; i++) {
    trace(bands[i]);

    /* вывод:
       Aerosmith
       Cream
       Led Zeppelin
       The Beatles
       The Clash
       The Who
    */
}
```

Чтобы выйти из этой ситуации, вызываем метод *sort( )*, передавая в него функцию сравнения *bandNameSort*:

```
var bands:Array = ["The Clash",
                    "The Who",
                    "Led Zeppelin",
                    "The Beatles",
                    "Aerosmith",
                    "Cream"];
bands.sort(bandNameSort);
for(var i:int = 0; i < bands.length; i++) {
    trace(bands[i]);
    /* вывод:
       Aerosmith
       The Beatles
       The Clash
       Cream
       Led Zeppelin
       The Who
    */
}

function bandNameSort(band1:String, band2:String):int
{
    band1 = band1.toLowerCase();
    band2 = band2.toLowerCase();
    if(band1.substr(0, 4) == "the ") {
        band1 = band1.substr(4);
    }
    if(band2.substr(0, 4) == "the ") {
        band2 = band2.substr(4);
    }
    if(band1 < band2) {
```

```
            return -1;
        }
        else {
            return 1;
        }
    }
```

Функция *bandNameSort( )* сначала переводит все названия в нижний регистр, обеспечивая сортировку без учета регистра. Затем проверяет на наличие «The» в начале названия. Если находит «The», берет часть строки, начиная с четвертого символа до конца, т.е. все после слова «The» и пробела.

Наконец, сравниваются две обрабатываемые строки. Если первая строка должна стоять первой, возвращается `-1`, и если первая строка должна быть второй, возвращается `1`. Как видите, результат больше соответствует ожиданиям.

Функция сравнения может быть сколь угодно сложной. Если сортируется список объектов, можно встроить логику, которая будет считывать несколько свойств объектов, осуществлять вычисления по их данным, проводить сравнение и возвращать результаты.

Представьте, что функция сравнения может вызываться сотни или даже тысячи раз при сортировке большого массива, поэтому не рекомендуется делать ее слишком сложной.

5.12. Расположение элементов массива в случайном порядке

Задача

Требуется расположить элементы массива в случайном порядке.

Решение

Используйте метод *sort( )* с функцией сравнения, которая случайным образом возвращает положительное или отрицательное число.

Обсуждение

Существует множество сценариев, в которых может потребоваться расположить элементы массива в случайном порядке. Например, моделируется игра, в которой надо перемешать буквы слова. Мы уже умеем разбирать строку в массив букв с помощью метода *split( )*, осталось только расположить элементы массива в случайном порядке. Это может также пригодиться при моделировании игры в карты, чтобы перетасовать колоду.

Эту задачу можно решить многими способами. Однако один из самых простых – создать функцию сравнения, которая случайным образом

возвращает положительные или отрицательные числа, и применить ее в методе *sort( )*. Более подробно о функциях сравнения рассказывается в рецепте 5.11.

Вот, вероятно, самая простая из возможных в данном случае функций сравнения:

```
function randomSort(elementA:Object, elementB:Object):Number {
    return Math.random() - .5
}
```

Math.random( ) возвращает число в диапазоне от 0,0 до 1,0. Если из возвращаемого результата вычитать 0,5, получаем диапазон случайных чисел от -0,5 до 0,5. Вспомним, что в функции сравнения возвращение отрицательного числа означает, что первый элемент должен стоять первым, а положительный результат указывает методу *sort( )* поставить первым второй элемент. Поскольку вероятность обоих исходов составляет 50/50, в итоге получаем совершенно случайный порядок расположения элементов массива.

Вот пример расположения элементов массива в случайном порядке:

```
var numbers:Array = new Array();
for(var i:int=0;i<20;i++) {
    numbers[i] = i;
}
numbers.sort(randomSort);
for(var i:int=0;i<numbers.length;i++) {
    trace(numbers[i]);
}
```

Тем самым создается массив из 20 последовательных чисел, которые затем перемешиваются и выводятся на экран. Можно проверить и убедиться, что теперь числа располагаются абсолютно случайным образом.

См. также

Более подробно о функциях сравнения смотрите в рецепте 5.11.

5.13. Получение наименьшего и наибольшего элементов

Задача

Из массива чисел требуется извлечь наименьший или наибольший элемент.

Решение

Сортируйте массив в цифровой последовательности и затем выберите первый или последний элемент сортированного массива.

Обсуждение

Быстро извлечь наименьшее или наибольшее значение массива можно путем его сортировки. Следующий пример иллюстрирует, как это сделать:

```
var scores:Array = [10, 4, 15, 8];
scores.sort(Array.NUMERIC);
trace("Наименьший: " + scores[0]);
trace("Наибольший: " + scores[scores.length - 1]);
```

Конечно, если важен существующий порядок расположения элементов массива, перед сортировкой необходимо сделать копию массива. См. рецепт 5.8.

Здесь можно применить методы *ArrayUtilities.min( )* и *ArrayUtilities.max( )* на выбор.

См. также

Рецепт 5.8.

5.14. Сравнение массивов

Задача

Для проверки эквивалентности требуется сравнить два массива.

Решение

Последовательно переберите и сравните все элементы обоих массивов.

Обсуждение

Массивы – это ссылочный тип данных. Поэтому, применяя оператор равенства к двум переменным массива, можно проверить только, указывают ли они на одну и ту же область памяти. Например:

```
var letters:Array = ["a", "b", "c", "d"];
var lettersPointer:Array = letters;
trace(letters == lettersPointer);  // На экран выводится: true
```

Однако если два массива эквивалентны, но занимают разные области памяти, оператор равенства возвращает `false`:

```
var letters1:Array = ["a", "b", "c", "d"];
var letters2:Array = ["a", "b", "c", "d"];
trace(letters1 == letters2];  // На экран выводится: false
```

Поэтому, чтобы сравнить два массива, необходимо перебрать и попарно сравнить все их элементы:

```
var equivalent:Boolean = true;
for(var i:int = 0; i < letters1.length; i++) {
```

```
        if(letters1[i] != letters2[i]) {
            equivalent = false;
            break;
        }
    }
    trace(equivalent);  // На экран выводится: true
```

Можно воспользоваться методом *ArrayUtilities.equals( )*. В него передаются два обязательных параметра: ссылки на два сравниваемых массива. Метод возвращает Булево значение, свидетельствующее о том, эквивалентны массивы или нет:

```
var letters1:Array = ["a", "b", "c", "d"];
var letters2:Array = ["a", "b", "c", "d"];
trace(ArrayUtilities.equals(letters1, letters2));
// На экран выводится: true
```

По умолчанию порядок элементов в обоих массивах должен совпадать. Если совпадение порядка расположения не имеет значения, методу *equals( )* можно задать третий параметр: Булево значение, указывающее на то, должен ли учитываться порядок:

```
var letters1:Array = ["a", "b", "c", "d"];
var letters2:Array = ["b", "a", "d", "c"];
trace(ArrayUtilities.equals(letters1, letters2));
// На экран выводится: false
trace(ArrayUtilities.equals(letters1, letters2, true));
// На экран выводится: true
```

Метод *equals( )* достаточно прост. В следующем фрагменте кода пояснения приведены в комментариях:

```
public static function equals(arrayA:Array,
                              arrayB:Array,
                              bNotOrdered:Boolean):Boolean {

    // Если число элементов массивов разное, очевидно, они не эквивалентны.
    if(arrayA.length != arrayB.length) {
        return false;
    }

    // Создаем копии каждого массива, чтобы
    // оригиналы оставались неизменными.
    var arrayACopy:Array = arrayA.concat();
    var arrayBCopy:Array = arrayB.concat();

    // Если порядок элементов массивов не имеет значения,
    // сортируем обе копии так, чтобы при сравнении
    // порядок их элементов совпадал.
    if(bNotOrdered) {
        arrayACopy.sort();
        arrayBCopy.sort();
    }

    // Перебираем и сравниваем все элементы массивов.
```

```
        // Если они не совпадают, удаляем копии и возвращаем false.
        for(var i:int = 0; i < arrayACopy.length; i++) {
            if(arrayACopy[i] != arrayBCopy[i]) {
                delete arrayACopy;
                delete arrayBCopy;
                return false;
            }
        }

        // В противном случае массивы эквивалентны.
        // Поэтому удаляем копии и возвращаем true.
        delete arrayACopy;
        delete arrayBCopy;
        return true;
    }
```

5.15. Создание ассоциативного массива

Задача

Требуется создать массив, использующий вместо числовых индексов именованные элементы.

Решение

Создайте ассоциативный массив.

Обсуждение

При работе с наборами данных, в которых каждый элемент имеет особое значение или важность, обычные массивы с числовыми индексами не всегда подходят.

Например, для такого набора данных, как имена членов комитета, вполне подойдет обычный массив с числовыми индексами:

```
var aMembers:Array = new Array("Franklin", "Gina", "Sindhu");
```

Однако если каждый член комитета играет особую роль, средств стандартного массива недостаточно, чтобы обозначить это. В таком случае можно использовать *ассоциативный массив*. В некоторых языках программирования его еще называют *хеш-таблицей*. В ActionScript, на самом деле, он является просто экземпляром класса *Object*. Ассоциативный массив вместо числовых индексов использует именованные элементы. Имена, используемые для обращения к элементам массива, часто называют *ключами* или *свойствами*. Ключи обеспечивают осмысленный контекст значению ассоциированного элемента.

В ActionScript ассоциативный массив можно создать, используя *нотацию объектных литералов* или добавляя элементы в объект. Несмотря на свое имя, класс *Array* не принимает участия в создании ассоциативных массивов. Его методы и свойства предназначены для работы

исключительно с массивами с числовыми индексами. Ассоциативные массивы должны быть экземплярами класса *Object*. Формально, поскольку класс *Object* является базовым классом всех классов ActionScript, все объекты ActionScript могут использоваться как ассоциативные массивы. Однако если нет особых причин использовать другой класс, лучше просто обратиться к универсальному классу *Object*.

Один из способов создания ассоциативного массива – использование нотации объектных литералов. В этом случае разделяемый запятыми список ключей и значений, записываемых через двоеточие (:), заключается в фигурные скобки({ }), как показано в следующем примере:

```
var memebers:Object = {scribe: "Franklin",
                       chairperson: "Gina",
                       treasurer: "Sindhu"};
```

Ассоциативный массив также можно создать, используя следующую многостроковую технику с конструктором *Object*. Нотация объектных литералов хороша для создания маленьких ассоциативных массивов за один шаг, но чтобы организовывать большие массивы, необходимо обратиться к методике с использованием конструктора *Object*. В этом случае код становится более удобным для чтения и свойства в ассоциативный массив можно добавлять, задавая свойства (ключи) в последующих строках. Например:

```
var members:Object = new Object();
members.scribe = "Franklin";
members.chairperson = "Gina";
members.treasurer = "Sindhu";
```

Хотя чаще используется конструктор *Object*, инициализировать ассоциативный массив можно с помощью пустого объектного литерала:

```
var members:Object = {};
```

Извлечь значения из ассоциативного массива можно двумя способами. Первый – через нотацию свойства (используя оператор «точка»):

```
trace(members.scribe); // На экран выводится: Franklin
```

Второй вариант извлечения значений из ассоциативного массива – использование *нотации доступа к массиву*. При этом за ссылкой на ассоциативный массив следует оператор доступа к массиву ([ ]). В операторе доступа к массиву используется *строковое* значение имени ключа, к которому организуется доступ:

```
trace(members["scribe"]); // На экран выводится: Franklin
```

Нотация доступа к массиву исключительно полезна в тех ситуациях, когда имеется множество ключей с именами, образующими последовательность. В этом случае можно динамически генерировать строковое значение ключа, тогда как в нотации свойства это невозможно, например:

```
var members:Object = new Object();
members.councilperson1 = "Beatrice";
members.councilperson2 = "Danny";
members.councilperson3 = "Vladamir";

for (var i:int = 1; i <= 3; i++) {
    trace(members["councilperson" + i];
}
```

Нотация доступа к массиву чаще всего используется при переборе всех элементов ассоциативного массива, как показано в рецепте 5.16.

Для чтения или записи значений ассоциативного массива может использоваться как нотация свойства, так и нотация доступа к массиву:

```
var members:Object = new Object();
members["councilperson"] = "Ruthie";
trace(members.councilperson);          // На экран выводится: Ruthie
members.councilperson = "Rebecca";
trace(members["councilperson"]);       // На экран выводится: Rebecca
```

См. также

В рецепте 5.16 более подробно рассказывается об организации доступа к именованным элементам ассоциативного массива.

5.16. Чтение элементов ассоциативного массива

Задача

Требуется перебрать элементы ассоциативного массива.

Решение

Обратитесь к выражению *for ... in*.

Обсуждение

Перебор элементов массивов с целочисленными индексами осуществляется с помощью выражения *for*. Однако к именованным элементам ассоциативных массивов невозможно обратиться через числовой индекс. Кроме того, элементы в ассоциативном массиве располагаются в произвольном порядке независимо от очередности их добавления в массив. Поэтому нет методов для сортировки, переворачивания или другого изменения порядка расположения элементов ассоциативных массивов.

К счастью, перебрать перечислимые элементы ассоциативного массива можно с помощью выражения *for...in*. Это выражение проходит по всем доступным для чтения свойствам заданного объекта. Синтаксис *for...in* следующий:

```
for (ключ in объект) {
    // Действия
}
```

Выражение *for...in* не требует явного обновляющего выражения, потому что число итераций определяется числом свойств рассматриваемого объекта. Обратите внимание, что *ключ* – это имя переменной, которая будет использоваться для хранения имени свойства при каждой итерации, а не имя конкретного свойства или ключа. С другой стороны, *объект* – это конкретный объект, свойства которого требуется читать. Например:

```
var members:Object = new Object();
members.scribe = "Franklin";
members.chairperson = "Gina";
members.treasurer = "Sindhu";

// Используем выражение for . . . in,
// чтобы перебрать все элементы.
for (var sRole:String in members) {
    // На экран выводится:
    // treasurer: Sindhu
    // chairperson: Gina
    // scribe: Franklin
    trace(sRole + ": " + members[sRole]);
}
```

С выражением *for...in* должна использоваться нотация доступа к массиву (квадратные скобки). Нотация свойства (с оператором «точка») не будет работать корректно, потому что значение, присваиваемое переменной `sRole`, – это *строковое* имя ключа, а не его идентификатор.

Цикл *for...in* не выводит все встроенные свойства объекта. Например, он выбирает специальные свойства, добавленные во время выполнения, но не перечисляет методы встроенных объектов, хотя они хранятся в свойствах объектов.

См. также

Рецепт 5.2.

6

Список отображения

6.0. Введение

Модель рендеринга в ActionScript 3.0 и Flash Player 9 радикально отличается от реализованной в предыдущих версиях. Традиционно центральным объектом рендерера был класс *MovieClip*. В каждом ролике *.swf* имелся корневой объект *MovieClip* (обычно называемый *Сценой*). У него могли быть дочерние объекты *MovieClip*, которые, в свою очередь, тоже могли содержать дочерние объекты *MovieClip*. Управление порядком, в котором отрисовывались *клипы*[1], строилось на понятии г*лубины* (чем меньше значение глубины, тем «глубже» располагаются объекты). Экземпляры *MovieClip* создавались методами *createEmptyMovieClip( )* (создать пустой *MovieClip*), *attachMovie( )* (присоединить *MovieClip*) или *duplicateMovieClip( )* (дублировать *MovieClip*). При создании *клип* автоматически добавлялся в визуальную иерархию и вследствие этого отрисовывался рендерером. Клипы не могли менять свое положение в иерархии; для этого его надо было сначала уничтожить и затем создать вновь в новой точке иерархии.

Новый рендерер сохранил иерархическую структуру, но не такую жесткую, и призван упростить и оптимизировать процесс рендеринга. Новая модель рендеринга опирается на понятие *списка отображения* и ориентируется на классы пакета *flash.display*. Список отображения – это иерархия, включающая все видимые объекты ролика *.swf*. Объект, не включенный в список отображения, не будет отрисован рендерером. В каждом ролике только один список отображения, состоящий из трех типов элементов:

---

[1] Понятия MovieClip, экземпляр MovieClip, объект класса MovieClip или просто клип в данном контексте употребляются взаимозаменяемо. – *Примеч. науч. ред.*

Сцена

Сцена – это корень иерархии списка отображения. У каждого ролика всего один объект *Stage*, в котором содержится вся иерархия объектов, отображаемых на экране. Сцена представляет собой контейнер, имеющий обычно всего одного потомка – основной класс приложения *.swf*-ролика. Доступ к сцене обеспечивает свойство `stage` любого отображаемого объекта в списке отображения.

Контейнеры отображаемых объектов

Контейнер отображаемых объектов – это объект, который может содержать дочерние отображаемые объекты. Сцена – это контейнер отображаемых объектов. К контейнерам отображаемых объектов относятся классы *Sprite*, *MovieClip* и *Shape*. При удалении контейнера отображаемых объектов из списка отображения удаляются и все его дочерние объекты.

Отображаемые объекты

Отображаемый объект – это визуальный элемент. Некоторые классы выполняют функцию и отображаемых объектов, и контейнеров отображаемых объектов, например *MovieClip*, тогда как другие классы являются только отображаемыми объектами, например *TextField* (текстовое поле). Уже созданный объект не появится на экране до тех пор, пока не будет добавлен в контейнер отображаемых объектов.

Пример дерева иерархии списка отображения показан на рис. 6.1. Сцена находится в самом верху иерархии, контейнеры отображаемых

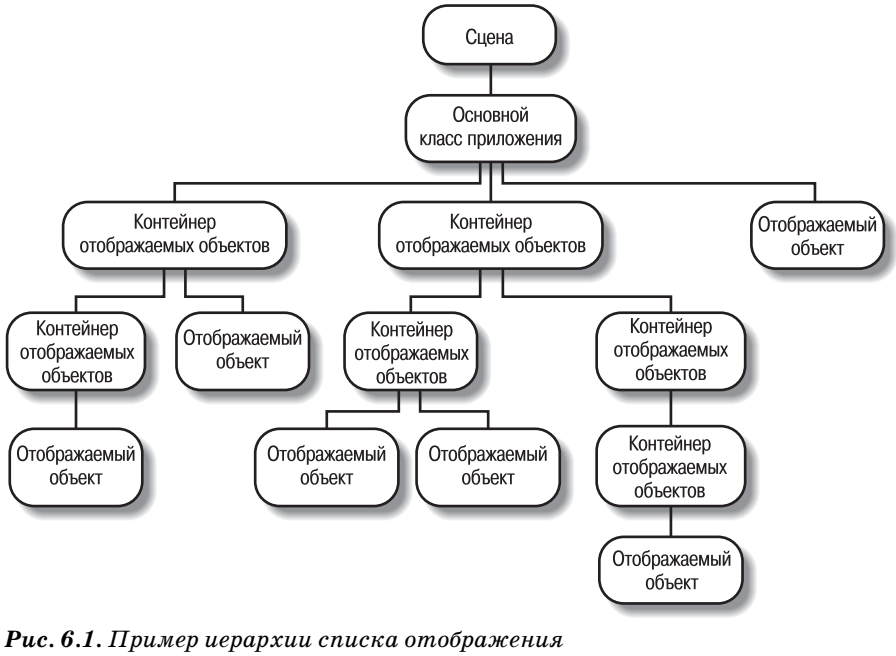


Рис. 6.1. *Пример иерархии списка отображения*

объектов – ветви, а отображаемые объекты – конечные узлы. Элементы, находящиеся вверху диаграммы, на экране отображаются под элементами, располагающимися на диаграмме ниже.

Переход к списку отображения обеспечивает программистам ряд преимуществ по сравнению с предыдущими версиями Flash Player. К ним относятся:

Повышение производительности

В списке отображения находится не только *MovieClip*, но и множество визуальных классов. Применение таких классов, как *Sprite*, снижает требования приложения к памяти, поскольку им не нужна временная шкала. Класс *Shape* может служить для частичной отрисовки, избавляя от необходимости перерисовывать весь экземпляр *MovieClip*. Имея в распоряжении такие более легковесные классы и максимально задействуя их возможности, можно сохранить столь драгоценные ресурсы, как память и процессорное время, что обеспечивает повышение общей производительности.

Упрощение управления глубиной

Иерархия отображения в новой модели списка отображения выполняет функцию управления глубиной. Ранее для создания экземпляров *MovieClip* на соответствующей глубине применялся метод *getNextHighestDepth( )* (получить следующее наибольшее значение глубины) и для управления визуальным порядком расположения – метод *swapDepths( )* (изменить глубины). Управлять глубиной было трудно, утомительно и часто требовало большой тщательности при написании кода. Это была реальность, с которой просто приходилось мириться. Теперь, с появлением модели списка отображения, вопросы порядка расположения решаются практически автоматически, что делает управление глубиной вчерашним днем.

Менее жесткая структура

Предыдущей модели была свойственна довольно негибкая и строгая иерархия. Для ее изменения приходилось уничтожать экземпляры *MovieClip* и повторно создавать их в новой точке. Зачастую это был болезненно медленный процесс, на который уходило много времени и ресурсов. Новая модель списка отображения намного более гибкая: благодаря появлению возможности *смены родителя* (*reparenting*) (обсуждается в рецепте 6.1) можно перемещать целые части дерева иерархии без снижения производительности из-за создания и уничтожения элементов, как это было ранее.

Упрощение создания визуальных элементов

Модель рендеринга списка отображения упрощает создание отображаемых объектов, особенно экземпляров специальных визуальных классов. В предыдущей модели приходилось расширять *MovieClip* в сочетании со специальным связыванием, которое ассоциировало библиотечный элемент с классом ActionScript. Затем метод *attach-*

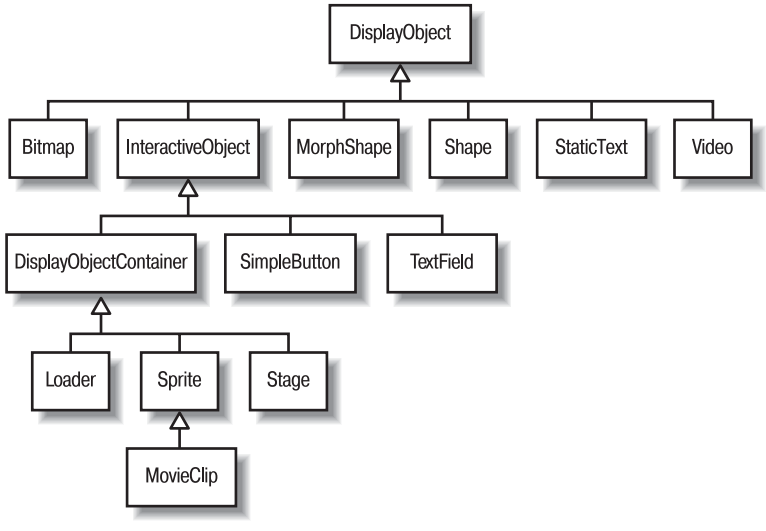


Рис. 6.2. *Иерархия классов списка отображения*

Movie( ) фактически создавал экземпляр специального класса. Теперь, при использовании модели списка отображения, отображаемые объекты создаются путем расширения одного из множества их классов, а экземпляры специальных визуальных классов создаются посредством ключевого слова `new`, что намного проще и понятнее. Подробности см. в рецепте 6.4.

Как уже отмечалось, основные классы модели списка отображения находятся в пакете *flash.display*. Старая модель была сфокусирована на классе *MovieClip*, а для списка отображения центральным является класс *DisplayObject* и его подклассы. Иерархия классов списка отображения показана на рис. 6.2.

У каждого из основных классов свое предназначение. Располагая разными классами, не только *MovieClip*, список отображения предлагает программистам большую гибкость, чем предыдущая модель. Самые широко используемые классы отображения перечислены в табл. 6.1.

Таблица 6.1. Широко используемые классы отображения

Класс отображения	Описание
`DisplayObject`	Базовый класс всех классов списка отображения. `DisplayObject` определяет свойства и методы, общие для всех отображаемых классов. Класс `DisplayObject` не предназначен для создания экземпляров напрямую.
`Bitmap`	Класс `Bitmap` обеспечивает возможность создания и управления изображениями посредством методов `BitmapData`; описывается в главе 8.

Таблица 6.1 (продолжение)

Класс отображения	Описание
`Shape`	Класс `Shape` содержит графические свойства, предусматривающие создание изображений с использований линий, заливок, окружностей, прямоугольников и т.д. Более подробно о нем в главе 7.
`Sprite`	`Sprite` аналогичен `Shape`, но может содержать дочерние отображаемые объекты, такие как текст и видеоизображение. `Sprite` можно рассматривать как `MovieClip` без временной диаграммы.
`MovieClip`	`MovieClip` – хорошо знакомый класс с временной диаграммой и методами для управления головкой воспроизведения. Поскольку `MovieClip` является подклассом `Sprite`, в нем можно рисовать и он также может содержать дочерние отображаемые объекты.
`Video`	Класс `Video` находится в пакете *flash.media*, но также является подклассом `DisplayObject`. Его экземпляры используются для воспроизведения видеоданных, как описывается в главе 16.
`TextField`	Класс `TextField`, находящийся в пакете *flash.text*, обеспечивает возможность создания динамических текстовых полей или полей ввода. Более подробно смотрите в главе 9.
`Loader`	Экземпляры класса `Loader` используются для загрузки внешних визуальных ресурсов, например роликов *.swf* или файлов изображений.

Для тех, кто работал с Flash ранее, переход к модели списка отображения потребует некоторого времени. Сложно уходить от старых привычек, да и модель не так проста. Однако как только вы войдете в курс дела, вы увидите, что не зря потратили время на изучение новой модели. Список отображения коренным образом меняет программирование визуальных объектов во Flash в лучшую сторону.

6.1. Добавление элемента в список отображения

Задача

Требуется добавить в список отображения новый отображаемый объект, чтобы он появился на экране.

Решение

Обратитесь к методам *addChild( )* и *addChildAt( )* класса *DisplayObectContainer*.

Обсуждение

Flash Player состоит из двух основных частей, виртуальной машины ActionScript (ActionScript Virtual Machine, AVM) и устройства ренде-

ринга (Rendering Engine), которые функционируют совместно, образуя единое целое. AVM отвечает за выполнение кода ActionScript, а устройство рендеринга отрисовывает объекты на экране. Поскольку Flash Player образован этими двумя основными частями, процесс создания изображения объекта на экране включает два этапа:

1. Отображаемый объект создается в механизме ActionScript.
2. Отображаемый объект создается в средстве рендеринга и отрисовывается на экране.

На первом этапе с помощью оператора `new` (новый) создается экземпляр отображаемого объекта. Любой объект, добавляемый в список отображения, должен быть прямым или непрямым подклассом *DisplayObject*, как *Sprite*, *MovieClip*, *TextField* или специальным классом, который создавает разработчик (см. рецепт 6.4). Экземпляр *TextField* можно было бы создать так:

```
var hello:TextField = new TextField();
```

Здесь создается отображаемый объект *TextField* в AVM. Но объект пока не отрисовывается на экране, потому что его нет в устройстве рендеринга. Чтобы создать объект в устройстве рендеринга, его надо добавить в иерархию списка отображения. Это можно сделать, вызвав методы *addChild( )* (добавить дочерний элемент) или *addChildAt( )* (добавить дочерний элемент в) экземпляра *DisplayObjectContainer*, который уже находится в иерархии списка отображения.

Метод *addChild( )* принимает один параметр – отображаемый объект, который контейнер должен добавить как дочерний. Следующий фрагмент кода – завершенный пример, демонстрирующий создание объекта в AVM и последующее его создание в устройстве рендеринга путем добавления в список отображения:

```
package {
  import flash.display.DisplayObjectContainer;
  import flash.display.Sprite;
  import flash.text.TextField;

  public class DisplayListExample extends Sprite {
    public function DisplayListExample() {
      // Создаем отображаемый объект
      // в механизме actionscript
      var hello:TextField = new TextField();
      hello.text = "hello";

      // Добавляя отображаемый объект в список
      // отображения, создаем его в устройстве рендеринга,
      // чтобы текстовое поле отрисовывалось на экране
      addChild( hello );
    }
  }
}
```

Здесь класс *DisplayListExample* (пример списка отображения) – основной класс приложения ролика *.swf*. Он расширяет класс *Sprite*. Согласно иерархии классов, описанной на рис. 6.2, *DisplayListExample* представляет собою непрямой подкласс *DisplayObjectContainer* и, следовательно, может иметь дочерние элементы – экземпляры класса *DisplayObject*. Благодаря этому посредством метода *addChild( )* можно добавлять отображаемый объект в контейнер в качестве дочернего элемента.

Отображаемый объект *TextField* создается в конструкторе *DisplayListExample*, тем самым объект создается в AVM. В данный момент этого объекта еще нет на экране, потому что устройство рендеринга пока не знает о нем. Лишь после того как он будет добавлен в список отображения (посредством вызова метода *addChild( )*), *TextField* можно будет увидеть.

Методы *addChild( )* и *addChildAt( )*лишь добавляют отображаемые объекты как дочерние элементы в контейнеры отображаемых объектов, но не обеспечивают их добавление в список отображения. Дочерние элементы добавляются в список отображения, только если контейнер, в который они размещаются, включен в список отображения.

Следующий фрагмент кода демонстрирует, что метод *addChild( )* не гарантирует добавления отображаемого объекта в список отображения. Создается контейнер и некоторый текст в нем, но поскольку контейнер не входит в список отображения, текст не виден:

```
// Создаем текстовое поле для отображения некоторого текста
var hello:TextField = new TextField();
hello.text = "hello";

// Создаем контейнер для размещения TextField
var container:Sprite = new Sprite();
// Добавляем TextField как дочерний элемент контейнера
container.addChild( hello );
```

Чтобы текст появился на экране, его контейнер необходимо добавить в список отображения. Для этого делается ссылка на контейнер отображаемых объектов, уже находящийся в списке отображения, например `root` или `stage`, и вызывается *addChild( )*, в который и передается отображаемый объект контейнера текста. Параметры `root` (корень) и `stage` – это свойства класса *DisplayObject*:

```
// Приводим специальную ссылку root к типу контейнера
// и добавляем контейнер с текстом, чтобы он появился на экране
DisplayObjectContainer( root ).addChild( container );
```

В контейнерах отображаемых объектов может быть по несколько дочерних объектов. Контейнер сохраняет их во внутреннем списке, и порядок расположения дочерних элементов в списке определяет визу-

альный порядок их отображения на экране. Каждый дочерний элемент занимает определенную позицию в списке, которая определяется значением целочисленного индекса (это очень похоже на массив). Позиция 0 – самый нижний элемент списка, он отрисовывается под дочерним элементом, занимающим позицию 1, который, в свою очередь, отрисовывается под дочерним элементом с индексом 2 и т. д. Эта схема аналогична концепции глубины, хорошо знакомой тем, кто уже имеет опыт работы с Flash, но проще в управлении. Позиции заполняются последовательно, т. е. не может быть так, что элементы занимают позицию 0 и позицию 2, но при этом позиция 1 остается свободной.

Если новый дочерний отображаемый объект добавляется с помощью метода *addChild( )*, он отрисовывается поверх всех остальных дочерних элементов контейнера, потому что *addChild( )* помещает его в самый верх списка дочерних элементов, присваивая ему следующий, самый большой индекс позиции. Чтобы добавить дочерний объект и одновременно определить его место в визуальном порядке расположения, применяется метод *addChildAt( )*.

Метод *addChildAt( )* принимает два параметра: дочерний отображаемый объект, который должен быть добавлен, и позицию в порядке расположения, которую он должен занять. Если задана позиция 0, дочерний объект будет добавлен в самый низ списка и окажется (визуально) под всеми другими дочерними элементами. Если ранее в заданной позиции находился другой дочерний элемент, он и все остальные объекты будут сдвинуты вверх на одну позицию, чтобы обеспечить возможность вставить новый дочерний элемент. Если задано недействительное значение позиции, например отрицательное или превышающее количество дочерних элементов в контейнере, будет сформирована ошибка *RangeError* (числовое значение вне допустимого диапазона) и дочерний элемент не будет добавлен.

В следующем примере создаются три цветных круга. Красный и синий добавляются методом *addChild( )*, поэтому синий располагается сверху, т. к. добавляется после красного. После двух вызовов *addChild( )* красный круг занимает позицию 0, и синий – позицию 1. Затем с помощью метода *addChildAt( )*, в котором в качестве местоположения объекта в списке задана позиция 1, между этими объектами вставляется зеленый круг. Синий круг, ранее занимавший позицию 1, сдвигается в позицию 2, и на его место вставляется зеленый круг. В результате красный круг, занимающий позицию 0, отрисован под зеленым кругом, занимающим позицию 1, который, в свою очередь, отрисован под синим кругом, находящимся в позиции 2.

```
package {
  import flash.display.*;
  public class CircleExample extends Sprite {
    public function CircleExample() {
      // Создаем три круга разных цветов и меняем их координаты,
```

```
        // чтобы они все не находились в одной точке, (0,0).
        var red:Shape = createCircle( 0xFF0000, 10 );
        red.x = 10;
        red.y = 20;
        var green:Shape = createCircle( 0x00FF00, 10 );
        green.x = 15;
        green.y = 25;
        var blue:Shape = createCircle( 0x0000FF, 10 );
        blue.x = 20;
        blue.y = 20;

        // Сначала добавляем красный круг, потом синий
        // (так что синий отрисовывается поверх красного)
        addChild( red );
        addChild( blue );

        // Помещаем зеленый круг между красным и синим
        addChildAt( green, 1 );
      }

      // Вспомогательная функция для создания
      // круга заданного цвета и радиуса
      public function createCircle( color:uint, radius:Number ):Shape {
        var shape:Shape = new Shape();
        shape.graphics.beginFill( color );
        shape.graphics.drawCircle( 0, 0, radius );
        shape.graphics.endFill();
        return shape;
      }
    }
  }
```

До сих пор речь шла только о добавлении новых элементов в список отображения, но что произойдет, если применить *addChild( )* к дочернему объекту, который уже находится в списке отображения другого контейнера? В этом состоит идея *смены родителя*: дочерний элемент удаляется из своего контейнера и размещается в другом контейнере.

При смене родителя отображаемого объекта необязательно сначала его удалять. Метод *addChild( )* позаботится обо всем сам.

Следующий пример показывает смену родителя в действии. Для отображения красного, зеленого и синего кругов создается контейнер и добавляется в список отображения. Также создается и добавляется в список отображения еще один контейнер. Затем красный круг перемещается из первого контейнера во второй. Поскольку второй контейнер в списке отображения выше первого, все дочерние объекты второго контейнера визуально располагаются поверх дочерних объектов первого контейнера. Поэтому красный круг отрисовывается поверх синего и зеленого. Красный круг был перемещен из первого контейнера во второй путем вызова метода *addChild( )*.

```
package {
  import flash.display.*;
  public class DisplayListExample extends Sprite {
    public function DisplayListExample() {
      // Создаем три круга разного цвета и меняем их координаты,
      // чтобы они все не находились в одной точке, (0,0).
      var red:Shape = createCircle( 0xFF0000, 10 );
      red.x = 10;
      red.y = 20;
      var green:Shape = createCircle( 0x00FF00, 10 );
      green.x = 15;
      green.y = 25;
      var blue:Shape = createCircle( 0x0000FF, 10 );
      blue.x = 20;
      blue.y = 20;

      // Создаем контейнер и размещаем в нем три круга
      var container1:Sprite = new Sprite();
      container1.addChild( red );
      container1.addChild( green );
      container1.addChild( blue );

      // Добавляем контейнер в список отображения
      addChild( container1 );

      // Создаем и добавляем в список отображения
      // второй контейнер
      var container2:Sprite = new Sprite();
      addChild( container2 );

      // Осуществляем смену родителя для красного круга,
      // в результате чего он отрисовывается поверх
      // зеленого и синего кругов.
      container2.addChild( red );
    }

    // Вспомогательная функция для создания круга
    // заданного цвета и радиуса
    public function createCircle( color:uint, radius:Number ):Shape {
      var shape:Shape = new Shape();
      shape.graphics.beginFill( color );
      shape.graphics.drawCircle( 0, 0, radius );
      shape.graphics.endFill();
      return shape;
    }
  }
}
```

См. также

Рецепты 6.2 и 6.4.

6.2. Удаление элемента из списка отображения

Задача

Требуется удалить элемент из списка отображения и, следовательно, с экрана.

Решение

Обратитесь к методам *removeChild( )* и *removeChildAt( )* класса *DisplayObectContainer*.

Обсуждение

В рецепте 6.1 было показано, как добавлять объекты в список отображения с помощью методов *addChild( )* и *addChildAt( )*. Для удаления дочернего объекта предназначены методы *removeChild( )* (удалить дочерний элемент) или *removeChildAt( )* (удалить дочерний элемент из).

Метод *removeChild( )* принимает один параметр, ссылку на отображаемый объект, который должен быть удален из контейнера. Если объект, указанный как подлежащий удалению, не является дочерним элементом контейнера, будет сформирована ошибка *ArgumentError* (недопустимый аргумент):

```
package {
  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.MouseEvent;

  public class RemoveChildExample extends Sprite {

    // Создаем локальную переменную для хранения ссылки
    // на TextField, чтобы потом можно было его удалить
    private var _label:TextField;

    public function RemoveChildExample() {
      _label = new TextField();
      _label.text = "Некоторый текст";

      // Добавляем в список отображения текстовое поле
      addChild( _label );

      // По щелчку в любой точке сцены удаляем надпись
      stage.addEventListener( MouseEvent.CLICK, removeLabel );
    }

    // Удаляем надпись из списка отображения
    // этого контейнера
    public function removeLabel( event:MouseEvent ):void {
      removeChild( _label );
    }
  }
}
```

В предыдущем примере кода создается локальная переменная `label`, в которой хранится ссылка на *TextField* в самом классе. Это необходимая мера, потому что в метод *removeChild( )* должна быть передана ссылка на подлежащий уничтожению отображаемый объект, так что `label` хранит ссылку для последующего применения. Если обойтись без `label`, то для удаления *TextField* пришлось бы получать ссылку на него, а это лишние усилия, или прибегнуть к методу *removeChildAt( )*.

Если мы не располагаем ссылкой на отображаемый объект, который следует удалить, то можно обратиться к методу *removeChildAt( )*. Аналогично методу *addChildAt( )*, *removeChildAt( )* принимает единственный параметр – индекс подлежащего уничтожению дочернего объекта в списке контейнера. Допустимые значения индекса находятся в диапазоне от `0` до `numChildren - 1`. Если задан недействительный индекс, например отрицательное значение или значение, превышающее число дочерних элементов в контейнере, формируется ошибка *RangeError* и объект не удаляется. Изменяем предыдущий фрагмент кода, применяя в нем метод *removeChildAt( )*, и получаем следующее:

```
package {
  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.MouseEvent;

  public class DisplayListExample extends Sprite {

    public function DisplayListExample() {
      var label:TextField = new TextField();
      label.text = "Некоторый текст";

      // Добавляем в список отображения текстовое поле
      addChild( label );

      // По щелчку мыши в любой точке сцены
      // удаляем надпись
      stage.addEventListener( MouseEvent.CLICK, removeLabel );
    }

    // Удаляем надпись из списка отображения
    // этого контейнера
    public function removeLabel( event:MouseEvent ):void {
      // Удаляем надпись, только если она существует
      if ( numChildren > 0 ) {
        removeChildAt( 0 );
      }
    }
  }
}
```

Самое большое изменение при переходе к методу *removeChildAt( )* заключается в том, что не нужно объявлять переменную `label` для хранения ссылки на *TextField*, чтобы иметь возможность позже удалить его. Теперь при добавлении в контейнер *RemoveChildExample* (пример

удаления дочернего элемента) объект `label` занимает позицию 0. Чтобы удалить этот отображаемый объект, достаточно удалить дочерний элемент в позиции 0.

При этом дочерний отображаемый объект не уничтожается, а лишь удаляется из контейнера и не отрисовывается. Чтобы полностью уничтожить отображаемый объект, всем ссылкам на него надо присвоить значение null.

Если требуется удалить все дочерние элементы контейнера, метод *removeChildAt( )* применяется в сочетании с циклом *for*. У каждого контейнера есть свойство `numChildren`, показывающее, сколько отображаемых объектов находится в списке отображения контейнера. Проходя по всем элементам контейнера с помощью цикла, можно удалить их все на основании позиции в списке дочерних элементов контейнера. Этот процесс несколько осложняется механизмом вычисления позиции.

При удалении любого дочернего элемента все остальные элементы, занимающие более высокие позиции, сдвигаются вниз на одну позицию, заполняя образовавшийся пробел. Рассмотрим контейнер с тремя дочерними элементами в позициях 0, 1 и 2. При удалении элемента из позиции 0 дочерний элемент, занимавший позицию 1, смещается вниз в позицию 0, а дочерний элемент, занимавший позицию 2, смещается вниз на позицию 1. Поскольку значения позиций меняются при каждом удалении дочернего элемента, существует два способа удаления всех дочерних элементов:

1. Всегда удалять дочерний элемент, занимающий позицию 0.
2. Удалять дочерние элементы в обратном порядке, начиная с конца.

До тех пор пока у отображаемого объекта есть дочерние элементы, в позиции 0 всегда будет находиться дочерний элемент. Поэтому в первом случае можно постоянно удалять только элемент, занимающий позицию 0, потому что в каждой итерации цикла это новый дочерний отображаемый объект.

Во втором случае удаление самого последнего дочернего элемента контейнера не приводит к смещению элементов для заполнения освободившейся позиции. Смещаются на одну позицию вниз только элементы, занимающие более высокую позицию, чем удаленный элемент. Последний дочерний элемент занимает самую высокую позицию в контейнере, поэтому ни один из оставшихся элементов не смещается.

Рекомендуется придерживаться первого подхода. Он реализован в классе *ascb.util.DisplayObjectUtilities*:

```
package ascb.util {
  import flash.display.*;
  public class DisplayObjectUtilities {
    // Удаляем все дочерние элементы контейнера
```

```
    public static function removeAllChildren(
                            container:DisplayObjectContainer ):void {

      // Поскольку значение numChildren меняется при каждом удалении
      // дочернего элемента, сохраняем его исходное значение,
      // чтобы можно было правильно осуществлять подсчет
      var count:int = container.numChildren;

      // Проходим по всем дочерним элементам и удаляем их
      for ( var i:int = 0; i < count; i++ ) {
        container.removeChildAt( 0 );
      }
    }
  }
}
```

Работа с методом не отличается сложностью и *DisplayObjectUtilities.removeAllChildren( )*показана в следующем примере:

```
package {
  import flash.display.*;
  import ascb.util.DisplayObjectUtilities;

  public class DisplayListExample extends Sprite {

    public function DisplayListExample() {

      // Добавляем пустые спрайты
      addChild( new Sprite() );
      addChild( new Sprite() );

      // Удаляем все дочерние элементы из этого контейнера
      DisplayObjectUtilities.removeAllChildren( this );

      // Демонстрируем, что все дочерние элементы
      // удалены, - на экран выводится: 0
      trace( numChildren );
    }

  }
}
```

См. также

Рецепт 6.1.

6.3. Изменение визуального порядка размещения объектов

Задача

Требуется изменить порядок, в котором объекты отрисовываются на экране, перемещая их на передний или на задний план относительно других отображаемых объектов.

Решение

Для изменения положения конкретного элемента обратитесь к методу *setChildIndex( )* класса *DisplayObectContainer*. Чтобы обеспечить правильное относительное позиционирование элементов одного уровня, обратитесь к методам *getChildIndex( )* и *getChildAt( )*.

Обсуждение

В рецептах 6.1 и 6.2 было показано, как модель списка отображения работает с визуальным порядком расположения (глубиной). В сущности каждый экземпляр *DisplayObjectContainer* имеет список дочерних элементов, и порядок их расположения в этом списке определяет порядок, в котором эти отображаемые объекты отрисовываются в контейнере. Каждому дочернему элементу присваивается индекс позиции от 0 до `numChildren - 1`, как в массиве. Дочерний элемент, занимающий позицию 0, отрисовывается внизу, под элементом в позиции 1 и др. В списке не может быть незанятых позиций. Если в контейнере три дочерних элемента, они всегда будут располагаться под индексами 0, 1 и 2 (а не, к примеру, 0, 1 и 6).

Класс DisplayObjectContainer предоставляет метод *setChildIndex( )* (задать индекс дочернего элемента) для изменения порядка расположения дочерних элементов в контейнере. Метод принимает два параметра: ссылку на дочерний элемент, подлежащий перемещению, и его новую позицию в контейнере. Заданный индекс позиции должен быть действительным значением. Отрицательные значения или значения, превышающие число элементов в контейнере, обусловят формирование *RangeError*, и функция не будет выполнена соответствующим образом.

В следующем примере создается три цветных круга, синий отрисовывается поверх всех остальных. С помощью метода *setChildIndex( )* синий круг перемещается под два другие за счет изменения его позиции в контейнере (с 2 на 0). Позиции остальных дочерних элементов корректируются соответствующим образом: красный перемещается в позицию 1, а зеленый в позицию 2:

```
package {
  import flash.display.*;
  public class SetChildIndexExample extends Sprite {
    public function SetChildIndexExample() {
      // Создаем три круга разных цветов и меняем их
      // координаты, чтобы они не располагались в одной точке, (0,0).
      var red:Shape = createCircle( 0xFF0000, 10 );
      red.x = 10;
      red.y = 20;
      var green:Shape = createCircle( 0x00FF00, 10 );
      green.x = 15;
      green.y = 25;
      var blue:Shape = createCircle( 0x0000FF, 10 );
```

```
            blue.x = 20;
            blue.y = 20;

            // Добавляем круги, красный под индексом 0,
            // зеленый - 1, и синий - 2
            addChild( red );
            addChild( green );
            addChild( blue );

            // Перемещаем синий круг под другие, размещая
            // его в самом низу списка под индексом 0
            setChildIndex( blue, 0 );
          }

          // Вспомогательная функция для создания круга
          // заданного цвета и радиуса
          public function createCircle( color:uint, radius:Number ):Shape {
            var shape:Shape = new Shape();
            shape.graphics.beginFill( color );
            shape.graphics.drawCircle( 0, 0, radius );
            shape.graphics.endFill();
            return shape;
          }
        }
      }
```

Одно из условий применения метода *setChildIndex( )*: должен быть известен индекс, задаваемый дочернему элементу. Если надо переместить дочерний элемент на самый задний план, то индекс равен 0. Если элемент перемещается на передний план, то в качестве индекса задается `numChildren - 1`. Но что делать, если один дочерний элемент надо просто расположить под другим?

Предположим, что есть два круга, зеленый и синий, и заранее их позиции не известны. Необходимо поставить синий круг за зеленым, но метод *setChildIndex( )* требует задать целое значение, соответствующее новой позиции. Методов, подобных *setChildAbove* (поставить дочерний элемент над) или *setChildBelow* (поставить дочерний элемент под), нет. Решение состоит в том, чтобы с помощью метода *getChildIndex( )* извлечь индекс дочернего элемента и использовать его для изменения позиции другого элемента. Метод *getChildIndex( )* принимает в качестве параметра отображаемый объект и возвращает его индекс в контейнере. Если переданный в метод отображаемый объект не является дочерним элементом данного контейнера, формируется ошибка *ArgumentError*.

В следующем примере тоже создаются зеленый и синий круги. Чтобы поставить синий круг под зеленый, к зеленому применяется метод *getChildIndex( )*. Присваивая синему кругу, изначально занимающему более высокую позицию, индекс зеленого, мы перемещаем его на позицию зеленого. Зеленый же переходит на следующую, более высокую позицию:

```
package {
  import flash.display.*;
  public class GetChildIndexExample extends Sprite {
    public function GetChildIndexExample() {
      // Создаем два круга разных размеров
      var green:Shape = createCircle( 0x00FF00, 10 );
      green.x = 25;
      green.y = 25;
      var blue:Shape = createCircle( 0x0000FF, 20 );
      blue.x = 25;
      blue.y = 25;

      // Добавляем круги в этот контейнер
      addChild( green );
      addChild( blue );

      // Перемещаем синий круг под зеленый.
      // Сначала извлекаем индекс зеленого круга,
      // затем присваиваем его значение синему.
      setChildIndex( blue, getChildIndex( green ) );
    }

    // Вспомогательная функция для создания круга
    // заданного цвета и радиуса
    public function createCircle( color:uint, radius:Number ):Shape {
      var shape:Shape = new Shape();
      shape.graphics.beginFill( color );
      shape.graphics.drawCircle( 0, 0, radius );
      shape.graphics.endFill();
      return shape;
    }
  }
}
```

Когда дочерний элемент перемещается на более низкую позицию, индексы всех элементов, начиная от заданного индекса и вверх до текущего индекса элемента, увеличатся на 1, а дочернему элементу будет присвоен заданный индекс. Когда дочерний элемент перемещается на более высокую позицию, все элементы, начиная от следующего за ним элемента до целевого индекса, перемещаются вниз на 1, а дочернему элементу присваивается заданный индекс.

В общем, если объект *a* находится над объектом *b*, то следующий код переместит *a* под *b*:

```
setChildIndex( a, getChildIndex( b ) );
```

И наоборот, если объект *a* находится под объектом *b*, этот код поставит *a* прямо над *b*.

До сих пор мы работали с дочерними элементами, на которые имели ссылки. Например, переменная *blue* (синий) ссылалась на отображаемый объект, представляющий синий круг, и ее можно было использо-

вать для изменения его индекса. Что делать, если ссылки на объект, который требуется переместить, нет? Ссылка на объект является первым обязательным параметром метода *setChildIndex( )*, поэтому если она недоступна посредством обычной переменной, ее надо каким-то способом получить. Выход – обратиться к методу *getChildAt( )*.

Метод *getChildAt( )* принимает один параметр – индекс в списке дочерних элементов контейнера – и возвращает ссылку на отображаемый объект с этим индексом. Если заданный индекс не является действительным индексом для этого списка, формируется ошибка *RangeError*.

В следующем примере создается несколько кругов разного цвета и размера, и они размещаются в разных точках экрана. По каждому щелчку кнопки мыши самый нижний дочерний элемент помещается поверх всех остальных:

```
package {
  import flash.display.*;
  import flash.events.*;
  public class GetChildAtExample extends Sprite {
    public function GetChildAtExample() {
      // Определяем список используемых цветов
      var color:Array = [ 0xFF0000, 0x990000, 0x660000, 0x00FF00,
                          0x009900, 0x006600, 0x0000FF, 0x000099,
                          0x000066, 0xCCCCCC ];
      // Создаем 10 кругов и выстраиваем их по диагонали
      for ( var i:int = 0; i < 10; i++ ) {
        var circle:Shape = createCircle( color[i], 10 );
        circle.x = i;
        circle.y = i + 10; // + 10 обеспечивает отступ от края сверху

        addChild( circle );
      }

      stage.addEventListener( MouseEvent.CLICK, updateDisplay );
    }

    // Перемещаем самый нижний круг на самый верх
    public function updateDisplay( event:MouseEvent ):void {
      // getChildAt(0) возвращает самый нижний
      // отображаемый объект, который затем перемещается
      // на самый верх за счет задания индекса
      // numChildren - 1 в setChildIndex
      setChildIndex( getChildAt(0), numChildren - 1 );
    }

    // Вспомогательная функция для создания круга
    // заданного цвета и радиуса
    public function createCircle( color:uint, radius:Number ):Shape {
      var shape:Shape = new Shape();
      shape.graphics.beginFill( color );
      shape.graphics.drawCircle( 0, 0, radius );
      shape.graphics.endFill();
```

```
            return shape;
        }
    }
}
```

См. также

Рецепты 6.1 и 6.2.

6.4. Создание специальных визуальных классов

Задача

Требуется создать новый тип *DisplayObject*.

Решение

Создайте новый класс, расширяющий *DisplayObject*, или один из его подклассов, чтобы он мог быть добавлен в контейнер отображаемых объектов посредством методов *addChild( )* или *addChildAt( )*.

Обсуждение

К преимуществам перехода к модели списка отображения следует отнести простоту создания новых визуальных классов. В прошлом существовала возможность расширять *MovieClip*, но для создания экранного экземпляра класса методом *attachMovie( )* в библиотеке, связанной с классом ActionScript, всегда должен был присутствовать символ *MovieClip*. Для создания специального визуального класса всегда приходилось привлекать другие средства, кроме ActionScript. С появлением модели списка отображения все стало намного проще. Теперь все можно делать исключительно в коде ActionScript.

В модели списка отображения, как рассказывалось во введении к данной главе, доступны и другие классы отображения, кроме *MovieClip*. Прежде чем создавать собственный визуальный класс, надо решить, какого типа он будет. Если просто создается специальная форма, расширяется класс *Shape*. Если создается специальная кнопка, вероятно, следует расширять *SimpleButton* (простая кнопка). Если создается контейнер для размещения других отображаемых объектов, хорошим выбором будет класс *Sprite* (если не нужна временная диаграмма). Если же требуется временная диаграмма, понадобится подкласс *MovieClip*.

Все доступные классы отображения специально приспособлены под определенные цели. Лучше сразу определиться, для чего создается специальный класс, и затем исходя из этого выбрать соответствующий родительский класс. Тщательно выбирая родительский класс, мы оптимизируем расходование ресурсов. Например, простой класс *Circle* не нужно наследовать от *MovieClip*, потому что ему не нужна времен-

ная диаграмма. В данном случае лучше остановить свой выбор на классе *Shape*, потому что это самый легковесный вариант, максимально соответствующий концепции круга.

Когда базовый класс выбран, остается только написать код класса. Разовьем пример с кругом и создадим новый класс *Circle*, расширяющий *Shape*. В новом ActionScript-файле *Circle.as* запишем следующий код:

```
package {
  import flash.display.Shape;

  /* Класс Circle – специальный визуальный класс */
  public class Circle extends Shape {

    // Локальные переменные для хранения свойств круга
    private var _color:uint;
    private var _radius:Number;

    /*
     * Конструктор: вызывается при создании Circle.
     * По умолчанию цвет черный, и радиус 10.
     */
    public function Circle( color:uint = 0x000000, radius:Number = 10 ) {
      // Сохраняем значения цвета и радиуса
      _color = color;
      _radius = radius;

      // При создании круга автоматически отрисовываем его
      draw();
    }

    /*
     * Отрисовываем круг исходя из
     * значений цвета и радиуса
     */
    private function draw():void {
      graphics.beginFill( _color );
      graphics.drawCircle( 0, 0, _radius );
      graphics.endFill();
    }
  }
}
```

В этом коде описывается новый отображаемый объект *Circle*. При создании экземпляра *Circle* в конструкторе можно задать и его цвет, и радиус. Для создания тела круга привлекаются методы Drawing API (прикладного интерфейса рисования) (обсуждается в рецепте 7.3) и свойство `graphics`, унаследованное от надкласса *Shape*.

Всегда лучше вынести весь код создания изображения в отдельный метод *draw( )* (рисовать). Конструктор *Circle* не занимается непосредственно отрисовкой круга. Для создания визуальных элементов он вызывает метод *draw( )*.

Осталось создать новые экземпляры нашего класса *Circle* и с помощью методов *addChild( )* или *addChildAt( )* добавить их в список отображения, чтобы они появились на экране. Для создания экземпляров класса предназначено ключевое слово `new`. В следующем примере кода создаются и добавляются в список отображения несколько экземпляров *Circle*:

```
package {
  import flash.display.Sprite;
  public class UsingCircleExample extends Sprite {
    public function UsingCircleExample() {
      // Создаем несколько кругов, используя класс Circle, и меняем
      // их координаты, чтобы они не располагались в одной точке, (0,0).
      var red:Circle = new Circle( 0xFF0000, 10 );
      red.x = 10;
      red.y = 20;
      var green:Circle = new Circle( 0x00FF00, 10 );
      green.x = 15;
      green.y = 25;
      var blue:Circle = new Circle( 0x0000FF, 10 );
      blue.x = 20;
      blue.y = 20;

      // Добавляем круги в список отображения
      addChild( red );
      addChild( green );
      addChild( blue );
    }
  }
}
```

См. также

Рецепты 6.1 и 7.3.

6.5. Создание простых кнопок

Задача

Требуется создать интерактивную кнопку, которая обеспечивает возможность пользователю по щелчку осуществлять некоторые действия, такие как утверждение формы или вычисление суммы.

Решение

Создайте экземпляр класса *SimpleButton* и объекты отображения для состояний `upState`, `downState`, `overState` и `hitTestState`. Или создайте подкласс *SimpleButton*, описывающий требуемое поведение кнопки.

Организуйте вызов метода по нажатию кнопки, обратившись к событию `click`.

Обсуждение

Модель списка отображения сильно упрощает создание кнопок посредством класса *SimpleButton*. Этот класс обеспечивает пользователю возможность взаимодействовать с отображаемым объектом с помощью мыши, а разработчику описывать это взаимодействие через различные состояния кнопки. Перечисленные здесь возможные состояния кнопки доступны как свойства класса *SimpleButton*:

`upState` (кнопка не нажата)

Отображаемый объект для стандартного состояния кнопки («не нажата»). В этом состоянии кнопка находится, когда курсор мыши не над ней.

`overState` (мышь над кнопкой)

Отображаемый объект, определяющий, как выглядит кнопка, когда по ней проводят курсором мыши. Когда курсор покидает область кнопки, она возвращается в состояние «не нажата».

`downState` (кнопка нажата)

Отображаемый объект, предназначенный для представления кнопки, когда она «нажата» (или по ней щелкнули). Кнопка переходит из состояния «мышь над кнопкой» в состояние «нажата», когда пользователь нажимает левую кнопку мыши.

`hitTestState` (активная зона)

Отображаемый объект, определяющий границы кнопки. Кнопка, в активную зону которой попадает курсор мыши, переходит в состояние «мышь над кнопкой». Обычно для состояния `hitTestState` задан тот же отображаемый объект, что и для `upState`. Фактически, `hitTestState` никогда не отображается на экране, это состояние нужно только для отслеживания движений мыши.

Состояние мыши обрабатывается классом *SimpleButton* и основывается на перемещениях мыши. Мы не можем управлять внутренней настройкой состояния кнопки (не нажата, нажата или мышь над кнопкой), можно только задать отображаемый объект, который будет отражать то или иное состояние. Ассоциируя состояния с разными отображаемыми объектами, мы обеспечиваем пользователю обратную связь при взаимодействии с кнопкой.

В следующем примере создается новый экземпляр *SimpleButton* и с помощью приведенных ранее четырех свойств описываются состояния кнопки. Каждому свойству состояния кнопки должен быть задан экземпляр *DisplayObject*. Поэтому задействуется вспомогательный метод *createCircle( )* для создания кругов разных цветов, применяемых в различных состояниях:

```
package {
  import flash.display.*;
```

```
import flash.events.*;

public class SimpleButtonDemo extends Sprite {
  public function SimpleButtonDemo() {
    // Создаем простую кнопку и задаем ее местоположение
    var button:SimpleButton = new SimpleButton();
    button.x = 20;
    button.y = 20;

    // Создаем разные состояния кнопки.
    // Используя вспомогательный метод,
    // создаем круги разных цветов
    button.upState = createCircle( 0x00FF00, 15 );
    button.overState = createCircle( 0xFFFFFF, 16 );
    button.downState = createCircle( 0xCCCCCC, 15 );
    button.hitTestState = button.upState;

    // Добавляем слушатель события click, чтобы знать,
    // когда пользователь щелкает кнопку мыши
    button.addEventListener( MouseEvent.CLICK, handleClick );

    // Наконец, добавляем кнопку в список отображения
    addChild( button );
  }

  // Вспомогательная функция для создания круга
  // заданного цвета и радиуса
  private function createCircle( color:uint, radius:Number ):Shape {
    var circle:Shape = new Shape();
    circle.graphics.lineStyle( 1, 0x000000 );
    circle.graphics.beginFill( color );
    circle.graphics.drawCircle( 0, 0, radius );
    circle.graphics.endFill();
    return circle;
  }

  // Обработчик события, вызываемый при нажатии кнопки
  private function handleClick( event:MouseEvent ):void {
    trace( "Мышь щелкнула по кнопке" );
  }
 }
}
```

После запуска предыдущего фрагмента кода в ролике появляется зеленый круг. Если провести по этому кругу курсором, отрисовывается немного больший белый круг. Если щелкнуть по белому кругу, он превращается в немного меньший серый круг. Эти визуальные эффекты создает экземпляр *SimpleButton*, меняя свое состояние и переключая отображаемые объекты, определенные в четырех свойствах состояния, в зависимости от действий мыши.

Для добавления слушателя событий экземпляра `button` применяется метод *addEventListener( )*, как описывалось в рецепте 1.5. Событие `click`,

заданное с помощью MouseEvent.CLICK, обрабатывается в предыдущем коде методом *handleClick( )*. Когда пользователь щелкает по экземпляру кнопки, вызывается метод *handleClick( )*, обеспечивающий выполнение некоторых действий. В этом простом примере в консоли выводится короткое сообщение («Мышь щелкнула по кнопке»).

Свойство hitTestState, наверное, самое интересное из всех свойств состояния кнопки. В предыдущем коде для hitTestState задан тот же отображаемый объект, что и для upState. Это обычная практика, потому что кнопка должна быть активирована, когда курсор мыши проходит по ее отображаемому объекту состояния upState.

Отображаемый объект hitTestState всегда остается невидимым, но если его не задать, то взаимодействие с кнопкой будет невозможно. Не забывайте задавать hitTestState для *SimpleButton*, даже если это тот же объект, что и для состояния upState.

Если требуется контролировать активные границы кнопки, свойству hitTestState может быть задан любой отображаемый объект. Чтобы увеличить активную зону кнопки, немного изменим предыдущий код и введем такую строку:

```
button.hitTestState = createCircle( 0x000000, 50 );
```

При выполнении этого примера можно заметить, что, в отличие от предыдущего поведения, большой белый круг состояния «мышь над кнопкой» появляется еще до того, как курсор мыши входит в область зеленого круга. Это обусловлено расширением активной зоны до окружности радиусом 50, что увеличило площадь мишени для курсора мыши. Также обратите внимание, что для круга активной зоны задан черный цвет (0x000000). Это сделано специально, чтобы подтвердить тот факт, что отображаемый объект активной зоны всегда остается невидимым.

Альтернативой созданию *SimpleButton* и заданию четырех состояний отображения для каждой кнопки является написание подкласса *SimpleButton*, который определяет визуальный стиль кнопки и создает ее экземпляры. В рецепте 6.4 описывается, как создать новый визуальный класс. Следуя этой методике, можно написать собственную версию *SimpleButton*, упрощая задачу по добавлению множества кнопок в свой ролик.

Следующий код создает класс *RectangleButton* (прямоугольную кнопку). Он определяет поведение *SimpleButton* особого типа, который отрисовывает зеленый прямоугольник с надписью на нем:

```
package {
  import flash.display.*
  import flash.text.*;
  import flash.filters.DropShadowFilter;

  public class RectangleButton extends SimpleButton {
    // Надпись на кнопке
```

```
    private var _text:String;
    // Сохраняем ширину и высоту прямоугольника
    private var _width:Number;
    private var _height:Number;

   public function RectangleButton( text:String, width:Number, height:Number
) {
      // Сохраняем значения, чтобы использовать их
      // при создании состояний кнопки
      _text = text;
      _width = width;
      _height = height;

      // Создаем состояния кнопки на основании
      // значений ширины, высоты и надписи
      upState = createUpState();
      overState = createOverState();
      downState = createDownState();
      hitTestState = upState;
    }

    // Создаем отображаемый объект
    // для состояния кнопки «не нажата»
    private function createUpState():Sprite {
      var sprite:Sprite = new Sprite();

      var background:Shape = createdColoredRectangle( 0x33FF66 );
      var textField:TextField = createTextField( false );


      sprite.addChild( background );
      sprite.addChild( textField );

      return sprite;
    }

    // Создаем отображаемый объект
    // для состояния «мышь над кнопкой»
    private function createOverState():Sprite {
      var sprite:Sprite = new Sprite();

      var background:Shape = createdColoredRectangle( 0x70FF94 );
      var textField:TextField = createTextField( false );

      sprite.addChild( background );
      sprite.addChild( textField );

      return sprite;
    }

    // Создаем отображаемый объект
    // для состояния кнопки «нажата»
    private function createDownState():Sprite {
      var sprite:Sprite = new Sprite();

      var background:Shape = createdColoredRectangle( 0xCCCCCC );
```

```
            var textField:TextField = createTextField( true );

            sprite.addChild( background );
            sprite.addChild( textField );

            return sprite;
        }

        // Создаем прямоугольник со скругленными углами,
        // закрашенный определенным цветом
        private function createdColoredRectangle( color:uint ):Shape {
            var rect:Shape = new Shape();
            rect.graphics.lineStyle( 1, 0x000000 );
            rect.graphics.beginFill( color );
            rect.graphics.drawRoundRect( 0, 0, _width, _height, 15 );
            rect.graphics.endFill();
            rect.filters = [ new DropShadowFilter( 2 ) ];
            return rect;
        }

        // Создаем текстовове поле, в котором будет
        // отображаться надпись на кнопке
        private function createTextField( downState:Boolean ):TextField {
            var textField:TextField = new TextField();
            textField.text = _text;
            textField.width = _width;

            // Центрируем текст по горизонтали
            var format:TextFormat = new TextFormat();
            format.align = TextFormatAlign.CENTER;
            textField.setTextFormat( format );

            // Центрируем текст по вертикали
            textField.y = ( _height - textField.textHeight ) / 2;
            textField.y -= 2;  // Вычитаем 2 пиксела, чтобы обеспечить смещение

            // В состоянии «нажата» текст смещается вниз
            // и вправо относительно других состояний
            if ( downState ) {
                textField.x += 1;
                textField.y += 1;
            }

            return textField;
        }
    }
}
```

Поскольку вся логика отрисовки кнопки инкапсулирована в многократно используемый класс, создавать новые экземпляры кнопки становится намного проще. Вместо того чтобы создавать *SimpleButton* и определять в коде состояния кнопки для каждого экземпляра, теперь можно просто создать новый экземпляр *RectangleButton* и добавить его в список отображения.

Следующий пример показывает, как создать три разных прямоугольных кнопки, используя этот новый экземпляр:

```
package {
  import flash.display.*;
  public class SimpleButtonDemo extends Sprite {
    public function SimpleButtonDemo() {

      // Создаем три прямоугольные кнопки
      // разного размера и с разными надписями
      // и помещаем их в разных точках ролика

      var button1:RectangleButton = new RectangleButton( "Кнопка 1", 60, 100 );
      button1.x = 20;
      button1.y = 20;

      var button2:RectangleButton = new RectangleButton( "Кнопка 2", 80, 30 );
      button2.x = 90;
      button2.y = 20;

      var button3:RectangleButton = new RectangleButton( "Кнопка 3", 100, 40 );
      button3.x = 100;
      button3.y = 60;

      // Добавляем кнопки в список отображения,
      // чтобы они появились на экране
      addChild( button1 );
      addChild( button2 );
      addChild( button3 );
    }
  }
}
```

См. также

Рецепты 1.5, 6.1, 6.4 и 6.8.

6.6. Загрузка внешних изображений во время выполнения

Задача

Требуется загрузить в ролик внешнее изображение во время воспроизведения.

Решение

Для загрузки изображения (*.jpg*, прогрессивный *.jpg*, *.png* или *.gif*) и вывода его на экран используйте новый класс *Loader* (загрузчик).

Обсуждение

В рецепте 9.17 показано, как с помощью тега метаданных `[Embed]` вставить внешние ресурсы в ролик во время компиляции. Чтобы загрузить внешние изображения или ролики во время воспроизведения *.swf*, необходимо задействовать класс *Loader*.

Класс *flash.display.Loader* очень похож на класс *flash.net.URLLoader*, обсуждаемый в рецепте 19.3. Ключевое различие в том, что экземпляры *Loader* могут загружать внешние изображения и ролики и выводить их на экран, тогда как экземпляры *URLLoader* полезны для передачи данных.

Вот три основных шага для загрузки внешнего содержимого:

1. Создается экземпляр класса *Loader*.
2. Экземпляр *Loader* добавляется в список отображения.
3. Вызывается метод *load( )* (загрузить), загружающий внешний ресурс.

Метод *load( )* класса *Loader* отвечает за загрузку изображения или файла *.swf*. В качестве параметра он принимает объект *URLRequest*, который определяет URL ресурса, который должен быть загружен и выведен на экран.

Далее приведен небольшой пример применения экземпляра *Loader* для загрузки изображения *image.jpg* во время выполнения. В коде конструктора *LoaderExample* даются комментарии, соответствующие трем описанным ранее основным этапам загрузки:

```
package {
  import flash.display.*;
  import flash.net.URLRequest;
  public class LoaderExample extends Sprite {
    public function LoaderExample() {
      // 1. Создается экземпляр класса Loader
      var loader:Loader = new Loader();
      // 2. Экземпляр Loader добавляется в список отображения
      addChild( loader );
      // 3. Вызывается метод load(), загружающий внешний ресурс
      loader.load( new URLRequest( "image.jpg" ) );
    }
  }
}
```

При выполнении этого кода Flash Player ищет *image.jpg* в том же каталоге, из которого запускается ролик *.swf*, потому что объект *URLRequest* руководствуется относительным URL. Для указания месторасположения цели может служить как относительный, так и абсолютный URL, но загрузка ресурсов осуществляется в соответствии с политикой безопасности программной среды Flash Player, что обсуждалось

в рецепте 3.12. Когда ресурс загружен, он автоматически добавляется как дочерний элемент экземпляра *Loader*.

При работе с внешними ресурсами существует вероятность сбоя в процессе загрузки. Например, URL может неправильно указывать местоположение ресурса из-за опечатки. Или загрузка ресурса может быть прервана из-за того, что он находится вне безопасной зоны, определенной политикой безопасности программной среды. Или, возможно, ресурс слишком велик и его загрузка занимает очень много времени. Чтобы не смотреть на пустой экран во время загрузки ресурса, рекомендуется отображать предзагрузчик, информирующий о ходе процесса загрузки.

В таких ситуациях в свойство *contentLoaderInfo* (информация о содержимом загрузчика) экземпляра *Loader* необходимо добавить слушателей событий. Это обеспечит возможность реагировать на разные события. Свойство *contentLoaderInfo* – это экземпляр класса *flash.display.LoaderInfo*, созданного для обеспечения информации о загружаемой цели. Ниже представлен список полезных событий, рассылаемых экземплярами класса *LoaderInfo*, и их значений:

open (открытие)

Формируется, когда начинается загрузка ресурса.

progress (в процессе)

Формируется для отображения хода текущей операции.

complete (завершение)

Формируется, когда загрузка ресурса завершена.

init

Формируется, когда свойства и методы загруженного внешнего *.swf* доступны.

httpStatus

Формируется, когда при попытке загрузить ресурс выявлен код состояния неудавшегося HTTP-запроса.

ioError

Формируется при возникновении неустранимой ошибки, в результате которой загрузка отменяется, например в случае невозможности найти ресурс.

securityError (ошибка безопасности)

Формируется, если данные, которые требуется загрузить, находятся вне безопасной программной среды.

unload (выгрузка)

Формируется, если вызван метод *unload( )* для удаления загруженного содержимого или если вновь вызван метод *load( )* для замены уже загруженного содержимого.

Следующий пример показывает, как обрабатываются различные события, связанные с процессом загрузки изображения:

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.net.URLRequest;
  import flash.events.*;

  public class LoaderExample extends Sprite {
    public function LoaderExample() {
      // Создаем загрузчик и добавляем его в список отображения
      var loader:Loader = new Loader();
      addChild( loader );

      // Добавляем обработчики событий
      // для контроля за процессом загрузки
      loader.contentLoaderInfo.addEventListener( Event.OPEN, handleOpen );
      loader.contentLoaderInfo.addEventListener( ProgressEvent.PROGRESS,
handleProgress );
      loader.contentLoaderInfo.addEventListener( Event.COMPLETE,
handleComplete );

      // Загружаем внешнее изображение
      loader.load( new URLRequest( "image.jpg" ) );
    }

    private function handleOpen( event:Event ):void {
      trace( "открытие" );
    }

    private function handleProgress( event:ProgressEvent ):void {
      var percent:Number = event.bytesLoaded / event.bytesTotal * 100;
      trace( "загружено = " + percent );
    }

    private function handleComplete( event:Event ):void {
      trace( "загрузка завершена" );
    }
  }
}
```

При выполнении этого кода пользователь увидит в консоли сообщение об открытии ресурса, за ним последуют одно или два сообщения о прогрессе загрузки с указанием, какая часть ресурса загружена в процентном выражении, и в заключение будет выведено сообщение об успешном окончании загрузки.

Размещая код в обработчиках этих событий, можно обеспечить визуализацию процесса загрузки. Например, метод *handleOpen( )* может отвечать за создание предзагрузчика и добавление его в список отображения. Метод *handleProgress( )* может обновлять показатели объема загруженного содержимого в предзагрузчике, например выводить показания в процентах в экземпляре *TextField*. Наконец, метод *handle-*

Complete( ) мог бы осуществлять «очистку» и удалять предзагрузчик, когда ресурс полностью загружен. Код, реализующий все это, выглядел бы примерно так:

```
private function handleOpen( event:Event ):void {
  // Создаем простой текстовый предзагрузчик
  // и добавляем его в список отображения
  _loaderStatus = new TextField();
  addChild( _loaderStatus );
  _loaderStatus.text = "Загрузка: 0%";
}

private function handleProgress( event:ProgressEvent ):void {
  // Обновляем % загрузки, чтобы проинформировать пользователя о прогрессе
  var percent:Number = event.bytesLoaded / event.bytesTotal * 100;
  _loaderStatus.text = "Загрузка: " + percent + "%";
}

private function handleComplete( event:Event ):void {
  // Проводим очистку - предзагрузчик больше не нужен
  removeChild( _loaderStatus );
  _loaderStatus = null;
}
```

В предыдущем фрагменте кода предполагается, что переменная `_loaderStatus` (состояние загрузки) имеет тип *TextField*:

```
private var _loaderStatus:TextField;
```

Обработчики событий позволяют выводить информацию о загрузке в самом ролике, не прибегая к выражениям *trace( )*, как раньше. Пользователь сразу видит, что происходит. Это очень удобно при загрузке больших внешних ресурсов.

См. также

Рецепты 3.12, 6.7, 9.17 и 19.3.

6.7. Загрузка и взаимодействие с внешними роликами

Задача

В своем ролике требуется загрузить и взаимодействовать с внешним роликом.

Решение

Для загрузки файла *.swf* задействуйте новый класс *Loader*. Доступ к загруженному файлу осуществляйте через свойство `content` (содержимое) экземпляра *Loader*.

Обсуждение

В рецепте 6.6 показано, как с помощью класса *Loader* загружаются внешние изображения. Для загрузки внешних роликов *.swf* используется тот же подход – вызов *load( )* экземпляра *Loader* – только в этот метод передается URL не изображения, а *.swf*. Если *Loader* находится в основной иерархии отображения, *.swf* также появится на экране.

В данном рецепте создаются два отдельных файла *.swf*: *ExternalMovie.swf* и *LoaderExample.swf*. Первый ролик, *ExternalMovie.swf*, будет загружаться во время выполнения во второй, *LoaderExample.swf*. Вот код *ExternalMovie.swf*:

```
package {
  import flash.display.Sprite;
  import flash.display.Shape;
  public class ExternalMovie extends Sprite {
    private var _color:uint = 0x000000;
    private var _circle:Shape;

    public function ExternalMovie() {
      updateDisplay();
    }

    private function updateDisplay():void {
      // Если круг не создан, создаем его и делаем
      // видимым, добавляя в список отображения
      if ( _circle == null ) {
        _circle = new Shape();
        addChild( _circle );
      }

      // Убираем все ранее отрисованное содержимое
      // и рисуем новый закрашенный круг
      _circle.graphics.clear();
      _circle.graphics.beginFill( _color );
      _circle.graphics.drawCircle( 100, 100, 40 );
    }

    // Меняем цвет круга
    public function setColor( color:uint ):void {
      _color = color;
      updateDisplay();
    }

    // Получаем текущий цвет круга
    public function getColor():uint {
      return _color;
    }


  }
}
```

В коде для *ExternalMovie.swf* нет ничего необычного: при воспроизведении ролика отрисовывается черный круг. В первую очередь следует обратить внимание на то, что в этом коде два `public` метода для доступа и изменения цвета круга, *getColor( )* и *setColor( )*. При каждом вызове метода *setColor( )* круг перерисовывается и закрашивается заданным цветом.

Поскольку эти методы объявлены с ключевым словом `public`, они могут вызываться из ролика, который загружает *ExternalMovie.swf* во время выполнения. В отличие от этого, `private` метод *updateDisplay( )* недоступен для загружающего ролика. Более подробно о модификаторах доступа методов рассказано в рецепте 1.13.

Теперь, когда *ExternalMovie.swf* создан, необходимо создать новый *.swf*, который будет загружать внешний ролик. Это будет *LoaderExample.swf* со следующим кодом:

```
package {
  import flash.display.*;
  import flash.net.URLRequest;
  import flash.events.Event;

  public class LoaderExample extends Sprite {

    private var _loader:Loader;

    public function LoaderExample() {
      // Создаем Loader и добавляем его
      // в список отображения
      _loader = new Loader();
      addChild( _loader );

      // Добавляем обработчик события
      // для взаимодействия с загруженным роликом
      _loader.contentLoaderInfo.addEventListener( Event.INIT, handleInit );

      // загружаем внешний ролик
      _loader.load( new URLRequest( "ExternalMovie.swf" ) );
    }

    // Обработчик события, вызываемый, когда
    // загружаемый извне ролик готов к взаимодействию
    private function handleInit( event:Event ):void {
      // Здесь используется *, поскольку
      // во время компиляции тип не известен.
      var movie:* = _loader.content;

      // Вызываем метод внешнего ролика для извлечения
      // данных На экран выводится: 0
      trace( movie.getColor() );

      // Вызываем метод внешнего ролика, чтобы задать
      // данные. Задаем цвет объекта внешнего ролика,
      // в результате чего отрисовывается круг другого
      // цвета, в данном случае красного
```

```
            movie.setColor( 0xFF0000 );
        }
    }
}
```

Код *LoaderExample.swf* более интересен, потому что взаимодействует с загруженным роликом. Предыдущий код выполняет две основные задачи:

1. Слушает событие `init`.
2. Организовывает доступ к загруженному ролику посредством свойства `content`.

Событие `init` формируется, когда загруженный ролик инициализирован и его методы и свойства доступны для взаимодействия. Работа с роликом возможна только *после* формирования события `init` загрузчиком. Попытка взаимодействия с загруженным роликом до его инициализации приведет к формированию ошибки времени выполнения.

Чтобы управлять загруженным роликом, надо получить ссылку на него. Это делается с помощью свойства `content` класса *Loader*. В предыдущем коде переменная `loader` ссылается на *Loader*, который загружает внешний файл *.swf*, поэтому доступ к ролику можно получить через `loader.content`. Если бы не было переменной `loader`, вместо нее можно было бы использовать конструкцию `event.target.content`, потому что `event.target` ссылается на экземпляр, сформировавший событие, т. е. на тот же экземпляр, на который ссылается переменная `loader`.

Свойство `content` доступно только для чтения и возвращает объект *DisplayObject*. В коде *LoaderExample.swf* тип переменной ролика, возвращаемый свойством `content`, не указан. Используется символ `*`. Это необходимо, потому что попытка вызвать методы *getColor( )* или *setColor( )* по ссылке `movie`, если она задана как имеющая тип *DisplayObject*, приведет к ошибке компиляции.

У загруженного ролика, *ExternalMovie.swf*, два `public` метода, доступных для взаимодействия. Эти методы не являются частью класса *DisplayObject*, поэтому попытка вызвать один из них из переменной типа *DisplayObject* является ошибкой. Тип `*` позволяет вызывать любой метод загруженного ролика. Если в загруженном ролике нет вызываемого метода, во время выполнения формируется ошибка *ReferenceError* (недопустимое значение ссылки).

Метод возвращает в *LoaderExample.swf* цвет круга, созданного в *ExternalMovie.swf*. *LoaderExample.swf* сообщает, что значение цвета равно 0, или `0x000000`, и соответствует черному цвету. Метод *setColor( )* *LoaderExample.swf* позволяет менять цвет круга, отрисованного *ExternalMovie.swf*. В данном случае задается красный цвет, и можно увидеть, что *ExternalMovie.swf* обновляет экран после задания нового значения цвета.

Такое взаимодействие возможно только для файлов *.swf* версии 9 и выше. К файлам *.swf* версии 8 и ниже эта техника неприменима, потому что код на ActionScript 3.0 выполняется независимо от кода на ActionScript 1.0 и 2.0. Взаимодействовать с такими файлами *.swf* непросто. В этом случае для отправки и приема сообщений следует обратиться к классу *LocalConnection*. Подробнее об этом рассказано в главе 19.

См. также

Рецепты 1.13 и 6.6.

6.8. Организация взаимодействия с мышью

Задача

Требуется обеспечить возможность взаимодействия пользователей с роликом посредством мыши.

Решение

Задача решается при помощи различных событий мыши, позволяющих отслеживать ее взаимодействие с отображаемыми объектами типа *InteractiveObject* (интерактивный объект). Проверяйте положение мыши относительно отображаемого объекта с помощью доступных только для чтения свойств `mouseX` и `mouseY` экземпляра *DisplayObject* или свойств `localX` и `localY` экземпляра *MouseEvent*, передаваемого в обработчик события.

Обсуждение

Простейшее взаимодействие с мышью позволяет организовать класс *SimpleButton*, как описано в рецепте 6.5. С помощью этого класса нетрудно создать активизируемую щелчком мыши кнопку с различными визуальными состояниями: «не нажата», «мышь над кнопкой» и «нажата».

Однако кнопки не всегда обеспечивают достаточный уровень интерактивности. Обрабатывая различные события мыши, можно создавать любопытные интерактивные приложения. Например, интерактивные программы создания изображений отслеживают перемещения курсора и соответственно им отрисовывают линии на экране. Или, предположим, имеется лабиринт, который пользователь проходит с помощью мыши. Или это может быть игра в гольф, где движение мыши контролирует направление удара клюшки, а удар осуществляется по нажатию кнопки мыши.

Такие ситуации требуют применения специального отображаемого объекта *InteractiveObject*, который обеспечивает возможность отвечать на события мыши. Возвращаясь к введению к данной главе,

вспомним, что класс *InteractiveObject* является базовым классом, занимающим довольно высокую позицию в иерархии классов отображаемых объектов. Поэтому уже хорошо знакомые нам классы *Sprite*, *Loader*, *TextField* и *MovieClip* – все это примеры *InteractiveObject*, поскольку в иерархии они находятся под *InteractiveObject*.

Экземпляры *InteractiveObject* рассылают необходимые события, характерные для мыши. Далее приведен список самых полезных событий мыши:

click (щелчок)

Формируется, если пользователь нажимает и отпускает кнопку мыши, когда курсор находится над интерактивным объектом.

doubleClick (двойной щелчок)

Формируется, если пользователь дважды быстро нажимает и отпускает кнопку мыши, когда курсор находится над интерактивным объектом.

mouseDown

Формируется, если пользователь нажимает кнопку мыши, когда курсор находится над интерактивным объектом.

mouseUp

Формируется, если пользователь отпускает кнопку мыши, когда курсор находится над интерактивным объектом.

mouseOver («мышь над»)

Формируется, когда пользователь переносит курсор мыши на объект.

mouseMove (перемещение мыши)

Формируется, когда пользователь перемещает курсор мыши в рамках интерактивного отображаемого объекта.

mouseOut ()

Формируется, когда пользователь убирает курсор мыши с интерактивного отображаемого объекта.

mouseWheel (колесо мыши)

Формируется, если пользователь прокручивает колесо мыши, когда курсор находится над интерактивным отображаемым объектом.

Чтобы использовать эти события, достаточно вызвать метод *addEventListener( )* экземпляра *InteractiveObject* и определить обработчик события для обработки передаваемого в него *MouseEvent*.

В следующем фрагменте кода создается *Sprite*, в нем отрисовывается красный круг, и при каждом перемещении курсора мыши в круге в консоль выводится сообщение:

```
package {
```

```
import flash.display.Sprite;
import flash.events.*;
import flash.geom.Point;

public class InteractiveMouseDemo extends Sprite {
  public function InteractiveMouseDemo() {
    var circle:Sprite = new Sprite();
    circle.x = 10;
    circle.y = 10;
    circle.graphics.beginFill( 0xFF0000 );
    circle.graphics.drawCircle( 0, 0, 5 );
    circle.graphics.endFill();

    circle.addEventListener( MouseEvent.MOUSE_MOVE, handleMouseMove );

    addChild( circle );
  }

  // Обработчик события движения курсора,
  // когда он находится на круге
  private function handleMouseMove( event:MouseEvent ):void {
    trace( "курсор двигается" );
  }
 }
}
```

Обратите внимание, что в этом примере сообщение появляется при перемещении мыши, только когда курсор находится на круге. Круг в данном случае определяет границы *Sprite*.

События мыши формируются в интерактивном отображаемом объекте, только когда курсор находится в пределах этого объекта.

Другое широко распространенное применение событий мыши представляет собой результат желания отслеживать положение курсора для реализации интерактивных приложений. Например, чтобы отрисовать линию на экране с помощью мыши, необходимо точно знать ее местоположение. Существует два источника информации о положении курсора мыши:

- Свойства `mouseX` и `mouseY`, доступные в любом экземпляре *DisplayObject*.
- Свойства `localX` и `localY` экземпляра *MouseEvent*, переданного в обработчик событий мыши.

Свойства `mouseX` и `mouseY` обеспечивают информацию о положении курсора мыши относительно верхнего левого угла *DisplayObject*. Оба свойства доступны только для чтения; задать положение курсора мыши нельзя, можно только проверить его.

Итак, представим, что прямоугольник находится в точке с координатами (20, 50) и пользователь перемещает курсор в точку (25, 60). Свой-

ство mouseX прямоугольника возвращает значение 5, и свойство mouseY возвращает 10, потому что относительно прямоугольника мышь отстоит на 5 пикселов вправо и 10 пикселов вниз.

Свойства localX и localY *MouseEvent* также возвращают относительные координаты в системе координат, связанной с интерактивным отображаемым объектом, сформировавшим событие. Поэтому если прямоугольник, значение свойства mouseX которого равно 10, формирует событие mouseMove, свойство localX события также имеет значение 10.

Для преобразования относительных координат мыши в абсолютные предназначен метод *localToGlobal( )* класса *DisplayObject*. Метод *localToGlobal( )* принимает в качестве параметра объект *flash.geom.Point*, который определяет локальные координаты, и возвращает новый *Point* уже с глобальными координатами. В следующем фрагменте кода в центре внимания обработчик события и преобразование localX и localY в абсолютные координаты:

```
// Обработчик события mouseMove
private function handleMouseMove( event:MouseEvent ):void {
  /* На экран выводится:
  локальная x: 3.95
  локальная y: 3.45
  */
  trace( "локальная x: " + event.localX );
  trace( "локальная y: " + event.localY );

  // Создаем точку, координаты которой должны быть
  // преобразованы методом localToGlobal
  var localPoint:Point = new Point( event.localX, event.localY );
  // Преобразовываем локальные координаты относительно
  // отображаемого объекта, сформировавшего событие,
  // в глобальные координаты сцены
  var globalPoint:Point = event.target.localToGlobal( localPoint );

  /* На экран выводится:
  глобальная x: 13.95
  глобальная y: 13.45
  */
  trace( " глобальная x: " + globalPoint.x );
  trace( " глобальная y: " + globalPoint.y );
}
```

Реализацию интерактивного взаимодействия посредством обработки различных событий мыши можно представить на следующем примере простой программы создания изображений. По нажатию кнопки мыши начинается отрисовка. Линия на экране повторяет перемещения мыши. Когда пользователь отпускает кнопку, отрисовка прекращается:

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;
```

```
public class DrawingDemo extends Sprite {

  // Флаг, показывающий, находится ли мышь в режиме рисования
  private var _drawing:Boolean;

  public function DrawingDemo() {
    // Задаем стиль линии
    graphics.lineStyle( 2, 0xFFCC33 );

    // Рисование не начинается до тех пор,
    // пока пользователь не нажмет кнопку мыши
    _drawing = false;

    // Добавляем слушателей событий мыши в объект сцены,
    // чтобы получать уведомления о любых событиях мыши,
    // происходящих, когда курсор находится над роликом
    stage.addEventListener( MouseEvent.MOUSE_DOWN, startDrawing );
    stage.addEventListener( MouseEvent.MOUSE_MOVE, draw );
    stage.addEventListener( MouseEvent.MOUSE_UP, stopDrawing );
  }

  public function startDrawing( event:MouseEvent ):void {
    // Перемещаемся в текущее положение мыши,
    // чтобы быть готовыми к рисованию
    graphics.moveTo( mouseX, mouseY );
    _drawing = true;
  }

  public function draw( event:MouseEvent ):void {
    if ( _drawing ) {
      // Рисуем линию от последнего положения мыши до текущего
      graphics.lineTo( mouseX, mouseY );
    }
  }

  public function stopDrawing( event:MouseEvent ):void {
    _drawing = false;
  }

 }
}
```

См. также

Рецепты 6.5, 6.9 и главу 7.

6.9. Перемещение объектов с помощью мыши

Задача

Требуется создать интерфейс с возможностью перемещения объектов методом перетаскивания (drag-and-drop).

Решение

Чтобы реализовать поведение drag-and-drop, обратитесь к методам *startDrag( )*, *stopDrag( )* и свойству `dropTarget` класса *Sprite*. Или можно расширить класс *ascb.display.DraggableSprite* и применить методы *drag( )* и *drop( )*, что обеспечит визуально более равномерное перемещение объектов.

Обсуждение

Реализовать поведение drag-and-drop не так сложно, как может показаться. Класс *Sprite* включает методы, специально предназначенные для этого: *startDrag( )* (начать перетаскивание) и *stopDrag( )* (закончить перетаскивание).

Метод *startDrag( )* можно вызвать для любого экземпляра *Sprite*, чтобы заставить его следовать за мышью, создавая тем самым эффект перетаскивания. Чтобы остановить перетаскивание, вызывается метод *stopDrag( )* экземпляра *Sprite*. После завершения операции drag-and-drop можно проверить свойство `dropTarget` (цель перемещения) экземпляра *Sprite*, чтобы выяснить, на какой объект он помещен. Значение `dropTarget` нужно для проверки допустимости действия (например, что пиктограмма папки помещена в корзину для уничтожения).

Метод *startDrag( )* может принимать два необязательных параметра:

lockCenter

Если задано значение `true`, *Sprite* центрируется относительно курсора мыши независимо от того, где пользователь нажал кнопку мыши. Если `false`, курсор фиксируется в той точке *Sprite*, где была нажата кнопка. Значение по умолчанию равно `false`.

bounds

Прямоугольная область, которой ограничивается перемещение. *Sprite* не может быть вынесен за пределы этой области. Значение по умолчанию, `null`, подразумевает, что ограничения области нет.

В следующем примере кода эти методы применяются для реализации простого поведения drag-and-drop. Слева располагаются три прямоугольника – красный, зеленый и синий, – которые могут быть перемещены. Справа находится белый прямоугольник, являющийся целью для переноса цветных прямоугольников. При размещении цветного прямоугольника на белом последний закрашивается в тот же цвет, что и перенесенный на него прямоугольник:

```
package {
  import flash.display.Sprite;
  import flash.display.DisplayObject;
  import flash.events.MouseEvent;
  import flash.geom.Point;
  import flash.filters.DropShadowFilter;
```

```
public class ColorDrop extends Sprite {

  private var _red:Sprite;
  private var _green:Sprite;
  private var _blue:Sprite;
  private var _white:Sprite;

  // Сохраняем начальные координаты перетаскиваемого объекта Sprite,
  // чтобы можно было вернуть его в исходное положение
  private var startingLocation:Point;

  // Создаем прямоугольники интерфейса и, чтобы сделать
  // их интерактивными, обрабатываем события мыши
  public function ColorDrop() {
    createRectangles();
    addEventListeners();
  }

  private function createRectangles():void {
    _red = new Sprite();
    _red.graphics.beginFill( 0xFF0000 );
    _red.graphics.drawRect( 0, 10, 10, 10 );
    _red.graphics.endFill();

    _green = new Sprite()
    _green.graphics.beginFill( 0x00FF00 );
    _green.graphics.drawRect( 0, 30, 10, 10 );
    _green.graphics.endFill();

    _blue = new Sprite();
    _blue.graphics.beginFill( 0x0000FF );
    _blue.graphics.drawRect( 0, 50, 10, 10 );
    _blue.graphics.endFill();

    _white = new Sprite();
    _white.graphics.beginFill( 0xFFFFFF );
    _white.graphics.drawRect( 20, 10, 50, 50 );
    _white.graphics.endFill();

    addChild( _red );
    addChild( _green );
    addChild( _blue );
    addChild( _white );
  }

  private function addEventListeners():void {
    _red.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _red.addEventListener( MouseEvent.MOUSE_UP, place );

    _green.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _green.addEventListener( MouseEvent.MOUSE_UP, place );

    _blue.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _blue.addEventListener( MouseEvent.MOUSE_UP, place );
  }
```

```
        public function pickup( event:MouseEvent ):void {
          // Сохраняем исходное местоположение,
          // чтобы можно было вернуть цель назад
          startingLocation = new Point();
          startingLocation.x = event.target.x;
          startingLocation.y = event.target.y;

          // Начинаем перетаскивать Sprite, по которому щелкнули мышью,
          // и применяем фильтр отбрасываемой тени,
          // чтобы создать эффект глубины
          event.target.startDrag();
          event.target.filters = [ new DropShadowFilter() ];

          // Переносим цель в верх списка отображения, чтобы
          // он располагался поверх всех остальных объектов
          setChildIndex( DisplayObject( event.target ), numChildren - 1 );
        }

        public function place( event:MouseEvent ):void {
          // Прекращаем перетаскивание Sprite и убираем эффект
          // глубины (т.е. отбрасываемую тень)
          event.target.stopDrag();
          event.target.filters = null;

          // Проверяем, действительно ли Sprite помещен на белый
          // прямоугольник и, если это так, обновляем цвет
          if ( event.target.dropTarget == _white ) {
            // Определяем цвет перемещенного прямоугольника
            // и применяем его к белому прямоугольнику
            var color:uint;
            switch ( event.target ) {
              case _red: color = 0xFF0000; break;
              case _green: color = 0x00FF00; break;
              case _blue: color = 0x0000FF; break;
            }

            _white.graphics.clear();
            _white.graphics.beginFill( color );
            _white.graphics.drawRect( 20, 10, 50, 50 );
            _white.graphics.endFill();
          }

          // Возвращаем перемещенный Sprite
          // в исходное положение
          event.target.x = startingLocation.x;
          event.target.y = startingLocation.y;
        }

      }
    }
```

Рассмотрим этот код. Все прямоугольники добавляются в список отображения и определяются соответствующие слушатели событий `mouse-`

Down и mouseUp. При получении события mouseDown от одного из цветных прямоугольников начинается процесс перемещения (pickup).

Прежде всего сохраняются исходные координаты прямоугольника. Это обеспечивает возможность возвратить его в исходное положение после перемещения. Далее для прямоугольника вызывается метод *startDrag( )*, чтобы начать его перемещение по экрану. Здесь применяется фильтр *DropShadowFilter*, чтобы обеспечить визуальный эффект глубины, как будто перемещаемый прямоугольник выступает над всеми остальными отображаемыми объектами. Наконец прямоугольник помещается в самый верх списка отображения посредством метода *setChildIndex( )* и во время следования за мышью отрисовывается поверх всех остальных объектов.

При регистрации события mouseUp метод *place( )* начинает операцию размещения прямоугольника. Сначала вызывается метод *stopDrag( )* прямоугольника, чтобы он прекратил следовать за мышью, и удаляется фильтр для отмены эффекта глубины. Далее проводится проверка свойства dropTarget прямоугольника, удостоверяющая, что он был опущен на белый прямоугольник. Если выясняется, что белый прямоугольник и есть dropTarget, то он закрашивается в тот же цвет, что и прямоугольник, перемещенный на него. Наконец, поскольку теперь прямоугольник изменил свое местоположение из-за того, что следовал за мышью, он возвращается в исходное положение.

Этот код работает хорошо, но есть два небольших замечания: свойство dropTarget не всегда обеспечивает достоверную информацию и объект двигается неравномерно.

С момента вызова метода *startDrag( )* значение свойства dropTarget постоянно меняется. Это позволяет реализовать обратную связь при переносе объекта над разными возможными целями перемещения – исходя из текущего значения dropTarget можно отображать возможность размещения переносимого объекта на том или ином объекте. Однако изменение dropTarget происходит *не* тогда, когда курсор покидает отображаемый объект, а когда он переходит на *новый*. Это создает проблему, если курсор уходит с объекта и не переходит на другой. В этом случае dropTarget по-прежнему указывает на объект, над которым курсор побывал последним, даже несмотря на то, что курсор уже вышел за его пределы. То есть нет гарантии, что курсор действительно находится над отображаемым объектом, указанным в dropTarget.

Чтобы увидеть этот эффект в действии:

1. Возьмите красный прямоугольник.
2. Пронесите его над белым прямоугольником.
3. Проведите мышью подальше вправо, чтобы красный прямоугольник оказался вне белого.
4. Когда красный прямоугольник окажется *вне* белого, отпустите его.

Белый прямоугольник окрасится в красный цвет, потому что `dropTarget` по-прежнему ссылается на него, даже несмотря на то, что красный прямоугольник перенесен дальше.

Скорректировать это поведение можно с помощью метода *hitTestPoint( )*. Он определяет, находится ли курсор мыши в границах отображаемого объекта `dropTarget`. Метод принимает координаты курсора *x* и *y* и возвращает значения `true` или `false`, свидетельствующие о положении курсора относительно объекта. Необязательный третий *Булев* параметр нужен для определения активной зоны объекта. Если задано значение `false`, рассматривается ограничивающий прямоугольник объекта, тогда как при значении `true` в расчет берутся только фактические очертания объекта. По умолчанию равен `false`.

В методе *place( )*, который отвечает за правильное размещение цветного прямоугольника, в условие, проверяющее `dropTarget`, добавляем вызов *hitTestPoint( )*. Этот метод дает возможность удостовериться в том, что курсор находится в рамках белого прямоугольника, прежде чем будет позволено завершить перенос цветного прямоугольника:

```
if ( event.target.dropTarget == _white
  && _white.hitTestPoint( _white.mouseX, _white.mouseY ) ) {
```

Другая трудность – неравномерное движение перемещаемых методом drag-and-drop объектов вследствие несинхронного обновления экрана. Дело в том, что события мыши не зависят от процесса рендеринга. Частота обновления экрана определяется частотой смены кадров ролика, поэтому изменения, связанные с перемещением мыши, будут отражены только после обычного обновления экрана (контролируемого частотой смены кадров).

Чтобы обойти это препятствие, в класс *MouseEvent* включен метод *updateAfterEvent( )* (обновить после события). Он обычно вызывается при обработке события `mouseMove` и сообщает Flash Player о том, что экран изменился и его надо перерисовать. Это позволяет устранить задержки, обусловленные обновлением экрана с частотой смены кадров, перерисовывать экран синхронно с движением мыши и сделать движение переносимого объекта плавным.

К сожалению, *updateAfterEvent( )* не работает с методом *startDrag( )*. Даже если обработчик события `mouseMove` добавляется с единственной целью – чтобы вызвать *updateAfterEvent( )* для обновления рендеринга, это ничего не даст. С методом *startDrag( )* связана и другая трудность – одновременно можно перетаскивать только один *Sprite*. Хотя, наверное, эта трудность не главная, но все-таки это накладывает существенные ограничения.

Для разрешения этих трудностей в библиотеку «ActionScript 3.0. Сборник рецептов» (которая находится по адресу *http://www.rightactionscript.com/ascb*) был добавлен визуальный класс *DraggableSprite* (перетаскиваемый спрайт). Его можно найти в пакете *ascb.display*.

Класс *DraggableSprite* наследует от *Sprite* и содержит два новых метода с подходящими именами: *drag( )* и *drop( )*. Метод *drag( )* принимает такие же параметры и используется так же, как и метод *startDrag( )*. Поведение метода *drop( )* аналогично *stopDrag( )*.

Основное отличие в том, что доступная в *DraggableSprite* функциональность drag-and-drop реализована специальным кодом отслеживания мыши, который не полагается на внутренние механизмы Flash Player для отслеживания мыши. Тем самым преодолеваются оба негативных аспекта *startDrag( )*. Одновременно можно перетаскивать несколько экземпляров *DraggableSprite*, и проблема задержки рендеринга устранена корректной работой метода *updateAfterEvent( )*.

Однако в *DraggableSprite* нет свойства `dropTarget`. Вместо него применяется метод *getObjectsUnderPoint( )* (получить объекты под курсором), который возвращает объекты, находяшиеся под курсором мыши, и на основании полученной информации определяет возможность размещения на них переносимого объекта.

Метод *getObjectsUnderPoint( )* возвращает массив отображаемых объектов, которые являются дочерними объектами контейнера, вызвавшего метод. Элемент, находящийся в конце массива, в позиции `length-1` — самый верхний (объект, находящийся прямо под курсором). Элемент в позиции 0 самый нижний. Проверяя наличие белого прямоугольника в списке объектов, находящихся под курсором в данный момент, можно определить, можно ли отпустить переносимый объект.

Следующий код реализует такое же поведение drag-and-drop, как раньше, но место *Sprite* в нем занимает *DraggableSprite*:

```
package {
  import flash.display.Sprite;
  import flash.display.DisplayObject;
  import flash.events.MouseEvent;
  import flash.geom.Point;
  import flash.filters.DropShadowFilter;

  import ascb.display.DraggableSprite;

  public class ColorDrop extends Sprite {

    private var _red:DraggableSprite;
    private var _green:DraggableSprite;
    private var _blue:DraggableSprite;
    private var _white:Sprite;

    // Сохраняем начальные координаты перетаскиваемого Sprite,
    // чтобы можно было вернуть его в исходное положение
    private var startingLocation:Point;

    // Создаем прямоугольники интерфейса и, чтобы сделать
    // их интерактивными, обрабатываем события мыши
    public function ColorDrop() {
      createRectangles();
```

```
    addEventListeners();
  }

  private function createRectangles():void {
    _red = new DraggableSprite();
    _red.graphics.beginFill( 0xFF0000 );
    _red.graphics.drawRect( 0, 10, 10, 10 );
    _red.graphics.endFill();

    _green = new DraggableSprite()
    _green.graphics.beginFill( 0x00FF00 );
    _green.graphics.drawRect( 0, 30, 10, 10 );
    _green.graphics.endFill();

    _blue = new DraggableSprite();
    _blue.graphics.beginFill( 0x0000FF );
    _blue.graphics.drawRect( 0, 50, 10, 10 );
    _blue.graphics.endFill();

    _white = new DraggableSprite();
    _white.graphics.beginFill( 0xFFFFFF );
    _white.graphics.drawRect( 20, 10, 50, 50 );
    _white.graphics.endFill();

    addChild( _red );
    addChild( _green );
    addChild( _blue );
    addChild( _white );
  }

  private function addEventListeners():void {
    _red.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _red.addEventListener( MouseEvent.MOUSE_UP, place );

    _green.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _green.addEventListener( MouseEvent.MOUSE_UP, place );

    _blue.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _blue.addEventListener( MouseEvent.MOUSE_UP, place );
  }

  public function pickup( event:MouseEvent ):void {
    // Сохраняем исходное местоположение,
    // чтобы можно было вернуть цель назад
    startingLocation = new Point();
    startingLocation.x = event.target.x;
    startingLocation.y = event.target.y;

    // Начинаем перетаскивать Sprite, по которому
    // щелкнули мышью, и применяем фильтр отбрасываемой
    // тени,чтобы создать эффект глубины
    event.target.drag();
    event.target.filters = [ new DropShadowFilter() ];

    // Переносим цель в верх списка отображения, чтобы
    // он располагался поверх всех остальных объектов
```

```
            setChildIndex( DisplayObject( event.target ), numChildren - 1 );
        }

        public function place( event:MouseEvent ):void {
            // Прекращаем перетаскивание Sprite
            // и убираем эффект глубины
            event.target.drop();
            event.target.filters = null;

            // Получаем список объектов контейнера,
            // находящегося под курсором мыши
            var dropTargets:Array = getObjectsUnderPoint(
                                                new Point( mouseX, mouseY ) );

            // Отображаемый объект в позиции length-1 - самый
            // верхний объект. Это перетаскиваемый прямоугольник.
            // Если белый прямоугольник находится сразу под ним,
            // перетаскиваемый прямоугольник можно отпустить.
            if ( dropTargets[ dropTargets.length - 2 ] == _white ) {
                // Определяем цвет перенесенного прямоугольника
                // и применяем его к белому прямоугольнику
                var color:uint;
                switch ( event.target ) {
                    case _red: color = 0xFF0000; break;
                    case _green: color = 0x00FF00; break;
                    case _blue: color = 0x0000FF; break;
                }

                _white.graphics.clear();
                _white.graphics.beginFill( color );
                _white.graphics.drawRect( 20, 10, 50, 50 );
                _white.graphics.endFill();
            }

            // Возвращаем перенесенный Sprite
            // в исходное положение
            event.target.x = startingLocation.x;
            event.target.y = startingLocation.y;
        }

    }
}
```

См. также

Рецепты 6.4 и 6.8.

7

Программное создание изображений и масок

7.0. Введение

Маски ActionScript позволяют создавать изображения различных отображаемых объектов, таких как *Shape*, *Sprite*, *Button* и *MovieClip*. У каждого из этих классов есть свойство `graphics`, которое представляет собой экземпляр класса *flash.display.Graphics*. Класс *Graphics* определяет API для программного создания графического содержимого. Большинство рецептов данной главы посвящено работе с классом *Graphics*.

Классы *Shape*, *Sprite*, *Button* и *MovieClip* определяют свойства `graphics`, являющиеся ссылками на экземпляры *Graphics*, и поэтому нет необходимости создавать новые объекты *Graphics*. Свойство `graphics` отображаемого объекта обеспечивает рисование в рамках этого объекта. Следующий код задает стиль линии объекта *Graphics*, принадлежащего спрайту `sampleSprite`:

```
sampleSprite.graphics.lineStyle(1);
```

Класс *Graphics* определяет API для рисования основных линий и простых форм. Однако некоторые распространенные фигуры сложно нарисовать с помощью *Graphics*. Поэтому библиотека AS3CBLibrary (*http://www.rightactionscript.com/ascb*) включает класс *ascb.drawing.Pen*. Класс *Pen* (перо) – это посредник (оболочка) для класса *Graphics*. Новый экземпляр *Pen* можно создать, передав в него ссылку на объект *Graphics*:

```
var pen:Pen = new Pen(sampleSprite.graphics);
```

Класс *Pen* играет роль посредника при запросах к методам класса *Graphics*. Это означает, что из класса *Pen* можно вызывать любой метод *Graphics*. Кроме того, *Pen* определяет API, который упрощает ри-

сование дуг, эллипсов, многоугольников, звезд и т. д. Методы класса *Pen* обсуждаются в соответствующих рецептах данной главы.

7.1. Как задать стиль линии

Задача

Требуется задать свойства стиля линии.

Решение

Обратитесь к методу *lineStyle( )*.

Обсуждение

Для программного создания изображений прежде всего необходимо задать свойства стиля линий объекта *Graphics*. По умолчанию стиль линии не определен, поэтому, если не задать эти свойства, линии и заливки не будут отображаться. Стиль линий можно задать с помощью метода *lineStyle( )* (стиль линии) объекта *Graphics*.

Метод принимает восемь параметров, из которых обязательным является только первый:

thickness (толщина)

Толщина линии (в пикселах), по умолчанию равна 1. Диапазон допустимых значений от 0 до 255, где 0 – минимальное значение толщины. Значения, выходящие за рамки диапазона допустимых значений, приводятся к ближайшему действительному значению.

color (цвет)

Цвет линии. Значение по умолчанию равно `0x000000`.

alpha (прозрачность)

Прозрачность (альфа-фактор) линии. Диапазон допустимых значений – от 0 до 1. Значение по умолчанию равно 1.

pixelHinting (пиксельная привязка)

Булево значение, показывающее, должны ли линии отображаться по целым пикселам. По умолчанию равно `false`.

scaleMode (режим масштабирования)

Одна из констант класса *flash.display.LineScaleMode*: `NORMAL` (по умолчанию), `NONE`, `VERTICAL` и `HORIZONTAL`. Когда задано значение `NORMAL`, толщина линии меняется соответственно изменению масштаба объекта, в котором она отрисована. Например, если спрайт, содержащий линию толщиной 1 пиксел, увеличивается в два раза, толщина линии увеличивается до 2 пикселов. Если задан режим масштабирования `NONE`, толщина линии не изменяется. Когда задан режим `VERTICAL`, толщина линии не меняется, если объект меняет раз-

мер только в вертикальном направлении (но меняется, если объект меняет размер по горизонтали). Аналогично, если задан режим HORIZONTAL, толщина линии не меняется при изменении размера объекта только в горизонтальном направлении.

caps (концы линии)

Строка, определяющая форму концов линии. Значение представляет собой одну из констант класса *flash.display.CapsStyle*: NONE, ROUND (по умолчанию) и SQUARE.

joints (соединения)

Строка, определяющая тип соединений отрезков. В качестве значения выступает одна из констант класса *flash.display.JointStyle*: BEVEL (фаска), MITER (клин) и ROUND (скругление) (по умолчанию).

miterLimit (предельная длина клина)

Если задан тип соединения линий MITER, можно определить *предельную длину клина*. Значение по умолчанию равно 3. Диапазон допустимых значений – от 1 до 255. Предельная длина клина определяет, на каком расстоянии от точки соединения отрезков заканчивается клин углового соединения. Эффект от применения этого параметра определяется не только его значением, но и толщиной линии, и углом между соединяющимися отрезками.

Все параметры могут отсутствовать, поэтому самый простой вызов метода *lineStyle( )* имеет следующий вид:

```
sampleSprite.graphics.lineStyle(1);
```

Для изменения стиля линии метод *lineStyle( )* можно вызывать в любой момент. Например, можно задать линию толщиной 20 пикселов зеленого цвета, нарисовать отрезок, задать линию толщиной 10 пикселов синего цвета и провести второй отрезок линией нового стиля.

После вызова метода *clear( )* стиль линии должен задаваться повторно, потому что вызов этого метода возвращает стиль линии в стандартное, неопределенное, состояние.

7.2. Как задавать градиентные стили линии

Задача

Требуется нарисовать линии, применяя градиентные стили.

Решение

Обратитесь к методу *Graphics.lineGradientStyle( )*.

Обсуждение

Метод *lineGradientStyle( )* позволяет отрисовывать линии с градиентной заливкой. Но все равно сначала должен быть вызван метод *line-*

Style( ). После задания базовых стилей линии, чтобы применить к линиям градиентные стили, можно вызвать *lineGradientStyle( )*. Параметры метода *lineGradientStyle( )* точно такие же, что и для метода *beginGradientFill( )*.

См. также

Более подробно о параметрах рассказано в рецепте 7.13.

7.3. Как нарисовать линию

Задача

Требуется нарисовать линию при помощи кода на ActionScript.

Решение

Чтобы нарисовать линию начиная от текущего местоположения пера до заданной точки, обратитесь к методу *Graphics.lineTo( )*.

Обсуждение

Самое простое, что можно нарисовать с помощью ActionScript, – это прямая линия. Flash принимает текущее положение пера за начальную точку, поэтому требуется задать только координаты конечной точки. Для проведения линии от текущего положения пера до заданной точки применяется метод *Grahics.lineTo( )*:

```
// Рисуем линию от текущего положения пера до точки (100,100)
// в системе координат, связанной с объектом sampleSprite.
sampleSprite.graphics.lineTo(100, 100);
```

В случае применения методов ActionScript для рисования все линии и заливки создаются в рамках отображаемого объекта, ассоциированного с объектом *Graphics*, из которого вызываются эти методы. Например, в приведенном выше коде линия отрисовывается в `sampleSprite`.

Как упоминалось ранее, при использовании методов программного интерфейса рисования, таких как *lineTo( )*, Flash отрисовывает линию начиная от текущего положения пера. Если до этого перо никак не перемещалось (посредством вызова методов *lineTo( )*, *curveTo( )* или *moveTo( )*, например), то оно располагается в начале системы координат отображаемого объекта, в точке (0,0). Метод *moveTo( )* позволяет перемещать перо, перенося его в заданную точку, а линия при этом не отрисовывается:

```
// Перемещаем перо в точку (200,20) объекта sampleSprite
sampleSprite.graphics.moveTo(200, 20);
```

Метод *moveTo( )* необходим, когда требуется начать рисование не из центра отображаемого объекта или отрисовывать отдельно стоящие линии или формы:

```
// Задаем стиль линии: толщина 1 пиксел,
// черная, полностью непрозрачная
sampleSprite.graphics.lineStyle(1);

// Рисуем пунктирную линию, состоящую из горизонтальных штрихов и пробелов
sampleSprite.graphics.lineTo(10, 0);
sampleSprite.graphics.moveTo(15, 0);
sampleSprite.graphics.lineTo(25, 0);
sampleSprite.graphics.moveTo(30, 0);
sampleSprite.graphics.lineTo(40, 0);
sampleSprite.graphics.moveTo(45, 0);
sampleSprite.graphics.lineTo(55, 0);
```

Как отмечалось в рецепте 7.1, стиль линии можно менять между отрисовкой линий или сегментов кривой. Таким образом, тогда как предыдущий код отрисовывает четыре черных отрезка, следующий рисует четыре отрезка разных цветов:

```
// Задаем стиль линии: толщина 1 пиксел, черная, непрозрачная
sampleSprite.graphics.lineStyle(1);

// Рисуем пунктирную линию, состоящую из горизонтальных штрихов и пробелов
sampleSprite.graphics.lineTo(10, 0);
sampleSprite.graphics.moveTo(15, 0);

// Меняем цвет линии на синий
sampleSprite.graphics.lineStyle(1, 0x0000FF);
sampleSprite.graphics.lineTo(25, 0);
sampleSprite.graphics.moveTo(30, 0);

// Меняем цвет линии на зеленый
sampleSprite.graphics.lineStyle(1, 0x00FF00);
sampleSprite.graphics.lineTo(40, 0);
sampleSprite.graphics.moveTo(45, 0);

// Меняем цвет линии на красный
sampleSprite.graphics.lineStyle(1, 0xFF0000);
sampleSprite.graphics.lineTo(55, 0);
```

Метод *Pen.drawLine( )* пригодится, если требуется провести линию из одной точки в другую. В предыдущих примерах было показано, как нарисовать ряд отрезков для создания пунктирной линии. Объект *Pen* позволяет упростить пример:

```
var pen:Pen = new Pen(sampleSprite.graphics);

// Рисуем пунктирную линию как ряд отрезков
pen.drawLine(0, 0, 10, 0);
pen.drawLine(15, 0, 25, 0);
pen.drawLine(30, 0, 40, 0);
pen.drawLine(45, 0, 55, 0);
```

См. также

Рецепт 7.1.

7.4. Как нарисовать кривую

Задача

Требуется нарисовать кривую средствами ActionScript.

Решение

Обратитесь к методу *Graphics.curveTo( )*.

Обсуждение

Задав стиль линии, можно посредством метода *curveTo( )* нарисовать кривую. Метод *curveTo( )* отрисовывает кривую Безье (процесс ее построения оптимизирован для сохранения производительности), для чего нужны три точки: начальная, контрольная и конечная:

- Начальная точка всегда определяется текущим положением пера.
- Конечная точка – это точка холста, к которой требуется провести линию.
- Контрольная точка определяет форму кривой. Она представляет собой точку пересечения касательных, проведенных из начальной и конечной точек кривой. Контрольная точка никогда не лежит на кривой . На рис. 7.1 показана контрольная точка кривой.

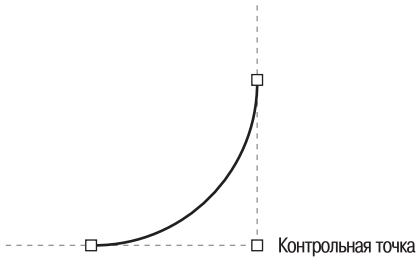


***Рис. 7.1.** Контрольная точка кривой*

Метод *curveTo( )* требует четыре параметра. Первые два определяют координаты `X` и `Y` контрольной точки; следующие два – координаты `X` и `Y` конечной точки. Приведенный ниже код рисует кривую, координаты контрольной и конечной точек которой (0,100) и (100,100) соответственно:

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.curveTo(0, 100, 100, 100);
```

7.5. Как нарисовать дугу

Задача

Требуется нарисовать дугу.

Решение

Это делается при помощи метода *Pen.drawArc( )*.

Обсуждение

Дуга – это часть контура круга. Нарисовать дугу методом *curveTo( )* сложно по разным причинам. Однако метод *Pen.drawArc( )* позволяет быстро нарисовать дугу любого радиуса и длины. Метод *drawArc( )* принимает следующие параметры:

x Координата *x* центра дуги (центра окружности).

y Координата *y* центра дуги.

radius (радиус)

Радиус дуги.

arc

Угловое измерение дуги, заданное в градусах.

startingAngle (начальный угол)

Начальный угол дуги, по умолчанию равный 0.

radialLines (радиальные линии)

Булево значение, показывающее, отрисовываются ли радиальные линии, соединяющие дугу с центром. Значение по умолчанию равно `false`. Если задано `true`, дуга выглядит как сектор.

Следующий код отрисовывает дугу с радиальными линиями, радиусом 50, углом дуги 80° и начальным углом 20°:

```
var pen:Pen = new Pen(graphics);
pen.drawArc(100, 100, 50, 80, 20, true);
```

7.6. Как нарисовать прямоугольник

Задача

Требуется нарисовать прямоугольник во время выполения.

Решение

Нарисуйте четыре соединенных отрезка под прямыми углами друг к другу с помощью метода *Graphics.drawRect( )*. Для изображения прямоугольников со скругленными углами обратитесь к методам *Graphics.drawRoundRect( )* или *Graphics.drawRoundRectComplex( )*.

Обсуждение

ЧС помощью метода *lineTo( )* рисуем четыре линии, образующие прямоугольник:

```
// Задаем черную сплошную линию толщиной 1 пиксел
sampleSprite.graphics.lineStyle(1, 0, 100);

// Рисуем четыре линии, образующие периметр прямоугольника
sampleSprite.graphics.lineTo(100, 0);
sampleSprite.graphics.lineTo(100, 50);
sampleSprite.graphics.lineTo(0, 50);
sampleSprite.graphics.lineTo(0, 0);
```

Как видите, нарисовать простой прямоугольник – премудрость небольшая. Однако в таком четырехкратном вызове метода *lineTo( )* есть несколько недостатков. Самый очевидный из них – минимум пять строк кода: одна для задания стиля линии и четыре, чтобы отрисовать отрезки. Другой недостаток в том, что в этом случае нарисовать повернутый прямоугольник или прямоугольник со скругленными углами не так просто.

Метод *Graphics.drawRect( )* упрощает рисование обычных прямоугольников. Метод принимает четыре обязательных параметра, определяющих координаты X и Y верхнего левого угла и ширину и высоту прямоугольника. Следующий фрагмент кода отрисовывает прямоугольник 100×50, координаты верхнего левого угла которого (0,0):

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.drawRect(0, 0, 100, 50);
```

Метод *Graphics.drawRoundRect( )* отрисовывает прямоугольник со скругленными углами равного радиуса. Метод принимает те же параметры, что и *drawRect( )*, и один дополнительный, определяющий значение радиуса скругления углов. Следующий код отрисовывает тот же прямоугольник, что и в предыдущем случае, за исключением того, что его углы скруглены с радиусом скругления 20:

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.drawRoundRect(0, 0, 100, 50, 20);
```

Метод *Graphics.drawRoundRectComplex( )* работает так же, как и *drawRoundRect( )*, за исключением того, что в нем можно задать радиус скругления для каждого угла отдельно. Метод принимает тот же набор параметров, что и *drawRect( )*, плюс четыре параметра для радиусов. Радиусы задаются в последовательности верхний левый, верхний правый, нижний левый и нижний правый углы. Следующий код отрисовывает такой же прямоугольник, как в предыдущем примере, только все углы имеют разный радиус скругления:

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.drawRoundRect(0, 0, 100, 50, 0, 20, 5, 25);
```

Вызывая методы *beginFill( )*, *beginGradientFill( )* или *beginBitmapFill( )* перед вызовом метода или методов рисования прямоугольника и *endFill( )* после, можно нарисовать закрашенный прямоугольник:

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.beginFill(0xFF0000);
sampleSprite.graphics.drawRectComplex(0, 0, 100, 50);
sampleSprite.graphics.endFill();
```

7.7. Как нарисовать круг

Задача

Требуется нарисовать круг во время воспроизведения ролика.

Решение

Задача решается при помощи метода *Grahics.drawCircle( )*.

Обсуждение

Нарисовать круг стандартными методами рисования не так просто, как может показаться. Во Flash для построения кривой Безье используется только одна контрольная точка, поэтому чтобы нарисовать круг, который выглядит как круг, необходимо минимум восемь сегментов. Для того чтобы нарисовать каждый из сегментов круга, необходимо провести немалое количество вычислений и написать много строк кода. Но класс *Graphics* предлагает метод *drawCircle( )*, который значительно упрощает задачу по отрисовке круга. Метод *drawCircle( )* принимает три обязательных параметра:

x Координата *x* центра круга.

y Координата *y* центра круга.

radius

Радиус круга.

Следующий код рисует круг радиусом 50 с центром в точке (100,100):

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.drawCircle(100, 100, 50);
```

Нарисовать концентрические круги достаточно просто: задаем для всех кругов одни и те же координаты центра:

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.drawCircle(100, 100, 50);
sampleSprite.graphics.drawCircle(100, 100, 100);
```

Закрасить круг можно, вызвав метод *beginFill( )*, *beginGradientFill( )* или *beginBitmapFill( )* перед вызовом *drawCircle( )*, и *endFill( )* после *drawCircle( )*:

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.beginFill(0xFF0000);
sampleSprite.graphics.drawCircle(100, 100, 50);
sampleSprite.graphics.endFill();
```

7.8. Как нарисовать эллипс

Задача

Требуется нарисовать эллипс (овал) во время выполнения.

Решение

Обратитесь к методу *Pen.drawEllipse( )*.

Обсуждение

Эллипс – более абстрактная форма круга. Эллипс определяется двумя взаимно перпендикулярными радиусами, которые называются *большим и малым радиусами*, а иногда просто «радиус *x*» и «радиус *y*».

Рисовать эллипс не намного сложнее, чем круг. Однако, в отличие от рисования кругов, в классе *Graphics* нет метода для рисования эллипсов. Поэтому программно создать эллипс проще всего с помощью одного из методов класса *Pen*. Метод *drawEllipse( )* предусматривает следующие четыре параметра:

x Координата *x* центра эллипса.

y Координата *y* центра эллипса.

xRadius

Радиус эллипса в направлении *x* (большая ось).

yRadius

Радиус эллипса в направлении *y* (малая ось).

Следующий код определяет объект *Pen* и рисует эллипс:

```
var pen:Pen = new Pen(sampleSprite.graphics);

pen.drawEllipse(100, 100, 100, 50);
```

7.9. Как нарисовать треугольник

Задача

Требуется нарисовать треугольник во время выполнения.

Решение

Обратитесь к методу *Pen.drawTriangle( )*.

Обсуждение

Треугольник можно определить и начертить по длинам двух сторон и углу между ними. Это лучше, чем задавать три стороны, потому что две стороны и угол между ними всегда определяют треугольник, тогда как три отрезка произвольной длины могут и не образовывать треугольник.

Рисование треугольника по двум сторонам и углу требует довольно сложных вычислений, поэтому самый простой способ программно нарисовать треугольник – использовать метод *drawTriangle( )* класса *Pen*. Метод *drawTriangle( )* принимает до шести параметров:

x Координата X центра масс (центра) треугольника.

y Координата Y центра масс треугольника.

ab Длина стороны между вершинами *a* и *b*.

ac Длина стороны между вершинами *a* и *c*.

angle

Угол (в градусах) между сторонами *ab* и *ac*.

rotation

Угол поворота треугольника в градусах. Если значение `0` или `undefined`, сторона *ac* параллельна оси *x*.

Определив экземпляр *Pen*, можно вызвать метод *drawTriangle( )* и быстро рисовать треугольник, как показано в следующем примере:

```
var pen:Pen = new Pen(sampleSprite.graphics);
pen.drawTriangle(100, 100, 100, 200, 40);
```

Перед методом *drawTriangle( )* можно вызвать методы *beginFill( )*, *beginGradientFill( )* или *beginBitmapFill( )*, чтобы применить к этой форме заливку. Конечно же, вслед за методом *drawTriangle( )* необходимо вызвать метод *endFill( )*.

Следующий код отрисовывает закрашенный треугольник, выровненный по верхнему левому углу:

```
var pen:Pen = new Pen(sampleSprite.graphics);
pen.beginFill(0xFF0000);
pen.drawTriangle(100, 100, 100, 200, 40);
pen.endFill();
```

См. также

В рецепте 7.10 показано, как рисовать равнобедренный треугольник (и задавать трехстороннюю форму) методом *drawRegularPolygon( )*.

7.10. Как нарисовать правильный многоугольник

Задача

Требуется нарисовать правильный многоугольник (многоугольник, у которого все стороны равны) во время выполнения.

Решение

Задача решается с помощью метода *Pen.drawRegularPolygon( )*.

Обсуждение

Чтобы нарисовать правильный многоугольник, можно создать метод, определяющий углы и координаты вершин посредством основных тригонометрических выражений. Конечно, надо знать тригонометрию и придется написать довольно много кода, поэтому намного проще обратиться к методу *drawRegularPolygon( )* класса *Pen*.

Метод *drawRegularPolygon( )* принимает до пяти параметров:

x Координата *x* центра многоугольника.

y Координата *y* центра многоугольника.

sides (стороны)

Число сторон многоугольника.

length

Длина стороны в пикселах.

rotation

Угол поворота многоугольника в градусах.

Определив экземпляр *Pen*, можно быстро нарисовать правильные многоугольники с любым числом сторон (минимум три стороны, конечно, чтобы это и вправду был многоугольник):

```
var pen:Pen = new Pen(sampleSprite.graphics);

// Рисуем пятиугольник со стороной 50 пикселов
pen.drawRegularPolygon(100, 100, 5, 50);
```

Как и в случае любых других описываемых в данной главе методов рисования, можно создать закрашенный многоугольник, вызывая метод *beginFill( )*, *beginGradientFill( )* или *beginBitmapFill( )* перед *drawRegularPolygon( )*, и метод *endFill( )* после *drawRegularPolygon( )*:

```
var pen:Pen = new Pen(sampleSprite.graphics);
pen.beginFill(0xFF0000);
pen.drawRegularPolygon(100, 100, 5, 50);
pen.endFill();
```

7.11. Как нарисовать звезду

Задача

Требуется нарисовать звезду.

Решение

Обратитесь к методу *Pen.drawStar( )*.

Обсуждение

Метод *Pen.drawStar( )* позволяет быстро нарисовать форму звезды программно. Метод принимает следующие параметры:

x Координата *x* центра звезды.

y Координата *y* центра звезды.

points

Число лучей звезды.

innerRadius (внутренний радиус)

Радиус внутренней части звезды.

outerRadius (внешний радиус)

Радиус внешней части звезды.

rotation

Значение по умолчанию равно 0; можно задать угол поворота в градусах.

Следующий пример рисует пятиконечную звезду:

```
var pen:Pen = new Pen(sampleSprite.graphics);
pen.drawStar(100, 100, 5, 50, 100);
```

7.12. Закрашивание формы прозрачным или непрозрачным цветом

Задача

Требуется нарисовать фигуру и закрасить ее прозрачным или непрозрачным цветом во время выполнения.

Решение

Для начала и завершения закрашивания формы во время выполнения предназначены методы *Graphics.beginFill( )* и *Graphics.endFill( )* соответственно.

Обсуждение

Чтобы отрисовать закрашенную форму, перед вызовом всех методов для рисования вызывается метод *beginFill( )*. После всех методов, отрисовывающих форму, вызывается метод *endFill( )*.

Нельзя закрасить форму, уже отрисованную во время разработки или выполнения. Прежде чем рисовать форму, которую требуется закрасить, необходимо вызвать метод *beginFill( )*.

В данном примере создается зеленый непрозрачный квадрат:

```
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.beginFill(0x00FF00);
sampleSprite.graphics.lineTo(100, 0);
sampleSprite.graphics.lineTo(100, 100);
sampleSprite.graphics.lineTo(0, 100);
sampleSprite.graphics.lineTo(0, 0);
sampleSprite.graphics.endFill();
```

Метод *MovieClip.beginFill( )* принимает два параметра:

fillColor

RGB-значение цвета заливки.

alpha

Значение в диапазоне от 0 (прозрачный) до 1 (непрозрачный), определяющее степень прозрачности, по умолчанию равно 1.

Для создания прозрачной закрашенной формы задается *alpha* меньше 1. Если параметр *alpha* равен 0, форма кажется незакрашенной. Нулевое значение *alpha* часто оказывается очень полезным. Например, требуется создать перетаскиваемый клип, части которого прозрачны. Здесь поможет задание параметра *alpha* для этих частей равным 0.

Метод *endFill( )* не требует никаких параметров. Он просто завершает закрашивание, начатое методами *beginFill( )*, *beginGradientFill( )* или *beginBitmapFill( )*. Чтобы избежать неожиданных результатов при закрашивании, перед вызовом метода *endFill( )* перо должно быть возвращено в точку, с которой начиналось рисование, чтобы фигура оказалась замкнутой.

См. также

Рецепт 7.13.

7.13. Градиентное закрашивание формы

Задача

Требуется нарисовать форму и закрасить ее градиентом во время выполнения.

Решение

Для начала и завершения отрисовки формы во время выполнения применяются методы *beginGradientFill( )* и *endFill( )* соответственно.

Обсуждение

Градиентная заливка (*gradient fill*) характеризуется плавным переходом цветов друг в друга. Flash поддерживает линейные градиенты, в которых цвета сменяются слева направо. (Вертикальная градиентная заливка реализуется посредством поворота градиента при помощи транспонирования матрицы, которое и обсуждается в данном рецепте.) Flash также поддерживает радиальные градиенты, в которых цвета расходятся радиально из центральной точки. Создание фигуры с градиентной заливкой начинается с метода *beginGradientFill( )*, точно так же, как создание закрашенной сплошным цветом фигуры начинается методом *beginFill( )*. Разница в том, что метод *beginGradientFill( )* требует более сложного набора параметров:

gradientType (тип градиента)

Одна из констант класса *flash.display.GradientType*. Возможные варианты: `LINEAR` (линейная) или `RADIAL` (радиальная).

colors

Массив RGB-значений цветов, используемых в градиентной заливке. Цвета отображаются слева направо в линейном градиенте или радиально от центра в радиальном градиенте.

alphas

Массив значений прозрачности, соответствующих цветам в массиве параметра *colors*.

ratios (коэффициенты)

Массив чисел, соответствующих элементам массивов *colors* и *alphas*. Значения массива *ratios* определяют участки градиента, на которых каждый цвет отображается без смешения. Диапазон допустимых значений *ratios* – от 0 (самая левая точка при линейной или самая внутренняя при радиальной заливке) до 255 (самая правая или внешняя точка).

matrix (матрица)

Объект *flash.geom.Matrix*, определяющий преобразование градиента. По умолчанию градиент единичный (1×1), и для правильного заполнения формы его необходимо преобразовать. Класс *Matrix* определяет метод *createGradientBox( )* (создать блок градиента), предназначенный для заполнения объекта. Метод *createGradientBox( )* принимает следующие праметры:

scaleX

Коэффициент масштабирования по горизонтали. Поскольку преобразовывается единичный градиент (1×1), значение *scaleX* соответствует ширине заливки.

scaleY

Коэффициент масштабирования по вертикали. Поскольку преобразовывается единичный градиент (1×1), значение *scaleY* соответствует высоте заливки.

rotation

Разворот градиента в радианах. Градусы преобразуются в радианы путем умножения на *Math.PI/180*. Значение по умолчанию равно 0.

tx

Величина смещения в направлении по оси *x*. Значение по умолчанию – 0.

ty

Величина смещения в направлении по оси *y*, по умолчанию равная 0.

spreadMethod (метод заполнения)

Одна из констант класса *flash.display.SpreadMethod*. Возможные значения: `PAD` (дополнение), `REFLECT` (отражение) и `REPEAT` (повторение). По умолчанию равна `PAD`, и в этом случае градиент дополняется с любой стороны цветом, прилежащим к данной стороне. Например, определен градиент шириной 100 пикселов, начинающийся справа с красного цвета, заканчивающийся слева синим. Если этот градиент применяется к прямоугольнику шириной 200 пикселов, 100 пикселов справа будут закрашены синим цветом. Если задан метод `REFLECT`, то градиент непрерывно повторяется, как при зеркальном отражении. В том же примере градиента от красного к синему шириной 100 пикселов градиент заполнит прямоугольник 200×200 пикселов градиентными полосками от красного к синему и от синего к красному. Если задано значение `REPEAT`, прямоугольник заполняется красно-синими градиентными полосками.

interpolationMethod (метод интерполяции)

Одна из констант класса *flash.display.InterpolationMethod*. Возможные значения: `LINEAR_RGB` и `RGB`. Значение по умолчанию – `RGB`. Метод интерполяции влияет на то, как цвета переходят друг в друга.

focalPointRatio (коэффициент смещения фокальной точки)

Значение от `-1` до `1`, определяющее положение фокуса для радиальных градиентов (не оказывает никакого влияния на линейные градиенты). По умолчанию равно 0 (фокус помещается в центр). Значение `-1` размещает фокус на левом краю градиента, и `1` – на правом.

В следующем примере создается радиальный градиент, заполняющий круг:

```
var matrix:Matrix = new Matrix();
matrix.createGradientBox(100, 100, 0, 50, 50);
var colors:Array = [0xFF0000, 0x0000FF];
var alphas:Array = [100, 100];
var ratios:Array = [0x00, 0xFF];
sampleSprite.graphics.lineStyle(1);
sampleSprite.graphics.beginGradientFill(GradientType.GRADIENT, colors,
alphas, ratios, matrix);
sampleSprite.graphics.drawCircle(100, 100, 50);
sampleSprite.graphics.endFill();
```

7.14. Заливка формы растровым изображением

Задача

Требуется применить к форме заливку растровым изображением.

Решение

Это делается при помощи метода *Graphics.beginBitmapFill( )*.

Обсуждение

Метод *Graphics.beginBitmapFill( )* обеспечивает возможность применять растровое изображение как заливку формы. Метод принимает следующие параметры:

bitmap (растровое изображение)

Объект *BitmapData*, выступающий в качестве заливки.

matrix

По умолчанию растровое изображение применяется без изменений. Чтобы масштабировать, повернуть, наклонить или сдвинуть изображение, можно использовать объект *flash.geom.Matrix*.

repeat (повторить)

Булево значение, показывающее, должно ли повторяться изображение для мозаичного заполнения формы, и по умолчанию равное `true`. Если задано значение `false`, повторяются крайние пикселы изображения до заполнения всей формы.

smooth (гладкий)

Булево значение, определяющее применение сглаживания при увеличении растрового изображения более чем на 100%, и по умолчанию равное `false`.

В следующем примере класса растровое изображение загружается по URL, копируется в объект *BitmapData* и используется как заливка для отрисовываемых программно кругов:

```
package {

  import flash.display.Sprite;
  import flash.geom.Matrix;
  import flash.display.Loader;
  import flash.net.URLRequest;
  import flash.display.BitmapData;
  import flash.events.Event;

  public class Drawing extends Sprite {

    private var _loader:Loader;

    public function Drawing() {
      _loader = new Loader();
      _loader.load(new URLRequest("http://www.rightactionscript.com/
samplefiles/image2.jpg"));
      _loader.contentLoaderInfo.addEventListener(Event.COMPELTE,
onImageLoad);
    }

    private function onImageLoad(event:Event):void {
      var bitmap:BitmapData = new BitmapData(_loader.width, _loader.height);
      bitmap.draw(_loader, new Matrix());
      var matrix:Matrix = new Matrix();
      matrix.scale(.1, .1);
      var sampleSprite:Sprite = new Sprite();
      sampleSprite.graphics.lineStyle(1);
      sampleSprite.graphics.beginBitmapFill(bitmap, matrix);
      sampleSprite.graphics.drawCircle(100, 100, 100);
      sampleSprite.graphics.endFill();
      addChild(sampleSprite);

    }

  }
}
```

7.15. Создание масок

Задача

Требуется создать маску во время выполнения.

Решение

Обратитесь к свойству *DisplayObject.mask*.

Обсуждение

Маски могут служить для создания уникальных форм или визуальных эффектов. Например, маски позволяют создавать эффекты стирания и переходы. Маски применяются для реализации интересных

анимационных эффектов, когда в определенный момент времени видимой остается только маскированная часть изображения. Можно даже анимировать изменение формы маски и использовать ее для маскирования растровых изображений (в клипах).

Любой отображаемый объект может выступать в качестве маски для другого отображаемого объекта посредством свойства `mask`. В следующей строке кода *maskSprite* задается как маска *sampleSprite*:

```
sampleSprite.mask = maskSprite;
```

В приведенном ниже примере отрисовываются две формы и одна из них используется в качестве маски. Обратите внимание, что оба отображаемых объекта добавляются в список отображения методом *addChild( )*. Хотя, в большинстве случаев маски будут работать, даже если объект-маска не внесен в список отображения, рекомендуется включать этот объект в список отображения:

```
var maskSprite:Sprite = new Sprite();
var pen:Pen = new Pen(maskSprite.graphics);
pen.beginFill(0xFFFFFF);
pen.drawArc(100, 100, 50, 80, 20, true);
pen.endFill();
var maskedSprite:Sprite = new Sprite();
maskedSprite.graphics.lineStyle();
maskedSprite.graphics.beginFill(0xFF0000);
maskedSprite.graphics.drawRect(0, 0, 200, 200);
maskedSprite.graphics.endFill();
maskedSprite.mask = maskSprite;
addChild(maskedSprite);
addChild(maskSprite);
```

В следующем примере маска следует за указателем мыши. Маска назначается загрузчику, содержащему загруженное изображение, в результате чего пользователь может видеть только часть изображения, находящуюся под мышью:

```
var loader:Loader = new Loader();
loader.load(new URLRequest("http://www.rightactionscript.com/samplefiles/
image2.jpg"));
addChild(loader);
var maskSprite:Sprite = new Sprite();
maskSprite.graphics.lineStyle();
maskSprite.graphics.beginFill(0xFFFFFF);
maskSprite.graphics.drawCircle(0, 0, 50);
maskSprite.graphics.endFill();
loader.mask = maskSprite;
addChild(maskSprite);
maskSprite.startDrag(true);
```

8

Растровые изображения

8.0. Введение

Во Flash 8 появился класс *BitmapData* – одно из самых важных нововведений на тот момент. С самого своего появления Flash был векторным инструментом. В векторной графике все графические элементы описываются математическими выражениями. Например, линия начинается в точке *x0*, *y0* и продолжается до точки *x1*, *y1*. Растровое изображение, напротив, описывается прямоугольной сеткой значений, где каждому пикселу поставлено в соответствие значение цвета.

Два основных преимущества векторной графики – масштабируемость и размер файла. При изменении масштаба векторного изображения фактически раздвигаются или сдвигаются точки, образующие линии и кривые изображения. Таким образом, можно увеличивать и уменьшать масштаб практически до любых значений, сохраняя при этом гладкость линий и кривых. А в растровом изображении даже при незначительном увеличении каждый пиксел превращается в больший прямоугольник.

Поскольку векторное изображение – это не более чем список координат линий, кривых и форм, размер его файла довольно мал по сравнению с растровым изображением. В растровых изображениях содержатся значения для каждого пиксела. Для изображения размером 100×100 пикселов это минимум 10000 значений. Конечно, в большинстве растровых изображений применяются разные формы сжатия для уменьшения размера файла, но даже в этом случае они могут быть довольно большими.

Преимущества векторной графики сыграли большую роль в том, что Flash стал таким популярным во Всемирной паутине медиа-форматом. Однако у растровых изображений есть свои плюсы. Например, с их помощью намного удобнее представлять фотографии. Для описания всех

форм и цветовых оттенков фотографии потребовалось бы столько векторов, что размер такого файла во много раз превысил бы размер файла растрового изображения этой же фотографии.

Другое преимущество растровых изображений состоит в том, что обычно их обработка требует меньшей нагрузки на процессор, чем обработка векторов. В векторном изображении сначала должно быть вычислено положение каждой точки, а затем по формулам же построены и отрисованы линии и кривые. Рендеринг сложных изображений может занимать довольно много времени. В случае с растровыми изображениями все довольно просто и не зависит от сложности изображения. Если говорить об анимации, то применение растровой графики позволяет получить довольно существенный выигрыш в скорости и производительности.

В версиях Flash до Flash 8 поддержка растровых изображений была минимальной. Хотя их можно было загружать и выводить на экран, что-то сделать с ними во время выполнения было практически невозможно. Класс *BitmapData* предлагает замечательный набор средств для создания и работы с растровыми изображениями во Flash во время выполнения.

8.1. Создание объекта BitmapData

Задача

Требуется создать в приложении новое растровое изображение.

Решение

Для создания нового объекта *BitmapData* применяется конструктор класса *BitmapData*.

Обсуждение

Класс *BitmapData* представляет пикселы растрового изображения и содержит множество методов для добавления содержимого и работы с этим изображением. Первый шаг – создать экземпляр класса, вызвав конструктор:

```
var bitmap:BitmapData = new BitmapData(width, height,
                                       transparent, fillColor);
```

Этот класс является частью пакета *flash.display*, поэтому нельзя забывать импортировать его в самом начале. Параметры `width` и `height` определяют размер создаваемого изображения. Следующий параметр – Булево значение, определяющее, создается ли изображение с альфа-каналом (`true`) или без него (`false`). И параметр `fillColor` определяет исходный цвет фона изображения.

Параметры *width* и *height* обязательны, а *transparent* и *fillColor* по умолчанию принимают значения `true` и `0xFFFFFFFF` соответственно, если не заданы явно в конструкторе.

В качестве значения параметр *fillColor* принимает 32-разрядное значение цвета, т. е. он поддерживает альфа-канал. Конечно, альфа-канал цвета заливки уместен, только если в создаваемом *BitmapData* для параметра `transparent` задано значение `true`. В противном случае все цвета рассматриваются как абсолютно непрозрачные.

В следующем примере создается изначально абсолютно прозрачный объект *BitmapData*, поскольку в `fillColor` для альфа-канала задано нулевое значение:

```
var bitmap:BitmapData = new BitmapData(100, 100,
                                       true, 0x00FFFFFF);
```

Новый объект *BitmapData* сразу после создания существует только в памяти. Хотя уже можно создавать и обрабатывать его содержимое, он не станет видимым до тех пор, пока не будет добавлен в список отображения.

См. также

В рецепте 8.2 рассказывается, как сделать *BitmapData* видимым.

8.2. Как добавить растровое изображение в список отображения

Задача

Объект *BitmapData* создан, и теперь требуется сделать его видимым.

Решение

Создайте *Bitmap* с помощью *BitmapData* и добавьте его в список отображения.

Обсуждение

Чтобы сделать что-либо видимым в приложении на ActionScript 3.0, необходимо добавить это в *список отображения*. (Список отображения рассмотрен в главе 6.) Чтобы добавить объект в список отображения, вызывается метод *addChild( )* основного класса приложения или любого другого объекта, уже находящегося в списке отображения. Однако метод *addChild( )* принимает только объекты, являющиеся подклассами *flash.display.DisplayObject*. Класс *BitmapData* наследуется только от *Object*, поэтому не может быть добавлен в список отображения напрямую.

Чтобы внести *BitmapData* в список отображения, применяется класс *flash.display.Bitmap*, представляющий собою подкласс *DisplayObject*.

Это своего рода оболочка для *BitmapData*, позволяющая вывести его на экран.

При создании нового экземпляра *Bitmap* в его конструктор передается ссылка на *BitmapData*. После этого *Bitmap* можно добавить в список отображения с помощью метода *addChild( )*. В следующем примере создается *BitmapData* с красной заливкой, после чего он выводится на экран посредством *Bitmap*:

```
var bitmap:BitmapData = new BitmapData(100, 100, true, 0xffff0000);
var image:Bitmap = new Bitmap(bitmap);
addChild(image);
```

См. также

Как создать растровое изображение, рассказывается в рецепте 8.1.

8.3. Наполнение растрового изображения графическим содержимым

Задача

В спрайте или любом другом отображаемом объекте имеется некоторое графическое содержимое, которое требуется отрисовать в растровом изображении.

Решение

Чтобы наполнить растровое изображение содержимым, используйте метод *draw( )* класса *BitmapData*.

Обсуждение

Вновь созданный объект *BitmapData* – это всего лишь пустой прямоугольник заданного цвета. В клипе или спрайте может быть некое графическое содержимое или какой-то отображаемый объект, который хотелось бы отрисовать в новом растровом изображении. Метод *draw( )* объекта *BitmapData* предназначен как раз для этого: в него передается объект, который требуется отрисовать. Также можно передать экземпляр класса *flash.geom.Matrix*, который обеспечивает возможность масштабировать, поворачивать, сдвигать или наклонять графическое представление объектов, прежде чем они будут отрисованы. Этот параметр может отсутствовать, однако если необходимо использовать последующие параметры метода, но не хочется изменять объект, то передается *null*. Также, по желанию, можно передать объект *ColorTransform*, который меняет цвет объекта. Подробно о *ColorTransform* рассказывается в рецепте 10.1. В следующем примере в объекте *BitmapData* под именем `bitmap` отрисовывается без изменений спрайт `sprite`:

```
bitmap.draw(_sprite);
```

Это важно, потому что класс *BitmapData* не располагает достаточными средствами для рисования базовых фигур. Можно закрасить пиксел в определенный цвет, создать закрашенный прямоугольник или заливку, здесь есть и некоторые специализированные шумовые функции, но не хватает базовых, например функций рисования линии, кривой, овала или контура. Поэтому, чтобы выйти из ситуации, можно с помощью методов программного интерфейса рисования создавать содержимое в клипе или спрайте и затем переносить этот объект в растровое изображение. В следующем примере создаются *BitmapData* и *Sprite*. Затем методами программного рисования в спрайте рисуется овал и этот спрайт отрисовывается в *BitmapData*:

```
var bitmap:BitmapData = new BitmapData(100, 100, true, 0x00ffffff);
var sprite:Sprite = new Sprite();
sprite.graphics.beginFill(0xff0000, 100);
sprite.graphics.drawEllipse(0, 25, 100, 50);
sprite.graphics.endFill();
bitmap.draw(sprite);
```

Обратите внимание, что фон растрового изображения прозрачный (`0xffffff`), поэтому, хотя оно имеет форму прямоугольника, на экране виден только овал, отрисованный в нем.

См. также

Рецепт 8.1, где рассказывается, как создать растровое изображение, рецепт 8.2, из которого можно узнать, как сделать *BitmapData* видимым, и рецепт 10.1.

8.4. Как загрузить внешнее изображение в экземпляр класса Bitmap

Задача

Требуется загрузить внешнее растровое изображение и работать с ним, как с объектом *BitmapData*.

Решение

Для загрузки изображения предназначен класс *flash.display.Loader*. загрузив изображение, организуйте доступ к свойству *content*, которое является экземпляром *Bitmap*. Доступ к свойству *bitmapData* этого объекта *Bitmap* обеспечивает прямой доступ к объекту *BitmapData*, представляющему загруженное изображение.

Обсуждение

Внешнее растровое изображение загружается с помощью класса *Loader*. Он принимает объект *URLRequest*, содержащий URL загружаемого

изображения. Событие *complete* загрузчика сообщает о завершении загрузки изображения. Ниже приведен код создания загрузчика, подписки на событие *complete* и запуска загрузки изображения.

```
package {
    import flash.display.Sprite;
    import flash.display.Loader;
    import flash.events.Event;
    import flash.net.URLRequest;

    public class BitmapLoader extends Sprite {
        private var _loader:Loader = new Loader();

        public function BitmapLoader() {
            _loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
                                                       onComplete);
            _loader.load(new URLRequest("image.jpg"));
        }
    }
}
```

Сам по себе этот код просто загружает и выводит на экран заданное изображение. Далее создаем метод `onComplete` (по завершении), который обеспечивает доступ к данным только что загруженного растрового изображения:

```
    public function onComplete(event:Event):void {
        var image:Bitmap = Bitmap(_loader.content);
        var bitmap:BitmapData = image.bitmapData;
        addChild(image);
    }
```

Сначала получаем ссылку на свойство *content* загрузчика. Это отображаемый объект, представляющий загруженное содержимое. Если загружен внешний *.swf*, это объект типа *MovieClip*. Однако в данном случае загружалось растровое изображение, поэтому свойство *content* содержит экземпляр *Bitmap*. Значение свойства необходимо привести к типу *Bitmap*, чтобы компилятор не жаловался при попытках доступа к свойствам, присутствующим только в объектах класса *Bitmap*.

После этого можно осуществлять доступ к объекту *BitmapData*, содержащемуся в экземпляре *Bitmap*, посредством его свойства *BitmapData*. Это обеспечивает возможность изменять и управлять загруженным изображением. В данном примере в нем отрисовывается белый квадрат.

Также можно было бы отрисовать загруженное изображение в новом *BitmapData* и вывести его на экран.

```
public function onComplete(event:Event):void {
    var loadedImage:Bitmap = Bitmap(_loader.content);

    // Создаем новый BitmapData и отрисовываем
    // в нем загруженное изображение.
```

```
        var bitmap:BitmapData = new BitmapData(loadedImage.width,
                                    loadedImage.height, false, 0xffffffff);
        bitmap.draw(loadedImage, new Matrix())

        // Создаем новый Bitmap, используя BitmapData,
        // и выводим его на экран.
        var image:Bitmap = new Bitmap(bitmap);
        addChild(image);

        // Манипулируем пикселами
        bitmap.fillRect(new Rectangle(0, 0, 50, 50), 0xffffffff);
    }
```

См. также

Рецепт 8.1, чтобы узнать, как создавать растровое изображение, и рецепт 8.2 для получения информации о том, как сделать *BitmapData* видимым.

8.5. Работа с пикселами

Задача

Требуется задавать или читать значения отдельных пикселов растрового изображения.

Решение

Задача решается с помощью методов *getPixel( )*, *setPixel( )*, *getPixel32( )* и *setPixel32( )* класса *BitmapData*.

Обсуждение

Задавать и читать значения пикселов растрового изображения в ActionScript 3.0 довольно просто. Чтобы получить значение пиксела, в один из методов *getPixel* передаются его координаты x, y. Чтобы задать цвет пиксела, в один из методов *setPixel* передаются координаты и значение цвета.

Методы *getPixel( )* и *setPixel( )* предназначены для работы с непрозрачными экземплярами *BitmapData*, тогда как методы *getPixel32( )* и *setPixel32( )* применяются к изображениям, поддерживающим прозрачность. Непрозрачные изображения имеют разрядность 24 бита, по 8 бит для красного, зеленого и синего каналов. В прозрачных изображениях добавляется еще один 8-разрядный альфа-канал, что в сумме дает 32 бита. Поддерживает *BitmapData* прозрачность или нет, определяется в его конструкторе (см. рецепт 8.1).

Следующий код создает 32-разрядный экземпляр *BitmapData* белого цвета и затем 1000 случайных пикселов делает красными и полупрозрачными:

```
var bitmap:BitmapData = new BitmapData(100, 100, true, 0xffffffff);
var image:Bitmap = new Bitmap(bitmap);
addChild(image);

for(var i:int = 0; i < 1000; i++) {
    bitmap.setPixel32(Math.round(Math.random() * 100),
                      Math.round(Math.random() * 100),
                      0x88ff0000);
}
```

Если метод *setPixel( )* применяется к 32-разрядному *BitmapData*, альфа-каналу этого пиксела присваивается значение, соответствующее абсолютно непрозрачному, независимо от заданного значения. Аналогично, если *setPixel32( )* применяется к непрозрачному изображению, значение альфа-канала игнорируется, поскольку изображение не имеет альфа-канала. В общем при работе с растровыми изображениями лучше пользоваться соответствующей парой методов.

Чтобы получить значение пиксела, в один из методов *getPixel*, в зависимости от типа растрового изображения, необходимо передать его координаты *x* и *y*. Метод возвращает 24- или 32-разрядное число. Следующий класс представляет собой элементарный цветоподборщик. Сначала он при помощи метода *noise( )* (шум) класса *BitmapData* генерирует случайные цвета и добавляет на сцену текстовое поле. Затем класс слушает событие *enterFrame*. Для каждого кадра он получает шестнадцатеричное значение цвета пиксела под курсором мыши и преобразует его в соответствующую строку.

```
package {
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.text.TextField;
    import flash.events.Event;

    public class ColorChooser extends Sprite {
        private var _bitmap:BitmapData;
        private var _textfield:TextField;

        public function ColorChooser() {
            _bitmap = new BitmapData(100, 100, false, 0xffffffff);
            var image:Bitmap = new Bitmap(_bitmap);
            addChild(image);
            _bitmap.noise(1000, 0, 255, 1|2|4, false);

            _textfield = new TextField();
            addChild(_textfield);
            _textfield.y = 100;

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
```

```
                var colorVal:Number = _bitmap.getPixel(mouseX, mouseY)
                _textfield.text = "#" + colorVal.toString(16).toUpperCase();
            }
        }
    }
```

См. также

В рецептах 8.1, 8.6, 8.7, 8.8, 8.9, 8.10 и 8.11 представлены другие способы добавления графического содержимого в растровое изображение.

8.6. Создание прямоугольной заливки

Задача

Требуется закрасить прямоугольную область растрового изображения.

Решение

Возьмите метод *fillRect( )* класса *BitmapData*.

Обсуждение

Класс *BitmapData* не обладает теми мощными средствами создания изображений, которые доступны в программном интерфейсе рисования для клипов и спрайтов. Однако у него есть базовый инструмент прямоугольной заливки. Работать с этим методом довольно легко. В него достаточно передать прямоугольную область, которую требуется закрасить, и цвет заливки:

```
_bitmap.fillRect(прямоугольник, цвет);
```

Прямоугольник должен быть экземпляром класса *flash.geom.Rectangle*. Экземпляр этого класса можно создать посредством его конструктора, задав *x* и *y* (ширину и высоту прямоугольника):

```
var rect:Rectangle = new Rectangle(0, 0, 50, 100);
```

Следующий фрагмент кода создает пустое растровое изображение с белым фоном и затем в его середине отрисовывает красный квадрат:

```
public function RectExample() {
    _bitmap = new BitmapData(100, 100, false, 0xffffffff);
    var image:Bitmap = new Bitmap(_bitmap);
    addChild(image);
    _bitmap.fillRect(new Rectangle(25, 25, 50, 50), 0xffff0000);
}
```

Обратите внимание, что для прозрачных и непрозрачных изображений существует единая версия *fillRect( )*. Если метод применяется к изображению, не имеющему альфа-канала, то в него передается 24-разрядное значение цвета, поскольку дополнительные разряды альфа-канала все равно будут проигнорированы. Работая с прозрачным изобра-

жением, помните, что необходимо передавать полное 32-разрядное число. Если передать 24-разрядное значение, то для альфа-канала будет принято минимальное значение (0%).

См. также

В рецептах 8.5, 8.7, 8.8, 8.9, 8.10 и 8.11 представлены другие способы добавления графического содержимого в растровое изображение.

8.7. Создание заливки области

Задача

Требуется закрасить большую область, возможно, неправильной формы.

Решение

Для решения нужен метод *floodFill( )* класса *BitmapData*.

Обсуждение

Синтаксис метода *floodFill( )* аналогичен методу *setPixel( )*. То есть в него передаются координаты *x*, *y* и цвет. Метод закрашивает заданный пиксел и все пикселы, окружающие его, в один цвет. Это аналогично применению инструмента заливки в большинстве графических программ, таких как Adobe Photoshop.

Следующий код демонстрирует этот метод в действии. Сначала создается растровое изображение и ряд случайных квадратов. Затем задается обработчик события `mouseDown`, который осуществляет заливку области выбранного пиксела:

```
package {
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.MouseEvent;
    import flash.geom.Rectangle;

    public class FloodFillDemo extends Sprite {
        private var _bitmap:BitmapData;

        public function FloodFillDemo () {
            var sprite:Sprite = new Sprite();
            addChild(sprite);
            _bitmap = new BitmapData(stage.stageWidth,
                                     stage.stageHeight,
                                     false, 0xffffffff);

            for(var i:int = 0; i < 20; i++) {
                _bitmap.fillRect(new Rectangle(
                             Math.random() * stage.stageWidth,
```

```
                                Math.random() * stage.stageHeight,
                                50, 50), Math.random() * 0xffffffff);
                    }
                    var image:Bitmap = new Bitmap(_bitmap);
                    sprite.addChild(image);
                    sprite.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
                }

                public function onMouseDown(event:MouseEvent):void {
                    _bitmap.floodFill(mouseX, mouseY, 0xffff0000);
                }
            }
        }
```

См. также

В рецептах 8.5, 8.7, 8.8, 8.9, 8.10 и 8.11 представлены другие способы добавления графического содержимого в растровое изображение.

8.8. Копирование пикселов

Задача

Требуется скопировать прямоугольную область из одного *BitmapData* в другой.

Решение

Задача решается при помощи метода *copyPixels( )* класса *BitmapData*.

Обсуждение

Принцип работы метода *copyPixels( )* (копировать пикселы) довольно прост: он принимает данные пикселов одного растрового изображения и переносит их в другое. В этом смысле он очень похож на метод *draw( )*. Однако *copyPixels( )* обеспечивает возможность управлять тем, какая часть растрового изображения копируется и куда. Для этого надо задать исходный прямоугольник и точку назначения:

```
bitmap.copyPixels(sourceBmp, srcRect, destPoint);
```

Исходный прямоугольник – это экземпляр класса *flash.geom.Rectangle*. Он определяет прямоугольную область в исходном объекте *BitmapData*, которая будет скопирована. Копируется только данная часть изображения.

Точка назначения – экземпляр *flash.geom.Point*. Она определяет координаты x, y целевого растрового изображения, куда требуется вставить скопированные пикселы.

В следующем примере показано, как скопировать несколько прямоугольных областей загруженного растрового изображения в другой объект *BitmapData*:

```
package {
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Loader;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.geom.Point;
    import flash.geom.Rectangle;

    public class AS3CB extends Sprite {
        private var _bitmap:BitmapData;
        private var _loader:Loader;

        public function AS3CB() {
            _loader = new Loader();
           _loader.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoad);
            _loader.load(new URLRequest("myphoto.jpg"));
            _bitmap = new BitmapData(stage.stageWidth,
                                     stage.stageHeight,
                                     false, 0xffffffff);
            var image:Bitmap = new Bitmap(_bitmap);
            addChild(image);
        }

        public function onLoad(event:Event):void {
            var loaderBmp:Bitmap = Bitmap(_loader.content);
            var w:Number = loaderBmp.width / 10;
            for(var i:int = 0; i < 10; i++) {
                _bitmap.copyPixels(loaderBmp.bitmapData,
                                   new Rectangle(i * w, 0,
                                                 w, loaderBmp.height),
                                   new Point(i * (w + 2), i));
            }
        }
    }
}
```

Часть кода, представляющая особый интерес, выделена жирным шрифтом.

Вертикальные полоски из загруженного изображения последовательно копируются в заданный прямоугольник и посредством объекта *Point* размещаются в новом растровом изображении. Расстояния между ними увеличиваются на два пиксела.

Есть еще два необязательных параметра. Можно определить другой объект *BitmapData* как альфа-канал и смешать альфа-уровни этого и существующего растрового изображения. Синтаксис такой:

```
bitmap.copyPixels(sourceBmp, srcRect, destPoint,
              alphaBmp, alphaDestRect, blend);
```

Это более развернутый вариант применения метода. Он не рассматривается здесь подробно, но читатель может самостоятельно поэкспериментировать с ним.

См. также

В рецептах 8.5, 8.7, 8.9, 8.10 и 8.11 представлены другие способы добавления графического содержимого в растровое изображение.

8.9. Копирование каналов

Задача

Требуется скопировать красный, зеленый, синий или альфа-каналы одного *BitmapData* в другой.

Решение

Для это нужен метод *copyChannel( )* класса *BitmapData*.

Обсуждение

Метод *copyChannel( )* (копировать канал) – это еще один метод, переносящий информацию из одного растрового изображение в другое. Фактически первые три аргумента аналогичны аргументам метода *copyPixels( )*. Кроме того, в методе присутствуют параметры, определяющие исходный канал и канал назначения:

```
bitmap.copyPixels(sourceBmp, srcRect, destPoint,
                  srcChannel, destChannel);
```

Значениями двух параметров каналов могут быть целые 1, 2, 4 или 8, которые соответствуют красному, зеленому, синему и альфа-каналам. Чтобы избежать опечаток, рекомендуется работать со статическими свойствами класса *BitmapDataChannel*: `RED`, `GREEN`, `BLUE` и `ALPHA`.

Достаточно указать методу, какой канал требуется взять из исходного изображения и в какой канал целевого изображения его надо скопировать. В следующем коде в новое растровое изображение с небольшим смещением копируются красный, зеленый и синий каналы загруженного изображения:

```
var loaderBmp:Bitmap = Bitmap(loader.content);
bitmap.copyChannel(loaderBmp.bitmapData,
                   loaderBmp.bitmapData.rect,
                   new Point(),
                   BitmapDataChannel.RED,
                   BitmapDataChannel.RED);

bitmap.copyChannel(loaderBmp.bitmapData,
                   loaderBmp.bitmapData.rect,
```

```
                    new Point(5, 5),
                    BitmapDataChannel.GREEN,
                    BitmapDataChannel.GREEN);

bitmap.copyChannel(loaderBmp.bitmapData,
                    loaderBmp.bitmapData.rect,
                    new Point(10, 10),
                    BitmapDataChannel.BLUE,
                    BitmapDataChannel.BLUE);
```

См. также

В рецептах 8.5, 8.7, 8.8, 8.10 и 8.11 представлены другие способы добавления графического содержимого в растровое изображение.

8.10. Создание шума

Задача

В растровом изображении требуется создать узор случайного шума.

Решение

Для решения нужен метод *noise( )* класса *BitmapData*.

Обсуждение

Фильтр, или узор, шума просто присваивает случайное значение каждому пикселу выбранной области. Можно было бы с помощью цикла перебрать все пикселы растрового изображения и методом *setPixel( )* изменить их цвет случайным образом. Однако есть более простой и мощный способ сделать то же самое в одной строке кода: метод *noise( )*.

Сам по себе метод *noise( )* создает случайное пестрое изображение, как то, которое можно увидеть, когда исчезает сигнал в телевизоре. Однако, сочетая шум с другими фильтрами, например фильтром размытия, можно получить очень полезные эффекты.

Метод *noise( )* вызывается прямо из экземпляра *BitmapData*. Его параметры очень просты:

```
bitmap.noise(seed, low, high, channel, grayscale);
```

Параметр *seed* (зерно) определяет узор шума. Его значением может быть любое целое число. Если дважды вызвать метод *noise( )* с одним и тем же значением *seed*, будет получен один и тот же узор шума. Чтобы всегда гарантированно получать разные узоры, при каждом вызове метода в него передают случайное число:

```
Math.random() * 100000
```

Параметры `low` (низкий) и `high` (высокий) определяют минимальное и максимальное значения для каждого пиксела. Допустимые значе-

ния находятся в диапазоне от 0 до 255. Чем больше значение, тем более яркие получаются рисунки.

Параметр channel определяет, к какому цветовому каналу растрового изображения будет применен шум. Можно задать значения 1, 2, 4 или 8 для красного, зеленого, синего и альфа-каналов соответственно или, во избежание опечаток, взять статические свойства класса *BitmapDataChannel*: RED, GREEN, BLUE и ALPHA,

Наконец, grayscale (шкала серого) – Булево значение. Если этот параметр равен true, то одно случайное значение применяется ко всем трем цветовым каналам, в результате получается бесцветный шум.

Вот несколько примеров. В первом создается растровое изображение, добавляется шум и полученное изображение выводится на экран:

```
bitmap = new BitmapData(stage.stageWidth,
                        stage.stageHeight,
                        false, 0xff000000);
bitmap.noise(1000, 0, 255, BitmapDataChannel.RED, false);
var image:Bitmap = new Bitmap(_bitmap);
addChild(image);
```

Здесь создается случайный шум в красном канале. Увеличение значения параметра low обеспечивает более светлый узор шума:

```
bitmap.noise(1000, 200, 255, BitmapDataChannel.RED, false);
```

Это изображение можно без труда превратить в изображение в оттенках серого, задавая последнему параметру значение true:

```
bitmap.noise(1000, 200, 255, BitmapDataChannel.RED, true);
```

Если параметру grayscale задано значение true, то уже неважно, для какого цветового канала создается шум. А для создания многоцветного шума можно комбинировать различные каналы с помощью оператора |:

```
bitmap.noise(1000, 0, 255, BitmapDataChannel.RED |
                        BitmapDataChannel.GREEN |
                        BitmapDataChannel.BLUE,
                        false);
```

Наконец, как упоминалось ранее, больше всего пользы от применения шума в сочетании с другими фильтрами или эффектами. Следующий код создает в растровом изображении некоторый шум и применяет фильтр горизонтального размытия. В результате получается широко распространенный эффект шлифованного металла:

```
bitmap = new BitmapData(stage.stageWidth, stage.stageHeight, false,
0xff000000);
bitmap.noise(1000, 128, 255, BitmapDataChannel.RED, true);
bitmap.applyFilter(bitmap,
                bitmap.rect,
```

```
                new Point(),
                new BlurFilter(30, 1, 3));
var image:Bitmap = new Bitmap(bitmap);
addChild(image);
```

Это всего лишь один пример сочетания шума с другими эффектами. Экспериментальным путем можно получить множество разнообразных результатов.

См. также

В рецептах 8.5, 8.6, 8.7, 8.8, 8.9 и 8.11 представлены другие способы добавления графического содержимого в растровое изображение.

8.11. Создание шума Перлина

Задача

Требуется создать случайные естественные эффекты, такие как облака, дым или воду.

Решение

Обратитесь к методу *perlinNoise( )* класса *BitmapData*.

Обсуждение

Как и метод *noise( )*, метод *perlinNoise( )* создает случайные узоры в растровом изображении. Однако для создания шума Перлина применяется алгоритм, в результате работы которого получаются гладкие, естественные на вид текстуры. Алгоритм был разработан Кеном Перлином (Ken Perlin) для сознания текстур в фильме «Tron». Эти текстуры идеально подходят для моделирования взрывов, дыма, воды и многих других естественных эффектов, а поскольку они генерируются алгоритмом, то требуют намного меньше памяти для хранения, чем текстуры, основанные на растровых изображениях. Метод вызывается так:

```
bitmap.perlinNoise(baseX, baseY, octaves, seed, stitch,
                   fractal, channels, grayscale, offsets);
```

Первые шесть параметров обязательные, последние три могут отсутствовать. Параметров много, поэтому создадим простой пример и посмотрим, за что отвечает каждый из них. В следующем коде создается растровое изображение, к нему применяется шум Перлина и затем полученное изображение выводится на экран:

```
bitmap = new BitmapData(stage.stageWidth, stage.stageHeight,
                                          false, 0xff000000);
bitmap.perlinNoise(100, 100, 1, 1000, false, false, 1, true, null);
var image:Bitmap = new Bitmap(bitmap);
addChild(image);
```

Добавим этот код в новый класс и запустим его, чтобы увидеть простой узор шума Перлина. Примерный результат показан на рис. 8.1. Теперь можно приступить к изменению параметров и посмотреть, к чему эти изменения приведут.

Начнем с самого начала. Параметры `baseX` и `baseY` определяют размер узора. Изменим значение `baseX` на 200 и `baseY` на 50 и посмотрим, как в результате этого наш шум растянется в горизонтальном направлении. Картинка уже немного напоминает рябь на воде (рис. 8.2).

Параметр `octaves` (октавы) – это целое число, определяющее, сколько итераций шума создается. Чем больше октав, тем более детализированный шум получается, и, конечно, тем больше время обработки.

Параметр *seed* абсолютно аналогичен параметру *seed* метода *noise( )*. Если при каждом запуске программы задавать одно и тоже значение

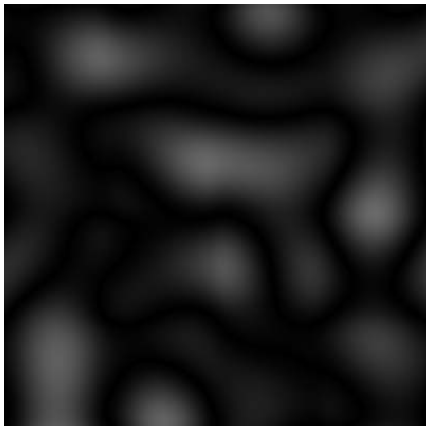


Рис. 8.1. *Простой шум Перлина*

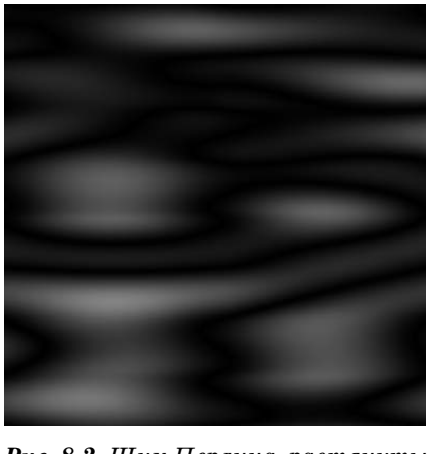


Рис. 8.2. *Шум Перлина, растянутый по оси x*

seed, получается один и тот же рисунок шума. Чтобы получать разные рисунки, надо присваивать параметру случайное значение.

Значение `true` параметра `stitch` (петля) обеспечивает согласованность левого и правого и верхнего и нижнего краев узора. Это позволяет создавать небольшие растровые изображения и заполнять ими формы без видимых швов, как при заливке растровым изображением в программном интерфейсе рисования:

```
bitmap = new BitmapData(100, 100, false, 0xff000000);
bitmap.perlinNoise(100, 100, 2, 1000, true, false, 1, true);
graphics.beginBitmapFill(bitmap);
graphics.drawRect(0, 0, stage.stageWidth, stage.stageHeight);
graphics.endFill();
```

Здесь создается растровое изображение размером 100×100 пикселов и в вызове метода *perlinNoise( )* параметру `stitch` присваивается значение `true`. Это растровое изображение используется как мозаичная заливка без видимых границ между отдельными элементами.

Если параметру `fractal` (фрактал) присвоить значение `true`, то края градиентов сглаживаются сильнее. Чтобы увидеть эффект в действии, начнем с этого кода:

```
bitmap = new BitmapData(stage.stageWidth, stage.stageHeight, false,
0xff000000);
bitmap.perlinNoise(200, 100, 5, 1000, false, false, 1, true, null);
var image:Bitmap = new Bitmap(bitmap);
addChild(image);
```

Увидев создаваемое им изображение, меняем значение `fractal` на `true`:

```
bitmap.perlinNoise(200, 100, 5, 1000, false, true, 1, true, null);
```

Заметили различия (рис. 8.3)? Этот параметр полезен для создания облаков или тумана, как видно из примера, в котором уже получилось что-то похожее на облака.

Следующие два параметра, `channel` и `grayscale`, аналогичны параметрам метода *noise( )*. Канал может иметь значение 1, 2, 4 или 8, представляющие красный, зеленый, синий и альфа-каналы соответственно. До сих пор в примерах для краткости фигурировало значение 1, но во избежание опечаток рекомендуется работать со статическими свойствами класса *BitmapDataChannel*: `RED`, `GREEN`, `BLUE` и `ALPHA`.

Конечно, если задать параметру `grayscale` значение `true`, канал уже не будет иметь значения, поскольку результирующее изображение будет черно-белым. Однако, задавая значение и один или более цветовых каналов, можно создавать цветные узоры. Следующий пример создает красные облака:

```
bitmap.perlinNoise(200, 100, 5, 1000, false, true,
                BitmapDataChannel.RED, false, null);
```

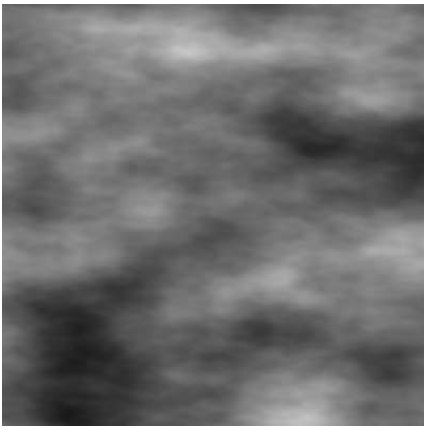


Рис. 8.3. *Шум Перлина, когда параметру fractal задано значение true*

В следующем коде за счет использования трех цветовых каналов создается многоцветный рисунок:

```
bitmap.perlinNoise(200, 100, 5, 1000, false, true,
                BitmapDataChannel.RED |
                BitmapDataChannel.GREEN |
                BitmapDataChannel.BLUE,
                false, null);
```

Также можно создать шум Перлина для альфа-канала, что пригодится для моделирования прозрачных облаков или дымки.

Последний параметр – `offsets` (смещения). Это массив объектов *Point*. Каждая точка определяет смещение октавы по осям X и Y. Если шум Перлина включает несколько октав, необходимо создать массив точек, равный по длине числу октав, и задать одну и ту же точку как значение всех элементов этого массива. В противном случае каждая октава при прокрутке будет смещаться по-разному или вообще не будет прокручиваться. Конечно, этот параметр может использоваться для создания некоторых интересных эффектов параллакса. Следующий пример показывает двухоктавный узор шума Перлина, который прокручивается по оси X:

```
package {
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.geom.Point;

    public class Clouds extends Sprite {
        private var _bitmap:BitmapData;
        private var _xoffset:int = 0;
```

```
        public function Clouds() {
            _bitmap = new BitmapData(stage.stageWidth, stage.stageHeight,
                                     true, 0xffffffff);
            var image:Bitmap = new Bitmap(_bitmap);
            addChild(image);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _xoffset++;
            var point:Point = new Point(_xoffset, 0);

            // используем одну и ту же точку
            // в обоих элементах массива смещений
            _bitmap.perlinNoise(200, 100, 2, 1000, false, true,
                                1, true, [point, point]);
        }
    }
}
```

См. также

В рецептах 8.5, 8.6, 8.7, 8.8, 8.9 и 8.10 представлены другие способы добавления графического содержимого в растровое изображение.

8.12. Использование порогового значения

Задача

Требуется изменить значение некоторых пикселов растрового изображения исходя из их текущего значения.

Решение

Для этого нужен метод *threshold( )* класса *BitmapData*.

Обсуждение

Метод *threshold( )*, вероятно, самый сложный метод *BitmapData*, но он становится довольно мощным инструментом, если понять принцип его работы. Метод использует два объекта *BitmapData*:

- `destBitmap` – растровое изображение, которое будет изменено.
- `sourceBitmap` – другое растровое изображение, передаваемое в метод как параметр. Метод использует данные пикселов этого изображения в вычислениях.

Посредством одного из шести операторов сравнения метод сравнивает каждый пиксел `sourceBitmap` с заданным значением. Если значения не совпадают, соответствующему пикселу `destBitmap` присваивается заданное значение цвета. Если значения одинаковые, пиксел или остается

неизменным, или в него копируется значение соответствующего пиксела sourceBitmap.

Вот синтаксис этого метода:

```
destBitmap.threshold(sourceBitmap,
                     sourceRect,
                     destPoint,
                     operation,
                     threshold,
                     color,
                     mask,
                     copySource)
```

Мы уже рассмотрели destBitmap и sourceBitmap. Следующий параметр, sourceRect, представляет собой экземпляр класса *flash.geom.Rectangle*. Он определяет, какая часть sourceBitmap берется для сравнения. Если требуется использовать все растровое изображение, в качестве значения этого параметра в метод передается sourceBitmap.rect.

Параметр destPoint определяет точку destBitmap, с которой начинается изменение. При наложении рисунка sourceBitmap на destBitmap его верхний левый угол располагается в этой точке. Если исходной должна быть точка с координатами (0,0), то в качестве значения этого параметра передается *new Point( )*.

Параметр operation (операция) – это одна из шести строк, эквивалентных операторам сравнения ActionScript. Это операторы <, <=, >, >=, == и !=. Например, задано значение <. В этом случае сравнение будет успешным, если значение данного пиксела меньше порогового значения, и неуспешным в противоположном случае.

Далее идет параметр threshold. Все пикселы сравниваются с этим значением. Это может быть полное 32-разрядное число, но сравнение с ним не всегда приводит к ожидаемым результатам. Причина кроется в принципе сравнения значений цветов. Например, 100-процентный красный пиксел (0xFFFF0000) в результате сравнения будет «больше, чем» 100-процентный синий (0xFF0000FF) или 100-процентный зеленый (0xFF00FF00) пиксел. Поэтому обычно для выделения определенного канала и сравнения с ним используется маска.

Параметр mask самый сложный для понимания. Он всего лишь изолирует определенный цветовой канал. Обычно сюда передается шестнадцатеричное значение, в котором цветовому каналу, который требуется заглушить, соответствуют два нуля (00), а каналу, который будет использоваться, соответствует FF. Например, значение 0x00FF0000 изолирует красный канал, значение 0xFF000000 изолирует альфа-канал. На рис. 8.4 представлена схема соответствия разрядов шестнадцатеричного значения цвета и цветовых каналов.

Следующие два параметра определяют, что происходит в результате сравнения. Параметр color – это значение цвета, которое присваивает-

***Рис. 8.4.** Структура шестнадцатеричного числа*

ся соответствующему пикселу `destBitmap` в случае выполнения условия сравнения. Параметр `copySource` определяет, что происходит, если условие сравнения не выполняется. Если задано `true`, значение соответствующего пиксела `sourceBitmap` копируется в `destBitmap`. Если задано `false`, пиксел `destBitmap` не изменяется.

Теперь рассмотрим несколько примеров. Возьмем фрагмент кода, в котором создаются исходное и целевое растровые изображения, в исходном изображении формируется шум Перлина. Затем целевое изображение добавляется в список отображения и к нему применяется метод *threshold( )* с использованием исходного изображения:

```
var srcBmp:BitmapData = new BitmapData(stage.stageWidth,
                                       stage.stageHeight,
                                       true, 0xffffffff);
srcBmp.perlinNoise(200, 100, 2, 1000, false, true, 1, true);

var destBmp:BitmapData = new BitmapData(stage.stageWidth, stage.stageHeight,
true, 0xffffffff);
var image:Bitmap = new Bitmap(destBmp);
addChild(image);
destBmp.threshold(srcBmp,              // sourceBitmap
                  srcBmp.rect,         // sourceRectangle
                  new Point(),         // destPoint
                  "<",                 // operator
                  0x00880000,          // threshold
                  0x00000000,          // color
                  0x00ff0000,          // mask
                  true);               // copySource
```

В данном случае метод сравнивает красный канал (как определено маской) пиксела со значением `0x00880000`. Если значение пиксела меньше заданного, пиксел становится прозрачным. Если нет, копируется значение пиксела исходного изображения. Поскольку шум Перлина был создан в оттенках серого, не имеет значения, какой из трех цветовых каналов (красный, зеленый или синий) задействуется. Если в качестве исходного задано полноцветное изображение, можно поэкспериментировать с разными цветовыми каналами и посмотреть, какой из них обеспечивает желаемый эффект:

- Как видите, в результате выполнения данного кода все темные области шаблона шума Перлина становятся прозрачными.
- Изменение значения параметра `operator` на «`>`» приводит к противоположному эффекту: выбираются все более светлые пикселы.

- Увеличивая или уменьшая значение `threshold`, например задавая `0x00330000` или `0x00AA0000`, можно выбрать большее или меньшее количество пикселов.
- Попытайтесь задать параметру `color` другое значение и посмотрите, к чему это приведет. Также измените значение `copySource` на `false` и увидите, что пикселы не копируются, сохраняются их исходные значения.

Еще хотелось бы отметить, что ничто не мешает взять в качестве исходного и целевого изображений одно и то же растровое изображение. В качестве порогового может выступать значение любого пиксела растрового изображения. Однако следует отдавать себе отчет, что изображение меняется навсегда, уже нельзя будет повторить операцию и вернуть все в исходное состояние.

Как и для многих методов класса *BitmapData*, вся мощь *threshold( )* проявляется, если он применяется в сочетании с другими. Например, если в предыдущий пример добавить всего одну строку кода, возникает тень, и узор превращается в трехмерное изображение:

```
destBmp.applyFilter(destBmp, destBmp.rect,
                    new Point(), new DropShadowFilter());
```

См. также

Другие способы изменения содержимого растрового изображения представлены в рецептах 8.13, 8.14 и 8.15.

8.13. Применение фильтра к растровому изображению

Задача

Требуется применить фильтр к *BitmapData*.

Решение

Возьмите метод *applyFilter( )* класса *BitmapData* или добавьте фильтр в массив фильтров объекта *Bitmap, который служит для отображения растра*.

Обсуждение

Как уже говорилось, существует два метода применения фильтра к растровому изображению. Первый – при помощи метода *applyFilter( )* непосредственно к самому *BitmapData*. Как и некоторые другие методы класса *BitmapData*, этот метод применяется к тому объекту *BitmapData*, который является целевым изображением, и может принимать другой объект *BitmapData* в качестве исходного растрового изображения:

```
destBmp.applyFilter(srcBmp, sourceRect, destPoint, filter);
```

где:

- `srcBmp` – растровое изображение, к которому применяется фильтр.
- `sourceRect` – определенная прямоугольная область исходного изображения, к которой применяется фильтр.
- Параметр `destPoint` определяет верхнюю левую точку в целевом изображении, от которой будут изменяться пикселы.
- `filter` – это, конечно, экземпляр класса *BitmapFilter*, который будет применен.

Конечно, изображение может быть одновременно и исходным, и целевым. Но такой подход считается «деструктивным», – не потому что он плохой, а потому, что навсегда изменяет растровое изображение.

Значение каждого пиксела вычисляется на основании его исходного содержимого и параметров фильтра. Полученное значение заменяет исходное. Это полезно в нескольких случаях:

- Если гарантированно никогда не потребуется изменять содержимое растрового изображения или сам фильтр.
- Если фильтр применяется многократно или повторно.

Например, следующий код делает 10 случайных пикселов вокруг курсора мыши белыми и затем применяет стандартный фильтр размытия. Это повторяется в каждом кадре, поэтому чем дольше курсор остается в одной точке, тем ярче она становится. Однако если курсор мыши перемещается, размытие многократно применяется к белым пикселам, в конце концов они исчезают. В результате получается светящийся след.

```
package {
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.filters.BlurFilter;
    import flash.events.Event;
    import flash.geom.Point;

    public class FilteredBitmap extends Sprite {
        private var _bitmap:BitmapData;
        private var _image:Bitmap;
        private var _blurFilter:BlurFilter;

        public function FilteredBitmap() {
            _bitmap = new BitmapData(stage.stageWidth, stage.stageHeight,
                                     false, 0xff000000);
            _image = new Bitmap(_bitmap);
            addChild(_image);
            _blurFilter = new BlurFilter();
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
```

```
        }

        public function onEnterFrame(event:Event):void {
            for(var i:int = 0; i < 100; i++) {
                _bitmap.setPixel(mouseX + Math.random() * 20 - 10,
                                 mouseY + Math.random() * 20 - 10,
                                 0xffffffff);
            }
            _bitmap.applyFilter(_bitmap, _bitmap.rect, new Point(),
                                        _blurFilter);
        }
    }
}
```

Как видите, деструктивный метод может быть созидательным. Сравним с недеструктивным методом, который задействует два разных растровых изображения. Одно – исходное, другое – целевое. Метод применяет фильтр к данным исходного растрового изображения и помещает результат в целевое изображение. Здесь нет ничего деструктивного, фильтр не меняет оригинальные пикселы исходного изображения.

```
package {
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.filters.BlurFilter;
    import flash.events.Event;
    import flash.geom.Point;

    public class FilteredBitmap2 extends Sprite {
        private var _bitmap:BitmapData;
        private var _bitmap2:BitmapData;
        private var _image:Bitmap;
        private var _blurFilter:BlurFilter;

        public function FilteredBitmap2() {
            _bitmap = new BitmapData(stage.stageWidth, stage.stageHeight,
                                     false, 0xff000000);
            _bitmap2 = new BitmapData(stage.stageWidth, stage.stageHeight,
                                      false, 0xff000000);
            _image = new Bitmap(_bitmap);
            addChild(_image);
            _blurFilter = new BlurFilter();
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            for(var i:int = 0; i < 100; i++) {
            _bitmap2.setPixel(mouseX + Math.random() * 20 - 10,
                              mouseY + Math.random() * 20 - 10,
                             0xffffffff);
            }
            _bitmap.applyFilter(_bitmap2, _bitmap.rect, new Point(),
```

```
                                        _blurFilter);
            }
        }
    }
```

Другой способ применения фильтра к растровому изображению – добавить его в свойство *filters* объекта *Bitmap*, заключающего в себе объект *BitmapData*. Это еще один недеструктивный способ применения фильтра, поскольку данные пикселов *BitmapData* остаются нетронутыми. Фильтр применяется только к объекту *Bitmap*, являющемуся оболочкой *BitmapData*. Следующий код показывает эту методику в действии:

```
package {
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.filters.BlurFilter;
    import flash.events.Event;

    public class FilteredBitmap3 extends Sprite {
        private var _bitmap:BitmapData;
        private var _image:Bitmap;

        public function FilteredBitmap3() {
            _bitmap = new BitmapData(stage.stageWidth, stage.stageHeight,
                                     false, 0xff000000);
            _image = new Bitmap(_bitmap);
            addChild(_image);
            _image.filters = [new BlurFilter()];
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            for(var i:int = 0; i < 100; i++) {
                _bitmap.setPixel(mouseX + Math.random() * 20 - 10,
                                 mouseY + Math.random() * 20 - 10,
                                 0xffffffff);
            }
        }
    }
}
```

Здесь можно увидеть, что фильтр применяется только один раз. Отрисованные пикселы немного размыты, но они не размываются до полного исчезновения. Если в какой-то момент удалить фильтр из *Bitmap*, белые пикселы останутся абсолютно четкими, поскольку размытие применяется только к контейнеру, а не к самим данным растрового изображения.

См. также

Другие способы изменения содержимого растрового изображения представлены в рецептах 8.12, 8.14 и 8.15.

8.14. Растворение одного растрового изображения в другом

Задача

Требуется растворить одно изображение в другом.

Решение

Задача решается при помощи метода *pixelDissolve( )* класса *BitmapData*.

Обсуждение

Как многие методы *BitmapData*, *pixelDissolve( )* использует исходное и целевое растровые изображения. При каждом его вызове заданное число случайных пикселов копируется из исходного изображения в целевое. Чтобы достичь полного «растворения», метод вызывается многократно или в обработчике события *enterFrame*, или в управляемой таймером функции. Об анимации, основанной на событиях таймера, рассказывается в рецепте 11.1.

При первом вызове в метод *pixelDissolve( )* передается случайное число, выступающее в качестве зерна для первого раунда копирования пикселов. Метод возвращает число, которое необходимо отслеживать и использовать как зерно в следующей итерации.

Синтаксис метода:

```
seed = srcBmp.pixelDissolve(destBmp, sourceRect, destPoint, seed,
                            numPixels, fillColor);
```

- Парметры *sourceRect* и *destPoint* аналогичны этим параметрам в других методах *BitmapData*, которые используют два растровых изображения. Параметр *seed* подробно рассмотрен в рецепте 8.10.
- Параметр *numPixels* определяет, сколько пикселов копируется при каждом вызове метода.
- Параметр *fillColor* предлагает простой способ изменения цвета отдельного пиксела на заданный. Если один объект *BitmapData* выступает в качестве исходного и целевого изображений одновременно, то идентичные пикселы не копируются (что не имеет никакого визуального эффекта), им просто присваивается цвет заливки, заданный этим параметром. Параметр можно проигнорировать при работе с двумя разными растровыми изображениями.

В первом примере фигурируют два растровых изображения – белое и черное – и происходит растворение по 1000 пикселов в каждом кадре:

```
package {
    import flash.display.Sprite;
```

```
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.geom.Point;

    public class Dissolve extends Sprite {
        private var _bitmap:BitmapData;
        private var _bitmap2:BitmapData;
        private var _image:Bitmap;
        private var _seed:Number;
        private var _pixelCount:int = 0;

        public function Dissolve() {
            _bitmap = new BitmapData(stage.stageWidth,
                                     stage.stageHeight,
                                     false, 0xffffffff);
            _bitmap2 = new BitmapData(stage.stageWidth,
                                      stage.stageHeight,
                                      false, 0xff000000);
            _image = new Bitmap(_bitmap);
            addChild(_image);
            _seed = Math.random() * 100000;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _seed = _bitmap.pixelDissolve(_bitmap2,
                                          _bitmap.rect,
                                          new Point(),
                                          _seed,
                                          1000);
            _pixelCount += 1000;
            if(_pixelCount > _bitmap.width * _bitmap.height) {
                removeEventListener(Event.ENTER_FRAME,
                                    onEnterFrame);
            }
        }
    }
}
```

Важно заметить, что в методе *pixelDissolve( )* нет средства определения завершения операции (т. е. момента, когда все пикселы одного изображения скопированы в другое). Отслеживать количество скопированных пикселов и останавливать процесс копирования, когда это количество превышает общее число пикселов в изображении (ширину умножить на высоту), должен программист.

В следующем примере один и тот же объект *BitmapData* играет роль исходного и целевого изображений и применяется черный цвет заливки:

```
package {
    import flash.display.Sprite;
    import flash.display.Bitmap;
    import flash.display.BitmapData;
```

```
    import flash.events.Event;
    import flash.geom.Point;

    public class Dissolve2 extends Sprite {
        private var _bitmap:BitmapData;
        private var _image:Bitmap;
        private var _seed:Number;
        private var _pixelCount:int = 0;

        public function Dissolve2() {
            _bitmap = new BitmapData(stage.stageWidth,
                                     stage.stageHeight,
                                     false,
                                     0xffffffff);
            _image = new Bitmap(_bitmap);
            addChild(_image);
            _seed = Math.random() * 100000;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _seed = _bitmap.pixelDissolve(_bitmap,
                                          _bitmap.rect,
                                          new Point(),
                                          _seed,
                                          1000,
                                          0xff000000);
            _pixelCount += 1000;
            if(_pixelCount > _bitmap.width * _bitmap.height)
            {
                removeEventListener(Event.ENTER_FRAME, onEnterFrame);
            }
        }
    }
}
```

Особенно эффективно растворяются две фотографии при большой частоте смены кадров. Если размер изображений заранее неизвестен, но требуется поддерживать постоянную скорость растворения, можно задать `numPixels` в процентном соотношении от размера изображения, как это сделано в следующем примере:

```
var numPixels:Number = _bitmap.width * _bitmap.height / 100;
_seed = _bitmap.pixelDissolve(_bitmap,
                              _bitmap.rect,
                              new Point(),
                              _seed,
                              numPixels,
                              0xff000000);
_pixelCount += numPixels;
```

Таким образом, в каждой итерации копируется 0,01 всех пикселов. При частоте 30 кадров в секунду растворение заняло бы (примерно) 3 секунды независимо от размера изображений.

См. также

Другие способы изменения содержимого растрового изображения представлены в рецептах 8.12, 8.13 и 8.15.

8.15. Прокрутка растрового изображения

Задача

Требуется прокручивать содержимое *BitmapData*.

Решение

Возьмите метод *scroll( )* класса *BitmapData*.

Обсуждение

Операция и синтаксис просты. В метод передается параметр, указывающий, на сколько пикселов требуется прокрутить содержимое растрового изображения вдоль осей X и Y:

```
_bitmap.scroll(xAmount, yAmount);
```

Метод копирует пикселы растрового изображения и вставляет их, смещая на заданное количество пикселов. Пикселы, не затронутые данной операцией, остаются неизменными.

Метод можно многократно вызывать в обработчике или срабатывающей по таймеру функции, чтобы анимировать прокрутку растрового изображения. Следующий код демонстрирует это. В нем генерируется и затем прокручивается узор шума Перлина:

```
public function Scroll() {
    _bitmap = new BitmapData(stage.stageWidth, stage.stageHeight,
                        false, 0xffffffff);
    _bitmap.perlinNoise(100, 100, 3, 1000, true, true, 1, true);
    _image = new Bitmap(_bitmap);
    addChild(_image);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

public function onEnterFrame(event:Event):void {
    _bitmap.scroll(-1, -1);
}
```

Обратите внимание, что при прокрутке края не копируются и не очищаются, они остаются нетронутыми.

См. также

Другие способы изменения содержимого растрового изображения представлены в рецептах 8.12, 8.13 и 8.14.

9

Текст

9.0. Введение

За отображение любого текста во Flash Player отвечает класс *flash.text.TextField*. Даже такие текстовые компоненты, как *TextArea* и *TextInput*, для вывода текста задействуют класс *TextField*. Flash Player предоставляет широкие функциональные возможности для текстовых полей, начиная от простого ввода текста пользователем до встраивания шрифтов и форматирования текста с помощью каскадных таблиц стилей (Cascading Style Sheets, CSS). В данной главе обсуждается все то, что можно сделать с текстом.

Как следует из предыдущего абзаца, класс *TextField* является частью пакета *flash.text*. Поэтому либо он должен быть импортирован, либо на него необходимо ссылаться с помощью полного имени класса. Во всех примерах данной главы предполагается, что класс импортирован:

```
import flash.text.TextField;
```

В ActionScript 3.0 текстовое поле создается на основе списка отображения, что сильно отличается от предыдущих версий этого языка. Там текстовое поле создавалось так:

```
var field:TextField = createTextField(name, depth);
```

Однако в ActionScript 3.0 новое текстовое поле не добавляется в список отображения автоматически. То есть если текстовое поле должно быть видимым, необходимо применять метод *addChild( )*. Как говорилось в главе 6, метод *addChild( )* определен для всех контейнеров отображаемых объектов, таких как *Sprite*, и добавляет переданный в качестве параметра объект в список отображения объекта, из которого был вызван. Например, следующая строка кода *TextField* добавляет объект в список отображения экземпляра класса *TextExample*:

```
package {
  import flash.display.Sprite;
  import flash.text.TextField;
  public class TextExample extends Sprite {
    public function TextExample() {
      var field:TextField = new TextField();
      addChild(field);
    }
  }
}
```

Если в примерах данной главы встречается объект с именем `field` (поле), обычно предполагается, что это объект *TextField*, который был создан посредством конструктора *TextField* и добавлен в список отображения методом *addChild( )*.

9.1. Создание контура вокруг текстового поля

Задача

Требуется создать рамку вокруг текстового поля.

Решение

Присвойте свойству `border` текстового поля значение `true`. Кроме того, посредством свойства `borderColor` можно задавать цвет рамки.

Обсуждение

По умолчанию у текстового поля нет видимой рамки; такое отображение текста считается предпочтительным (разве нужна рамка вокруг надписи на элементе?). Однако нередко требуется заключить текстовое поле в рамку, например, если пользователь должен ввести в поле некоторые данные, рамка покажет ему место для ввода значения. Чтобы рамка отображалась вокруг текстового поля, достаточно задать его свойству `border` (рамка) значение `true`:

```
field.border = true;
```

Чтобы отключить рамку, присвойте свойству `border` значение `false`.

По умолчанию рамка имеет черный цвет, но это можно изменить с помощью свойства `borderColor` (цвет рамки), которое принимает RGB-значение необходимого цвета:

```
field.borderColor = 0xFF00FF;  // Делаем рамку фиолетовой.
```

9.2. Создание фона в текстовом поле

Задача

Требуется создать в текстовом поле видимый фон, на котором будет отображаться текст.

Решение

Задайте свойству background текстового поля значение true. Кроме того, цвет фона можно менять, задавая значение свойства backgroundColor.

Обсуждение

По умолчанию в текстовом поле нет видимого фона. Однако это можно изменить, задавая свойству background (фон) этого объекта значение true:

```
field.background = true;
```

По умолчанию фон текстового поля, если он видимый, имеет белый цвет. Однако, задавая значение свойству backgroundColor (цвет фона) объекта, которое принимает соответствующее RGB-значение цвета, можно менять цвет фона, как показано в примере:

```
field.backgroundColor = 0x00FFFF; // Делаем фон голубым
```

9.3. Создание текстового поля ввода

Задача

Требуется создать поле ввода, в которое пользователь мог бы вводить текст.

Решение

Задайте свойству type текстового поля значение *TextFieldType.INPUT*.

Обсуждение

Существует два типа текстовых полей: динамические и поля ввода. По умолчанию текстовые поля динамические. Это означает, что полем можно управлять посредством ActionScript, но пользователь не может вводить в него текст. Чтобы обеспечить возможность пользовательского ввода, свойству type (тип) поля в качестве значения задается константа INPUT (ввод) класса *flash.text.TextFieldType*:

```
field.type = TextFieldType.INPUT;
```

Хотя это не обязательное требование, поля ввода обычно имеют рамку и фон. Если их не будет, пользователю может быть трудно найти и выбрать это поле:

```
field.border = true;
field.background = true;
```

Чтобы пользователь мог вводить текст, свойство selectable (выделяемое) поля должно иметь значение true (это значение по умолчанию). Не надо присваивать свойству selectable значение true, если оно не подвергалось изменениям и ему не присваивалось значение false.

Если требуется сделать поле, ранее объявленное как поле ввода, динамическим текстовым полем (чтобы пользователь не мог вводить в него свои данные), его свойству `type` задается значение `DYNAMIC`, которое представляет собой константу класса *flash.text.TextFieldType*:

```
field.type = TextFieldType.DYNAMIC;
```

9.4. Создание поля для ввода пароля

Задача

Требуется создать текстовое поле в стиле поля для ввода пароля, которое заменяет вводимые символы звездочками. Также можно запретить копирование из этого поля.

Решение

Задайте свойству `password` текстового поля значение `true`.

Обсуждение

Когда пользователь вводит пароль, желательно, чтобы никто другой не мог его прочитать. Это основная мера предосторожности для обеспечения безопасности. При вводе пароля принято отображать в текстовом поле не буквы, а звездочки. Таким образом, пользователь может видеть, что он успешно вводит значение, но никто другой не может прочитать, что было введено.

Чтобы создать поле ввода, которое автоматически маскирует символы звездочками, достаточно задать свойству *TextField.password* значение `true`:

```
field.password = true;
```

Если свойству `password` задано значение `true`, весь вводимый (или программно или пользователем) в текстовое поле текст отображается в виде звездочек:

```
field.password = true;
field.text = "пример текста";  // В текстовом поле отображается: ************
```

В текстовом поле для ввода пароля также отключена возможность копирования из него. Это не дает пользователю копировать пароли или подобные данные и вставлять их в простые текстовые документы.

Текстовые поля паролей чаще всего являются полями ввода, однако задать значение `true` свойству `password` можно и для динамического текстового поля.

Текстовые поля для ввода паролей скрывают данные на экране, но они не шифруют их и не делают более безопасными для передачи по сети. Если для приложения важна безопасность дан-

ных, то необходимо задействовать соответствующие технологии. Передачу данных по стандартному HTTP-протоколу безопасной не назовешь. Для обеспечения безопасности данных при передаче по сети применяются такие технологии, как SSL.[1]

9.5. Фильтрация текстового ввода

Задача

Требуется ограничить множество символов, которые пользователь может вводить в поле ввода.

Решение

Задача решается с помощью свойства `restrict` текстового поля.

Обсуждение

По умолчанию пользователь может вводить в поле ввода любые символы. Однако часто множество допустимых символов требуется ограничить. Например, в текстовом поле для ввода телефонных номеров желательно ограничить допустимые символы только числами и тире.

Свойство *TextField.restrict* (ограничить) обеспечивает возможность ограничить допустимые символы для текстового поля ввода, задавая строку разрешенных символов:

```
field.restrict = "abcdefg";
```

Этот код разрешает пользователю вводить любой из допустимых символов: *a*, *b*, *c*, *d*, *e*, *f* или *g*. Все остальные символы запрещены. Если пользователь попытается ввести *grabs*, на экране появится только *gab*, поскольку буквы *r* и *s* не входят в набор допустимых символов.

Если в качестве значения свойства `restrict` задана пустая строка, разрешается вводить *все* символы. Чтобы полностью запретить ввод, задайте свойству type значение DYNAMIC.

Также обратите внимание, что ActionScript различает прописные и строчные символы. Иначе говоря, есть разница между *a* и *A*. Если свойству `restrict` задано *abcdefg*, при вводе прописных эквивалентов допустимых символов (например, *A*, *B*, *C*) они будут переводиться в нижний регистр (*a*, *b*, *c*). Справедливо и обратное: при вводе символов в нижнем регистре, когда допускаются только их эквиваленты в верхнем регистре, все символы переводятся в верхний регистр.[2]

---

[1] SSL (Secure Socket Layer) – протокол защищенных сокетов. – *Примеч. перев.*

[2] Это верно только для символов латинского алфавита. – *Примеч. науч. ред.*

Свойство `restrict` поддерживает некоторые шаблоны, подобные регулярным выражениям. Таким образом, можно вводить диапазоны символов, указывая первый и последний символы через тире (–):

```
field.restrict = "a-zA-Z";   // Допускаются только буквы в верхнем
                             // и нижнем регистре
field.restrict = "a-zA-Z ";  // Допускаются только буквы и пробелы
field.restrict = "0-9";      // Допускаются только числа
```

Можно не только указать допустимые символы, но и некоторые символы. Для этого строку значения свойства `restrict` начинают с символа вставки (^):

Все символы и диапазоны символов, стоящие в строке `restrict` после символа вставки, запрещены. Например:

```
field.restrict = "^abcdefg"; // Допускаются все символы,
                             // кроме строчных от a до g
field.restrict = "^a-z";     // Запрещены все строчные
                             // буквы (но разрешены другие
                             // символы, включая прописные буквы.)
field.restrict = "0-9^5";    // Допускаются только цифры,
                             // кроме 5
```

Также допустимые символы можно задавать в кодировке Unicode. Например, для того чтобы запретить ввод символа > (Control-Z), в свойстве `restrict` можно задать его Unicode-код:

```
field.restrict = "^\u001A";
```

Если в свойстве `restrict` указывается литеральный символ, имеющий специальное значение (например, тире или знак вставки), он должен быть *экранирован* (*escaped*) с помощью двух обратных слэшей (не одного), как показано здесь:

```
field.restrict = "0-9\\-";      // Допускаются цифры и тире
field.restrict = "0-9\\^";      // Допускаются цифры и знаки вставки
```

Если обратный слэш выступает в качестве литерального символа, то перед ним ставятся три обратных слэша, всего в строке получается четыре слэша подряд:

```
field.restrict = "0-9\\\\";     // Допускаются цифры и обратный слэш
```

Свойство `restrict` распространяется только на символы, которые могут быть введены пользователем. Оно не оказывает никакого влияния на те символы, которые выводятся на экран программой.

9.6. Задание максимальной длины поля

Задача

Требуется ограничить длину строки, вводимой в текстовое поле.

Решение

Обратитесь к свойству текстового поля `maxChars`.

Обсуждение

По умолчанию пользователь может ввести в текстовое поле сколько угодно символов. Однако нередко возникают веские основания ограничивать длину вводимого текста. Например, если в текстовое поле пользователь должен ввести двухбуквенный код страны, вероятно, не лишним будет предотвратить возможность ввода более двух символов. Количество допустимых символов в свойстве `maxChars` задается так:

```
field.maxChars = 6;  // можно ввести максимум 6 символов
```

Если ранее свойству `maxChars` было задано некоторое определенное значение, задав ему значение `null`, можно разрешить вводить текст неограниченной длины.

См. также

Рецепт 9.5.

9.7. Вывод текста на экран

Задача

Требуется вывести на экран текст в текстовом поле.

Решение

Обратитесь к свойству текстового поля `text`.

Обсуждение

Текстовые поля применяются не только как поля ввода, но и для вывода текста на экран. В поле отображается значение свойства `text` (текст) текстового поля:

```
field.text = "этот текст будет отображен в поле";
```

В текстовой строке могут присутствовать специальные символы, такие как `\t` (табуляция) и `\n` (начало новой строки).

Дописать текст в конец строки можно посредством оператора `+=` или метода *appendText( )*:

```
field.appendText("новый текст");
```

См. также

Информация о поддержке текста, форматированного тегами HTML, представлена в рецепте 9.8.

9.8. Вывод на экран текста, форматированного тегами HTML

Задача

Требуется вывести в текстовом поле HTML-содержимое.

Решение

Задайте в качестве значения свойства `htmlText` HTML-содержимое, которое должно быть выведено на экран.

Обсуждение

При соответствующей настройке текстовые поля могут интерпретировать и выводить основные теги HTML. Это удобный способ ввода гиперссылок и простого форматирования, например задания цвета шрифта и выделения текста полужирным шрифтом.

Значение свойства `htmlText` текстового поля интерпретируется как HTML:

```
field.htmlText = "<u>Этот текст отображается с подчеркиванием.</u>";
```

Независимо от содержащихся в нем символов значение свойства `text` текстового поля выводится как неформатированный текст. Это значит, что даже если в свойстве `text` записано `<u>проверка</u>`, объект отображает `<u>проверка</u>`, а не `проверка`. Таким образом, если требуется просто вывести на экран код HTML, его надо записать как значение свойства `text` текстового поля:

```
field.text = "<u>текст с подчеркиванием</u>";

/* в текстовом поле выводится:
<u>текст с подчеркиванием</u>
*/
```

Это может пригодиться, когда в расположенных рядом текстовых полях требуется показать и сгенерированный HTML, и исходный HTML-код:

```
htmlCode = "<i>курсив</i>";
sourceHTML.text = htmlCode;
renderedHTML.htmlText = htmlCode;
```

Нельзя вывести и сгенерированный и несгенерированный HTML в одном текстовом поле. Попытки сделать это приводят к неустойчивым результатам.

К поддерживаемым текстовыми полями тегам HTML относятся: `<b>`, `<i>`, `<u>`, `<font>` (с атрибутами `face`, `size` и `color`), `<p>`, `<br>`, `<a>`, `<li>`, `<img>` и `<textformat>` (с атрибутами `leftmargin`, `rightmargin`, `blockindent`, `indent`, `leading` и `tabstops`, соответствующими свойствам класса *TextFormat* с такими же именами).

9.9. Сжатие пробелов

Задача

Требуется сократить расстояние между словами при отображении HTML-содержимого текстового поля.

Решение

Задайте свойству `condenseWhite` значение `true`.

Обсуждение

Когда в текстовое поле помещается HTML, необязательное свойство `condenseWhite` задает уменьшение количества пробелов между словами до одного, как это делается в большинстве HTML-броузеров. Например, в веб-броузере следующий текст будет сгенерирован только с одним пробелом между словами «здравствуй» и «друг», несмотря на то, что в исходном коде слова разделены несколькими пробелами:

```
здравствуй          друг
```

Однако в текстовых полях ActionScript пробелы сокращаются, только если свойству `condenseWhite` задано значение `true`:

```
field.condenseWhite = true;
field.htmlText = "здравствуй           друг";   // На экран выводится:
                                                // "здравствуй друг"
```

Свойство `condenseWhite` действует, только если свойству `html` задано значение `true`.

См. также

Рецепт 9.8.

9.10. Изменение размера текстового поля с целью вмещения содержимого

Задача

Требуется изменить размер видимой области текстового поля, чтобы вместить весь содержащийся в нем текст.

Решение

Обратитесь к свойству `autoSize`.

Обсуждение

Задавая свойство `autoSize` текстового поля, можно определить, как оно меняет размер, чтобы вместить содержимое. Возможные значения –

константы класса *flash.text.TextFieldAutoSize*: RIGHT, LEFT, CENTER и NONE. По умолчанию – NONE, т. е. размер текстового поля не меняется автоматически.

Если требуется, чтобы текстовое поле изменяло размер, фиксируя при этом положение верхнего левого угла, задается значение LEFT. Иначе говоря, нижний правый угол является точкой, которая перемещается при изменении размера текстового поля:

```
// Эти две строки делают одно и то же
field.autoSize = TextFieldAutoSize.LEFT;
field.autoSize = true;
```

Если требуется зафиксировать текстовое поле в центре, задается значение CENTER. Верхняя часть объекта остается неподвижной, текстовое поле меняет размер вниз и в равной степени вправо и влево:

```
field.autoSize = TextFieldAutoSize.CENTER;
```

Если требуется сохранить положение верхнего правого угла текстового поля и изменять размер в направлении нижнего левого угла, задается значение RIGHT:

```
field.autoSize = TextFieldAutoSize.RIGHT;
```

Если свойству *wordWrap* (переносить слова) задано значение false (значение по умолчанию), текстовое поле расширяется в горизонтальном направлении. В этом случае текстовое поле расширяется в вертикальном направлении, только если в выводимом тексте встречаются символы перевода строки. Следующий пример иллюстрирует текстовое поле, автоматически меняющее размер, чтобы вместить весь текст в одну строку:

```
var field:TextField = new TextField();
field.autoSize = TextFieldAutoSize.LEFT;
field.text = "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Morbi
tortor purus, aliquet a, ornare ac, suscipit a, est. Nullam hendrerit
molestie erat. Nunc nulla tortor, ullamcorper et, elementum vel, fringilla
sed, dui. Praesent fermentum interdum orci.";
addChild(field);
```

Следующий код добавляет символ новой строки в выводимый текст, поэтому текстовое поле автоматически меняет размер, чтобы вместить весь текст в две строки:

```
var field:TextField = new TextField();
field.autoSize = TextFieldAutoSize.LEFT;
field.text = "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Morbi
tortor purus, aliquet a, ornare ac, suscipit a, est.";
field.text += "\n";
field.text += "Nullam hendrerit molestie erat. Nunc nulla tortor, ullamcorper
et, elementum vel, fringilla sed, dui. Praesent fermentum interdum orci.";
addChild(field);
```

Когда свойству *wordWrap* задано значение `true`, текстовое поле не расширяется больше заданной ширины (свойство *width*), по умолчанию равно 100. Если свойству *autoSize* задано значение `RIGHT`, `LEFT` или `CENTER`, текст автоматически переносится на новую строку:

```
var field:TextField = new TextField();
field.autoSize = TextFieldAutoSize.LEFT;
field.wordWrap = true;
field.text = "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Morbi
tortor purus, aliquet a, ornare ac, suscipit a, est. Nullam hendrerit
molestie erat. Nunc nulla tortor, ullamcorper et, elementum vel, fringilla
sed, dui. Praesent fermentum interdum orci.";
addChild(field);
```

См. также

Рецепт 9.6.

9.11. Реализация прокрутки текста в программе

Задача

Требуется обеспечить прокрутку текста в текстовом поле посредством ActionScript.

Решение

Обратитесь к свойствам `scrollV`, `maxScrollV`, `bottomScrollV`, `scrollH` и `maxScrollH` текстового поля. С помощью свойства `mouseWheelEnabled` обеспечьте возможность прокрутки текста посредством колеса мыши.

Обсуждение

Управлять прокруткой текста в текстовом поле можно и без полосы прокрутки, средствами ActionScript. Например, требуется автоматически прокручивать содержимое текстового поля для отображения слова или выбранной фразы в тексте. Некоторые встроенные свойства позволяют программно управлять прокруткой текстового поля как по вертикали, так и по горизонтали. Вертикальной прокруткой управляют свойства `scrollV`, `maxScrollV` и `bottomScrollV`. Свойства `scrollH` и `maxScrollH` предназначены для горизонтальной прокрутки.

В каждом текстовом поле содержится некоторое количество строк (одна или сто – неважно). Каждая из них идентифицируется числом, начиная с 1. Некоторые из строк могут находиться в рамках текстового поля и быть видимыми, некоторые – нет. Поэтому, чтобы просмотреть строки, находящиеся за границами видимой части текстового поля, его содержимое необходимо прокрутить. Это иллюстрирует рис. 9.1. На нем изображено текстовое поле, Сплошной линией обозначены границы объекта (видимая область), пунктирной линией обведен весь ос-

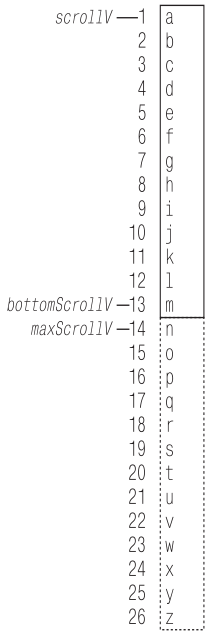


***Рис. 9.1.** Как работают scrollV, bottomScrollV и maxScrollV*

тальной текст, находящийся в объекте, но выходящий за рамки видимой области. Слева от текстового поля указаны номера строк текста. Три метки – `scrollV`, `bottomScrollV` и `maxScrollV` – показывают смысл соответствующих свойств текстового поля.

Свойство `scrollV` доступно для записи и чтения и указывает на верхнюю строку видимой области текстового поля. На рис. 9.1 значение `scrollV` равно 1. Чтобы прокрутить содержимое текстового поля, свойству `scrollV` необходимо присвоить номер новой строки. Если задать значение 6, то содержимое текстового поля будет прокручено вверх так, что строка под номером 6 станет самой верхней из отображаемых строк объекта. Значением свойства `scrollV` может быть только целое число, Flash не может прокручивать и отображать строку частично:

```
field.scrollV = 1;   // Прокрутка вверх
field.scrollV += 1;  // Прокрутка к следующей строке
field.scrollV = 6;   // Прокрутка к строке 6
```

С помощью свойства `bottomScrollV`, которое обозначает самую нижнюю видимую строку, можно перейти на следующую страницу содержимого текстового поля. Хотя значение `bottomScrollV` нельзя задать, по нему можно определить новое значение `scrollV`. На рис. 9.1 `bottomScrollV` имеет значение 13. Если `scrollV` задать значение 6, то `bottomScrollV` автоматически станет равно 18.

```
// Прокрутить текст до следующей страницы, сделав
// нижнюю строку предыдущей страницы первой
field.scrollV = field.bottomScrollV;
// Прокрутить текст до следующей страницы, не отображая
// нижнюю строку предыдущей страницы
field.scrollV = field.bottomScrollV + 1;
```

Для перехода на последнюю страницу содержимого текстового поля применяется свойство maxScrollV. Оно также доступно только для чтения. Это свойство содержит максимальное значение номера строки, которое может быть присвоено свойству scrollV. Поэтому значение свойства maxScrollV меняется только при изменении количества строк в текстовом поле (в результате ввода пользователем или задания в коде ActionScript). На рис. 9.1 значение maxScrollV равно 14, потому что в текстовом поле всего 26 строк и 13 из них видимые. Если scrollV задано значение 14, то последняя видимая строка имеет номер 26 (последняя строка текстового поля).

Не пытайтесь присвоить scrollV значение меньше 1 или больше maxScrollV. Это не приведет к ошибке, но прокрутить текст за рамки содержимого тоже не получится. Искусственно увеличить диапазон прокрутки можно, добавляя пустые строки в начало или конец содержимого текстового поля:

```
field.scrollV = field.maxScrollV;    // Прокручиваем вниз
```

Свойства вертикальной прокрутки ориентируются на номера строк, тогда как значения свойств горизонтальной прокрутки (scrollH и maxScrollH) задаются в пикселах. Во всем остальном scrollH и maxScrollH более или менее подобны scrollV и maxScrollV (среди свойств горизонтальной прокрутки нет аналога свойству bottomScrollV). Свойство scrollH доступно для чтения и записи, что позволяет управлять номером крайнего слева видимого пиксела, начиная с 0. Свойство maxScrollH доступно только для чтения и показывает максимальное значение в пикселах, которое может быть присвоено scrollH:

```
field.scrollH = 0;                   // Прокручиваем влево до предела
field.scrollH += 1;                  // Прокручиваем вправо на 1 пиксел
field.scrollH = field.maxScrollH;    // Прокручиваем вправо до предела
```

Суть maxScrollH для динамического текстового поля (поля, свойству type которого задано значение DYNAMIC) показана на рис. 9.2. Текст серого цвета находится за пределами видимой области текстового поля.

Суть maxScrollH для поля ввода (поля, свойству type которого задано значение input) показана на рис. 9.3.

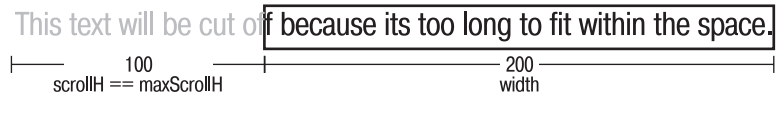


***Рис. 9.2.** Свойство maxScrollH для динамического текстового поля*

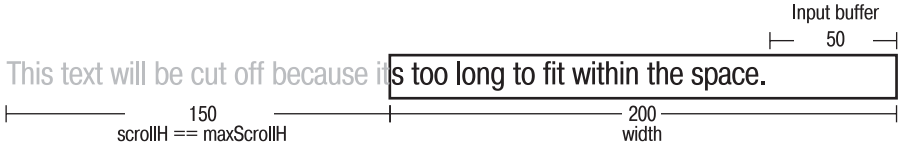


***Рис. 9.3.** Свойство maxScrollH для поля ввода*

Flash автоматически добавляет буферное пространство, предоставляя пользователю место для ввода. Опять же текст, выделенный серым, находится вне видимой области текстового поля.

Свойство *mouseWheelEnabled* позволяет управлять возможностью прокрутки текстового поля колесом мыши. По умолчанию оно имеет значение `false`. Задание значения `true` включает возможность прокрутки колесом мыши, как показано в данном примере:

```
field.mouseWheelEnabled = true;
```

9.12. Ответ на события прокрутки

Задача

Требуется осуществлять некоторые действия при прокрутке содержимого текстового поля.

Решение

Слушайте события прокрутки.

Обсуждение

При горизонтальной или вертикальной прокрутке текстового поля (имеется в виду изменение свойства `scrollV` или `scrollH` посредством кода ActionScript или полосы прокрутки) текстовое поле формирует событие прокрутки. Имя события прокрутки определяется константой `SCROLL` (прокрутка) класса *flash.events.Event*, а формируемое событие является объектом *flash.events.Event*. Следующий код регистрирует слушателя события прокрутки текстового поля `field`:

```
field.addEventListener(Event.SCROLL, onTextScroll);
```

А вот пример метода *onTextScroll( )*, который является слушателем события прокрутки:

```
private function onTextScroll(event:Event):void {
  trace("scroll");
}
```

См. также

Рецепты 9.23 и 9.24.

9.13. Форматирование текста

Задача

Требуется форматировать текст в текстовом поле.

Решение

Обратитесь к тегам HTML, передайте объект *TextFormat* в метод *TextField.setTextFormat( )* или используйте объект *StyleSheet* как значение свойства `styleSheet` текстового поля.

Обсуждение

Класс *TextField* не предоставляет возможности форматирования отдельных символов, хотя с помощью свойства `textColor` можно задавать цвет всего содержимого текстового поля. При этом существует несколько способов более тонкого форматирования текстовых полей:

- Применение тегов HTML, например `<font>`, `<b>` и `<u>`.
- Работа с объектом *TextFormat*.
- Работа с CSS.

У каждого способа есть преимущества и недостатки. HTML-форматирование относительно простое, но его труднее всего обслуживать. Работать с объектом *TextFormat* сложнее, чем с тегами HTML, однако, если требуется обеспечить сложное форматирование, это лучше, чем форматирование тегами HTML. Применение CSS и объекта *StyleSheet* обеспечивает самую большую гибкость. Стили загружаются в виде документа CSS, что сильно упрощает обслуживание, потому что документ CSS можно редактировать без повторного экспорта файла *.swf*.

Чтобы быстро и без труда отформатировать текст, проще всего использовать теги HTML. Например, следующий код обеспечивает отображение жирного шрифта с подчеркиванием:

```
field.htmlText = "<b>Текст жирным шрифтом</b> <u>Текст с подчеркиванием</u>";
```

Более сложное форматирование позволяют реализовать объекты *TextFormat*. Первый шаг в форматировании текста объектами *TextFormat* – создать экземпляр объекта с помощью метода-конструктора:

```
var formatter:TextFormat = new TextFormat();
```

Далее присваиваем свойствам объекта *TextFormat* необходимые значения:

```
formatter.bold = true;        // Жирный шрифт
formatter.color = 0xFFFF00;   // Желтый цвет
formatter.blockIndent = 5;    // Ширина отступа 5 пикселов
```

Передавая объект *TextFormat* в метод *setTextFormat( )* (задать формат текста) текстового поля, применяем форматирование ко всему существующему тексту поля:

```
field.setTextFormat(formatter);
```

При таком вызове метода *setTextFormat( )* форматирование объекта применяется ко всему тексту, уже присутствующему в текстовом поле. Форматирование не распространяется на текст, который будет внесен в поле после вызова *setTextFormat( )*. Если пользователь вводит еще какой-то текст, то исходный сохраняет примененное к нему форматирование, но введенный текст остается неформатированным. Форматирование аннулируется, если текстовое значение подвергается изменению средствами ActionScript:

```
field.text = "это текст примера";
field.setTextFormat(formatter);  // Применяется форматирование
field.text = "это новый текст";  // Нет форматирования
field.setTextFormat(formatter);  // Форматирование применяется повторно
field.text += "добавленный текст";   // Форматирование аннулируется
```

Если объект *TextFormat* изменен, необходимо повторно применить форматирование к текстовому полю, передавая измененный объект в метод *setTextFormat( )*. В противном случае изменения не будут отображены.

Поддержку CSS обеспечивает класс *flash.text.StyleSheet*. Конструктор *StyleSheet* не требует никаких параметров, как показано в примере:

```
var css:StyleSheet = new StyleSheet();
```

В табл. 9.1 перечислены поддерживаемые свойства CSS и их эквиваленты в ActionScript. При описании документа CSS или строки, интерпретируемой как CSS, указываются имена свойств CSS, а при описании объектов стилей указываются свойства ActionScript (см. далее).

Таблица 9.1. Свойства CSS, поддерживаемые Flash Player

Свойство CSS	Свойство ActionScript	Описание
`color`	`color`	Шестнадцатеричное значение как строка в формате `#RRGGBB`.
`display`	`display`	Способ отображения текста в текстовом поле. Возможные значения: `inline` (перед и после текста нет разрывов строки), `block` (стиль по умолчанию с разрывами строки перед и после) или `none` (скрытый).
`font-family`	`fontFamily`	Список имен шрифтов через запятую. Кроме встроенных в ролик шрифтов или именованных системных шрифтов, Flash Player поддерживает следующие группы шрифтов устройства: `sans`, `_serif`, `_typewriter`. Также могут использоваться шрифты `sans-serif`, `serif` и `mono`, они будут трактоваться как `sans`, `_serif` и `_typewriter` соответственно.
`font-size`	`fontSize`	Численный размер шрифта.

Таблица 9.1 (продолжение)

Свойство CSS	Свойство ActionScript	Описание
font-style	fontStyle	normal или italic.
font-weight	fontWeight	normal или bold.
kerning	kerning	true или false. Кернинг применяется только к поддерживающим его встроенным шрифтам. Кроме того, чтобы кернинг работал, файл *.swf* должен быть скомпилирован в Windows.
letter-spacing	letterSpacing	Расстояние между буквами в пикселах.
margin-left	marginLeft	Размер поля слева в пикселах.
margin-right	marginRight	Размер поля справа в пикселах.
text-align	textAlign	left, center, right или justify.
text-decoration	textDecoration	none или underline.
text-indent	textIndent	Отступ в пикселах.

Вновь созданный объект *StyleSheet* можно заполнить несколькими способами. Один из них состоит в том, чтобы определить объекты стилей и задать их методом *setStyle( )*. Объект стиля – это ассоциативный массив свойств ActionScript, представленных в табл. 9.1. Ниже приведен пример объекта стиля:

```
var sampleStyle:Object = new Object();
sampleStyle.color = "#FFFFFF";
sampleStyle.textAlign = "center";
```

Обратите внимание, что объект стиля можно определить и с помощью нотации объектных литералов, как показано ниже:

```
var sampleStyle:Object = {color: "#FFFFFF", textAlign: "center"};
```

Когда один или более объектов стиля описаны, их можно добавлять в объект *StyleSheet*. Делается это методом *setStyle( )*, который принимает два параметра: имя стиля и объект стиля. Можно определять стили как для тегов, так и для классов. Следующий фрагмент кода определяет CSS-класс `sample` (пример):

```
css.setStyle(".sample", sampleStyle);
```

Таблицу стилей можно определять с помощью объектов стилей и метода *setStyle( )*, но самый распространенный и удобный вариант работы с CSS – загружать и проводить синтаксический разбор внешнего документа CSS. Загрузка CSS во время выполнения обладает преимуществом возможности изменения стилей путем редактирования документа CSS, при этом не приходится повторно компилировать файл *.swf*.

Загрузка документа CSS осуществляется с помощью класса *flash.net.URLLoader*. Как только данные документа загружены, они могут слу-

жить для заполнения объекта *StyleSheet* путем их передачи в метод *parseCSS( )* объекта. Следующий пример использует *styles.css*, документ CSS, определенный следующим образом и сохраненный в том же каталоге, что и файл *.swf*:

```
p {
  font-family: _sans;
  color: #FFFFFF;
}
.emphasis {
  font-weight: bold;
  font-style: italic;
}
```

Следующий пример показывает, как загрузить документ CSS и использовать его данные для заполнения объекта *StyleSheet*:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.Event;
  import flash.text.TextFieldAutoSize;
  import flash.text.StyleSheet;
  import flash.net.URLLoader;
  import flash.net.URLRequest;

  public class CSSText extends Sprite {

    public function CSSText() {
         var loader:URLLoader = new URLLoader();
         loader.addEventListener(Event.COMPLETE, onLoadCSS);
         var request:URLRequest = new URLRequest("styles.css");
         loader.load(request);
    }

    private function onLoadCSS(event:Event):void {
      var css:StyleSheet = new StyleSheet();
      css.parseCSS(URLLoader(event.target).data);
         var field:TextField = new TextField();
         field.autoSize = TextFieldAutoSize.LEFT;
         field.wordWrap = true;
         field.width = 200;
         addChild(field);
         field.styleSheet = css;
        field.htmlText = "<p><span class='emphasis'>Lorem ipsum</span> dolor
        sit amet, consectetuer adipiscing elit. Morbi tortor purus, aliquet
        a, ornare ac, suscipit a, est. Nullam hendrerit molestie erat. Nunc
        nulla tortor, ullamcorper et, elementum vel, fringilla sed, dui.
        Praesent fermentum interdum orci.</ p>";
    }
  }
}
```

Есть несколько моментов, на которые следует обратить внимание при работе с CSS во Flash:

- CSS может применяться только к текстовым полям, отображающим HTML.
- HTML и CSS должны согласовываться. Например, если CSS определяет класс *someCSSClass*, то последний возымеет действие, только если HTML применяет его к некоторой части текста (как в предыдущем примере).
- HTML-текст должен вводиться *после* применения CSS.

Если требуется предоставить пользователю возможность выбора из ряда документов CSS, полезно будет сохранить HTML-текст в переменную или ассоциативный массив. Тогда HTML можно повторно использовать при каждой загрузке очередного CSS, как это делается в следующем примере:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.text.TextFieldAutoSize;
  import flash.text.StyleSheet;
  import flash.net.URLLoader;
  import flash.net.URLRequest;

  public class CSSText extends Sprite {

    private var _field:TextField;
    private var _html:String;

    public function CSSText() {
      var css1:TextField = new TextField();
      css1.text = "stylesheet 1";
      css1.selectable = false;
      var css1Container:Sprite = new Sprite();
      css1Container.addEventListener(MouseEvent.CLICK, onCSS1);
      css1Container.addChild(css1);
      addChild(css1Container);

      var css2:TextField = new TextField();
      css2.text = "stylesheet 2";
      css2.selectable = false;
      var css2Container:Sprite = new Sprite();
      css2Container.addEventListener(MouseEvent.CLICK, onCSS2);
      css2Container.addChild(css2);
      addChild(css2Container);

      css2Container.y = 25;

         _field = new TextField();
```

```
            _field.autoSize = TextFieldAutoSize.LEFT;
            _field.wordWrap = true;
            _field.width = 200;
            addChild(_field);
            _html = "<p><span class='emphasis'>Lorem ipsum</span> dolor sit
            amet, consectetuer adipiscing elit. Morbi tortor purus, aliquet a,
            ornare ac, suscipit a, est. Nullam hendrerit molestie erat. Nunc
            nulla tortor, ullamcorper et, elementum vel, fringilla sed, dui.
            Praesent fermentum interdum orci.</p>";
          _field.y = 50;
        }

        private function loadCSS(url:String):void {
            var loader:URLLoader = new URLLoader();
            loader.addEventListener(Event.COMPLETE, onLoadCSS);
            var request:URLRequest = new URLRequest(url);
            loader.load(request);

        }

        private function onCSS1(event:MouseEvent):void {
          loadCSS("styles.css");
        }

        private function onCSS2(event:MouseEvent):void {
          loadCSS("styles2.css");
        }

        private function onLoadCSS(event:Event):void {
          var css:StyleSheet = new StyleSheet();
          css.parseCSS(URLLoader(event.target).data);
            _field.styleSheet = css;
            _field.htmlText = _html;
        }
      }
    }
```

См. также

В рецепте 9.8 объясняется, как использовать форматированный тегами HTML текст, в рецепте 9.14 – как применять форматирование к вновь вводимому тексту, а не к существующему, а в рецепте 9.15 форматирование применяется к отдельным символам, а не ко всему полю.

9.14. Форматирование текста, вводимого пользователем

Задача

Требуется применить форматирование к тексту при его вводе в текстовое поле пользователем.

Решение

Примените объект *TextFormat* посредством свойства *defaultTextFormat* текстового поля.

Обсуждение

Чтобы применить форматирование к тексту, вводимому пользователем, необходимо задействовать свойство текстового поля *defaultTextFormat* (формат текста по умолчанию). Создается объект *TextFormat*, как в рецепте 9.13, и задается в качестве значения свойства *defaultTextFormat* текстового поля:

```
var formatter:TextFormat = new TextFormat();
formatter.color = 0x0000FF;       // Делаем текст синим
field.defaultTextFormat = formatter;
```

В случае использования *defaultTextFormat* форматирование применяется к тексту, вводимому в поле пользователем.

См. также

Рецепты 9.13 и 9.15.

9.15. Форматирование части существующего текста

Задача

Требуется форматировать не весь текст текстового поля, а только его часть, или применить разное форматирование к разным частям текстового поля.

Решение

Создайте объект *TextFormat* и используйте его для форматирования подстроки текстового поля с помощью одного из вариантов метода *setTextFormat( )*.

Обсуждение

Можно форматировать все текстовое поле, как показано в рецепте 9.13, или выбрать один из вариантов метода *setTextFormat( )* для форматирования только его части. Варианты метода позволяют применять форматирование только к заданному диапазону символов.

Вызывая метод *setTextFormat( )* и передавая в него два параметра, можно задавать форматирование одного символа текстового поля:

index

Отсчитываемый от нуля индекс символа, к которому должно быть применено форматирование.

textFormat

Ссылка на объект *TextFormat*.

В этом примере форматирование применяется только к первому символу:

```
field.setTextFormat(0, formatter);
```

Если требуется применить форматирование к диапазону символов, можно вызвать *setTextFormat( )* с тремя параметрами:

startIndex

Отсчитываемый от нуля индекс первого символа диапазона.

endIndex

Индекс символа после последнего символа рассматриваемого диапазона.

textFormat

Ссылка на объект *TextFormat*.

В этом примере осуществляется форматирование первых 10 символов:

```
field.setTextFormat(0, 10, formatter);
```

Можно заметить, что при форматировании частей текстового поля некоторые параметры форматирования применимы лишь в определенных ситуациях. Например, выравнивание текста возможно, только если форматирование применяется к первому символу строки.

См. также

Рецепт 9.13.

9.16. Задание шрифта текстового поля

Задача

Требуется с помощью ActionScript изменить шрифт, используемый для отображения некоторого текста.

Решение

Обратитесь к тегу `<font>` в HTML, задайте свойство `font` объекта *TextFormat* или свойство `font-family` в CSS.

Обсуждение

Программно задать шрифт, выбранный для отображения текста, можно несколькими способами. В случае форматирования тегами HTML применяется тег `<font>`. Например:

```
field.htmlText = "<font face='Arial'>Форматированный текст</font>";
```

Также можно прибегнуть к свойству `font` (шрифт) объекта *TextFormat*. В качестве значения этого свойства задается строковое имя шрифта (или шрифтов), который должен использоваться:

```
formatter.font = "Arial";
```

Кроме того, можно определить свойство `font-family` (семейство шрифтов) в CSS:

```
p {
  font-family: Arial;
}
```

Заданный шрифт должен быть установлен на компьютере, на котором воспроизводится ролик. Так бывает не всегда, поэтому можно задать несколько шрифтов через запятую:

```
formatter.font = "Arial, Verdana, Helvetica";
```

В предыдущем примере фигурирует объект *TextFormat*. Однако список шрифтов через запятую можно задать с помощью любой из описываемых здесь техник.

Шрифты применяются в порядке их расположения в списке: если Flash Player не находит первый шрифт, он пытается использовать следующий по списку.

Если ни один из заданных шрифтов не найден, берется системный шрифт по умолчанию. Можно задать *группу шрифтов*, в этом случае выбирается системный шрифт по умолчанию, входящий в данную категорию шрифтов. Предпочтительнее задавать группу шрифтов. Это позволяет Flash Player использовать системный шрифт по умолчанию, а разработчик, по крайней мере, будет знать общие характеристики шрифта. Существует три группы шрифтов: *_sans*, *_serif* и *_typewriter*. Обычным примером шрифтов группы *_sans* являются Arial или Helvetica, группы *_serif* – Times или Times New Roman, и в группу *_typewriter* входят такие шрифты, как Courier или Courier New.

См. также

Рецепты 9.13, 9.15 и 9.17.

9.17. Встраивание шрифтов

Задача

Требуется обеспечить правильное отображение текста, даже если предусмотренный шрифт не установлен на компьютере пользователя.

Решение

Встройте шрифт в ролик с помощью метатега [Embed]. Затем задайте свойству embedFonts текстового поля значение true и примените шрифт к текстовому полю с помощью тега <font>, объект *TextFormat* или CSS.

Обсуждение

Если необходимо гарантированно отображать текст заданным шрифтом, даже если этот шрифт не установлен на компьютере пользователя, используются встроенные шрифты. Встраивание шрифтов в ролик осуществляется с помощью метатега [Embed]. Он должен располагаться в файле ActionScript вне описания класса. Можно встраивать как шрифты TrueType, так и системные шрифты. Синтаксис встраивания шрифта TrueType с помощью метатега [Embed]:

```
[Embed(source="путьКФайлуTtf", fontName="ИмяШрифта",
                               mimeType="application/x-font-truetype")]
```

Путь к файлу шрифта TrueType может быть относительным или абсолютным, как в следующем примере:

```
[Embed(source="C:\Windows\Fonts\Example.ttf", fontName="ExampleFont",
mimeType="application/x-font-truetype")]
```

Значение атрибута *fontName* (имя шрифта) соответствует названию шрифта в CSS или ActionScript.

Синтаксис встраивания системных шрифтов аналогичен, за исключением того, что указывается атрибут *systemFont*, а не атрибут *source*. Значение атрибута *systemFont* – это имя встраиваемого системного шрифта. В следующем коде встраивается шрифт Times New Roman:

```
[Embed(systemFont="Times New Roman", fontName="Times New Roman", mimeType="
application/x-font-truetype")]
```

В предыдущем примере в атрибуте *fontName* указано фактическое имя системного шрифта.

Встроив шрифт, надо указать, что в текстовом поле должен использоваться именно он. Для этого свойству embedFonts текстового задается значение true. Значение по умолчанию – false, т. е. Flash задействует шрифты устройства. Если свойству embedFonts задано значение true, то в текстовом поле разрешены только встроенные шрифты. Если тексто-

вому полю, свойство `embedFonts` которого имеет значение `true`, задать шрифт устройства, в нем ничего не будет отображаться:

```
field.embedFonts = true;
```

После активации встроенных шрифтов надо указать, какие из них разрешены в текстовом поле. Делается это с помощью тега `<font>`, объекта *TextFormat* или CSS. Например, если значение *fontName* равно Times New Roman, то код может выглядеть так:

```
formatter.font = "Times New Roman";
```

В следующем примере шрифт задается тегом `<font>`:

```
field.htmlText = "<font family='Times New Roman'>Example</font>";
```

И следующий фрагмент кода показывает, как задать шрифт с помощью CSS:

```
var css:StyleSheet = new StyleSheet();
css.setStyle("p", {fontFamily: "Times New Roman"});
field.htmlText = "<p>Пример</p>";
```

Если свойство `embedFonts` равно `true`, то нельзя задать несколько шрифтов через запятую.

См. также

Рецепты 9.13 и 9.16.

9.18. Создание текста, который можно поворачивать

Задача

Требуется обеспечить отображение текста при повороте.

Решение

Используйте встроенные шрифты.

Обсуждение

Если не заданы встроенные шрифты, то в текстовых полях задействуются шрифты устройства. В большинстве случаев все работает идеально. Однако если требуется поворачивать текстовое поле или его родительский контейнер (например, надпись должна располагаться рядом с объектом вертикально), то нужны встроенные шрифты. При повороте текстового поля текст, написанный шрифтами устройства, исчезает.

См. также

Рецепт 9.17, в котором рассказано, как встраивать шрифты.

9.19. Вывод на экран символов Unicode

Задача

В приложении требуется вывести на экран текст с символами Unicode, который может содержать нелатинские символы.

Решение

Загрузите текст из внешнего источника. В значении свойства `text` текстового поля могут фигурировать символы таблицы кодов Unicode.

Обсуждение

Есть несколько способов вывести символы Unicode в текстовом поле:

- Загрузить текст, включающий символы Unicode, из внешнего источника (например, текстового файла, XML-документа, базы данных).
- Задать символ непосредственно в коде ActionScript как строковое значение.
- Обратиться к символам таблицы кодов Unicode.

Если текст загружается из внешнего источника, он задается как значение текстового поля. Более подробно о загрузке содержимого из внешних источников рассказано в главе 20.

Если среда разработки поддерживает Unicode (как, например, Flex Builder), то можно добавить Unicode-символ прямо в код приложения. В значении свойства `text` текстового поля может выступать Unicode-код символа. Все коды Unicode в ActionScript начинаются с префикса `\u`, за которым следует четырехразрядное шестнадцатеричное число. Код Unicode должен быть заключен в кавычки. Следующий пример выводит на экран знак регистрации торговой марки (®) обоими способами:

```
field.text = "Добавляем знак регистрации непосредственно (® ) или с помощью
кода Unicode (\u00AE)";
```

Если требуется найти символ Unicode, в Windows можно запустить утилиту Character Map (Таблица символов) через меню Start→Programs→Accessories→System Tools→Character Map (Пуск→Программы→Стандартные→Служебные→Таблица символов). В Mac OS X активируйте Input Menu (Меню ввода) (System Preferences→International→Input Menu (Настройки системы→Международные→Меню ввода)) и установите флажок Character Palette (Набор символов). Чтобы открыть Character Palette, отправляйтесь в строку меню, выберите флаг своей страны (например, звездно-полосатый для США) и вариант Show Character Palette (Показать набор символов).

См. также

Табл. A.1 приложения, в которой приведены коды Unicode.

9.20. Перемещение фокуса на текстовое поле

Задача

Требуется средствами ActionScript поместить фокус на текстовое поле.

Решение

Обратитесь к свойству *Stage.focus*.

Обсуждение

Программно установить фокус на определенное текстовое поле позволяет свойство *Stage.focus*. У каждого отображаемого объекта есть свойство *stage*, ссылающееся на экземпляр *Stage*. Поэтому в классе, расширяющем класс отображаемого объекта (*Sprite*, *MovieClip* и др.), следующий код передает фокус на текстовое поле `field`:

```
stage.focus = field;
```

При загрузке в броузер *.swf* не получает фокуса. Поэтому, чтобы иметь возможность программно задавать фокус элементу приложения Flash, необходимо переместить фокус на Flash Player.[1] Далее приведен рабочий пример, фокус размещается на текстовом поле при помощи кнопки спрайта:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.text.TextFieldType;
  import flash.events.MouseEvent;

  public class TextExample extends Sprite {

    public function TextExample() {
      var field:TextField = new TextField();
      field.border = true;
      field.background = true;
      field.type = TextFieldType.INPUT;
      addChild(field);
      var button:Sprite = new Sprite();
      button.graphics.lineStyle();
      button.graphics.beginFill(0xFFFFFF);
      button.graphics.drawRect(0, 0, 100, 50);
      button.graphics.endFill();
      button.addEventListener(MouseEvent.CLICK, onClick);
      button.y = 100;
```

[1] Проще говоря, пользователь после загрузки веб-страницы должен сделать щелчок мышью в области Flash-приложения. Без этого все действия с фокусом ввода во Flash-приложении работать не будут. – *Примеч. науч. ред.*

```
        addChild(button);
      }

      private function onClick(event:MouseEvent):void {
        stage.focus = TextField(getChildAt(0));
      }

    }
  }
```

Чтобы убрать фокус с текстового поля, необходимо присвоить свойству *Stage.focus* значение `null`:

```
stage.focus = null;
```

9.21. Как выделить текст средствами ActionScript

Задача

Требуется выделить часть текста в текстовом поле.

Решение

Задача решается при помощи метода *TextField.setSelection( )*.

Обсуждение

Метод *TextField.setSelection( )* выделяет часть текста в текстовом поле. Он принимает два параметра:

startIndex

Отсчитываемый от нуля индекс первого символа выделяемого текста.

endIndex

Индекс символа после выделенного текста.

Текст можно выделить, только если текстовое поле имеет фокус. Фокус помещается на элемент с помощью свойства *Stage.focus*, как говорилось в рецепте 9.20:

```
stage.focus = field;                // Помещаем фокус на текстовое поле
field.text = "это пример текста";   // Задаем текст
field.setSelection(0, 4);           // Выделяем слово "это"
```

Индексы первого и последнего символов выбранного диапазона можно получить с помощью свойств `selectionBeginIndex` (индекс начала выделения) и `selectionEndIndex` (индекс конца выделения).

См. также

Рецепты 9.20 и 9.23.

9.22. Как задать положение курсора ввода в текстовом поле

Задача

Требуется задать положение курсора ввода для текстового поля в коде ActionScript.

Решение

Обратитесь к методу *TextField.setSelection( )*.

Обсуждение

Положение курсора в текстовом поле можно задать методом *TextField.setSelection( )*. Для этого надо в качестве начального и конечного индексов передать одно и то же значение. В данном примере задается положение курсора в текстовом поле (предполагается, что фокус находится на нем):

```
// Устанавливаем курсор ввода перед первым символом
field.setSelection(0, 0);
```

С помощью доступного только для чтения свойства *caretIndex* (индекс символа вставки) можно получить положение курсора:

```
trace(field.caretIndex);
```

См. также

Рецепт 9.21.

9.23. Реакция на установку или снятие фокуса с текстового поля

Задача

Требуется осуществлять действие при установке или снятии фокуса с текстового поля.

Решение

Слушайте события *focusIn* и *focusOut*.

Обсуждение

При перемещении фокуса на текстовое поле оно формирует событие *focusIn* (фокус в) и при снятии фокуса – событие *focusOut* (фокус вне). В обоих случаях это объекты *flash.events.FocusEvent*. Класс *FocusEvent* определяет свойство *relatedObject* (связанный объект). Для событий *focusIn*, значением свойства *relatedObject* является ссылка на объект,

с которого переместился фокус. Для событий *focusOut* это ссылка на объект, который получил фокус. При регистрации слушателей используются константы класса *flash.events.FocusEvent* FOCUS_IN и FOCUS_OUT:

```
field.addEventListener(FocusEvent.FOCUS_IN, onFocus);
```

События *focusIn* и *focusOut* возникают уже после перемещения фокуса. Их нельзя отменить. Если это необходимо, слушайте события, происходящие до формирования *focusIn* и *focusOut*. События *keyFocusChange* (изменение фокуса клавишей) и *mouseFocusChange* (изменение фокуса мышью) относятся к отменяемым, которые формируются, когда пользователь пытается переместить фокус с текстового поля с помощью клавиатуры или мыши соответственно. Константы класса *FocusEvent*, KEY_FOCUS_CHANGE и MOUSE_FOCUS_CHANGE позволяют зарегистрировать слушателей этих событий. Метод *FocusEvent.preventDefault( )* отменяет поведение по умолчанию. В следующем примере запрещается перемещение фокуса с field1 на field2 с помощью клавиши Tab, если в поле field1 нет текста:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.text.TextFieldType;
  import flash.events.FocusEvent;

  public class Text extends Sprite {

    private var _field1:TextField;
    private var _field2:TextField;

    public function Text() {
      _field1 = new TextField();
      _field1.border = true;
      _field1.background = true;
      _field1.type = TextFieldType.INPUT;
      addChild(_field1);
      _field1.addEventListener(FocusEvent.KEY_FOCUS_CHANGE, onKeyFocus);
      _field2 = new TextField();
      _field2.border = true;
      _field2.background = true;
      _field2.type = TextFieldType.INPUT;
      addChild(_field2);
      _field2.y = 100;

    }

    private function onKeyFocus(event:FocusEvent):void {
      if(_field1.text == "") {
          event.preventDefault();
      }
    }

  }
}
```

9.24. Реакция на ввод текста пользователем

Задача

Требуется осуществлять действие при изменении содержимого текстового поля пользователем.

Решение

Слушайте событие *textInput*.

Обсуждение

Можно определить действия, которые будут выполняться при каждом изменении содержимого текстового поля пользователем независимо от характера изменения (удаляются ли символы или вырезаются, вводятся или вставляются). При внесении изменений в значение поля ввода оно формирует событие *textInput* (ввод текста). С помощью константы *lash.events.TextEvent.TEXT_INPUT* можно зарегистрировать слушателя этого события:

```
field.addEventListener(TextEvent.TEXT_INPUT, onTextInput);
```

Событие *textInput* – это объект *TextEvent*, и оно отменяемое. Класс *TextEvent* определяет свойство *text*, в котором содержится текст, введенный пользователем. Следующий пример гарантирует, что первый символ, вводимый пользователем в текстовое поле, не «a»:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.text.TextFieldType;
  import flash.events.TextEvent;
  import flash.events.TextEvent;

  public class Text extends Sprite {

    private var _field:TextField;

    public function Text() {
      _field = new TextField();
      _field.border = true;
      _field.background = true;
      _field.type = TextFieldType.INPUT;
      addChild(_field);
      _field.addEventListener(TextEvent.TEXT_INPUT, onTextInput);

    }

    private function onTextInput(event:TextEvent):void {
      if(event.text == "a" && _field.length == 0) {
          event.preventDefault();
      }
    }
  }
}
```

См. также

Рецепт 9.12.

9.25. Как добавить в текст гиперссылку

Задача

Требуется преобразовать некоторый отображаемый в текстовом поле текст в гиперссылку.

Решение

В свойстве `htmlText` объекта используйте HTML-теги `<a href>`. Или обратитесь к свойству `url` объекта *TextFormat*.

Обсуждение

Если решено добавлять гиперссылку с помощью HTML, необходимо ввести тег `<a href>` в свойство `htmlText` текстового поля следующим образом:

```
field.htmlText = "<a href='http://www.rightactionscript.com'>веб-сайт</a>";
```

Добавив атрибут `target` (цель) в HTML-тег `<a href>`, можно задать окно, в котором будет открываться ссылка. Например:

```
field.htmlText = "<a href='http://www.rightactionscript.com'
target='blank'>веб-сайт</a>";
```

Во Flash курсор мыши, попадая на текст, обозначенный как гиперссылка, приобретает форму руки с вытянутым указательным пальцем. Flash внешне никак не обозначает то, что текст является ссылкой, в отличие от большинства HTML-броузеров (в которых ссылка обозначается подчеркиванием и изменением цвета). Поэтому будет нелишним добавить разметку HTML, обеспечивающую подчеркивание и изменение цвета текста-ссылки:

```
var htmlLink:String = "<font color='#0000FF'><u>";
htmlLink += "<a href='http://www.rightactionscript.com'>веб-сайт</a>";
htmlLink += "</u></font>";
field.htmlText = htmlLink;
```

То же самое можно сделать и без HTML, с помощью объекта *TextFormat*. Класс *TextFormat* включает свойство `url` именно для этой цели. Чтобы связать форматированный текст с некоторым URL, задаем его как значение свойства `url`, например:

```
field.text = "веб-сайт";
var formatter:TextFormat = new TextFormat();
formatter.url = "http://www.rightactionscript.com/";
field.setTextFormat(formatter);
```

С помощью свойства target объекта *TextFormat* можно задать окно, в котором будет открываться ссылка:

```
field.text = "веб-сайт";
var formatter:TextFormat = new TextFormat();
formatter.url = "http://www.rightactionscript.com/";
formatter.target = "_blank";
field.setTextFormat(formatter);
```

Как и в случае с HTML-тегами, при создании гиперссылки с помощью объекта *TextFormat* Flash не предлагает никакой идентификации ссылки кроме изменения вида курсора. Разработчик может самостоятельно ввести цвет и/или подчеркивание текста-ссылки, чтобы пользователь мог отличить ее от другого текста. Для этого служат свойства color и underline объекта *TextFormat*:

```
field.text = "веб-сайт";
var formatter:TextFormat = new TextFormat();
formatter.color = 0x0000FF;
formatter.underline = true;
formatter.url = "http://www.rightactionscript.com/";
field.setTextFormat(formatter);
```

С помощью любой из приведенных в данном рецепте техник можно добавлять ссылки, которые указывают не только на протоколы *http* и *https*, как показано в примерах, но и на другие протоколы. Например, с помощью той же техники можно открыть новое сообщение электронной почты:

```
field.text = "email";
var formatter:TextFormat = new TextFormat();
formatter.color = 0x0000FF;
formatter.underline = true;
formatter.url = "mailto:joey@person13.com";
field.setTextFormat(formatter);
```

Однако не забывайте, что многие типы ссылок (например, ссылки *mailto*) работают, только если ролик воспроизводится в веб-броузере, в котором для протокола определен клиент по умолчанию.

Каскадные таблицы стилей позволяют применить расширенное форматирование к тегам <a href> посредством стилей a:link, a:active и a:hover, как показано в следующем примере:

```
var css:StyleSheet = new StyleSheet();
css.parseCSS("a {color: #0000FF;} a:hover {text-decoration: underline;}");
field.styleSheet = css;
field.htmlText = "<a href='http://www.rightactionscript.com'>веб-сайт</a>";
```

См. также

Рецепты 9.8 и 9.15.

9.26. Вызов ActionScript из гиперссылок

Задача

Требуется реализовать вызов метода ActionScript щелчком по гиперссылке.

Решение

Используйте протокол *event* и слушайте события ссылки.

Обсуждение

Во многих приложениях необходимо обеспечить вызов ActionScript щелчком по гиперссылке. В ActionScript 3.0 это сделать очень просто. Прежде всего необходимо описать гиперссылку с помощью протокола *event*:

```
field.htmlText = "<a href='event:http://www.rightactionscript.com'>веб-
сайт</a>";
```

Применение протокола *event* отменяет поведение по умолчанию. По щелчку по гиперссылке обычно открывается URL в веб-броузере, а при использовании протокола *event* формируется событие. Таким образом, необходимо зарегистрировать слушателя этого события. Тип формируемого события – *link* (ссылка), и при регистрации слушателя можно использовать константу *flash.events.TextEvent.LINK*:

```
field.addEventListener(TextEvent.LINK, onClickHyperlink);
```

Формируемое событие представляет собой объект класса *flash.events.TextEvent*. Свойство *text* объекта *event* содержит значение атрибута `href` без протокола *event*. Это означает, что в предыдущем примере значение свойства *text* объекта *event* равно *http://www.rightactionscript.com*. Поскольку при использовании протокола *event* гиперссылка не пытается открыть URL в окне броузера, значением `href` может быть не только действительный URL, а любая строка, по которой можно определить, какая гиперссылка выбрана.

9.27. Расширенное форматирование текста

Задача

Требуется применить расширенное форматирование текста.

Решение

Задача решается при помощи свойства *numLines* и методов *getCharBoundaries( )*, *getCharIndexAtPoint( )*, *getFirstCharInParagraph( )*, *getLineIndexAtPoint( )*, *getLineIndexOfChar( )*, *getLineLength( )*, *getLineMetrics( )*, *getLineOffset( )*, *getLineText( )* и *getParagraphLength( )*.

Обсуждение

В версиях Flash Player до и включая Flash Player 8 было трудно обеспечить большую точность при управлении и чтении форматирования текста. Однако начиная с Flash Player 8.5 класс *TextField* определяет API, обеспечивающий расширенные возможности работы с текстом.

Класс *TextField* определяет два метода для извлечения из текста информации о символах. Метод *getCharBoundaries( )* (получить границы символов) возвращает объект *flash.geom.Rectangle*, определяющий границы символа с заданным индексом. Метод *getCharIndexAtPoint( )* (получить индекс символа в точке) возвращает индекс символа, находящегося в точке с заданными координатами *x* и *y*. В следующем примере методы *getCharIndexAtPoint( )* и *getCharBoundaries( )* применяются для выделения символа по щелчку пользователя по нему:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.MouseEvent;
  import flash.geom.Rectangle;

  public class Text extends Sprite {

    private var _field:TextField;
    private var _highlight:Sprite;

    public function Text() {
      _field = new TextField();
      _field.border = true;
      _field.background = true;
      _field.multiline = true;
      _field.wordWrap = true;
      _field.selectable = false;
      _field.width = 400;
      _field.height = 400;
      addChild(_field);
      _field.text = "Lorem ipsum dolor sit amet, consectetuer adipiscing
       elit. Morbi tortor purus, aliquet a, ornare ac, suscipit a, est.
       Nullam hendrerit molestie erat. Nunc nulla tortor, ullamcorper et,
       elementum vel, fringilla sed, dui. Praesent fermentum interdum orci.";
      _field.addEventListener(MouseEvent.CLICK, onClick);
      _highlight = new Sprite();
      addChild(_highlight);
    }

    private function onClick(event:MouseEvent):void {
      var index:int = _field.getCharIndexAtPoint(mouseX, mouseY);
      var rectangle:Rectangle = _field.getCharBoundaries(index);
      _highlight.graphics.clear();
      _highlight.graphics.lineStyle(0, 0, 0);
      _highlight.graphics.beginFill(0x00FFFF, .25);
```

```
        _highlight.graphics.drawRect(rectangle.x, rectangle.y, rectangle.width,
            rectangle.height);
        _highlight.graphics.endFill();
      }

    }
  }
```

Также в классе *TextField* есть свойства и методы для получения информации о строках текста. Свойство *numLines* сообщает, сколько строк текста в текстовом поле. Метод *getLineIndexAtPoint( )* возвращает индекс строки по заданным координатам. Метод *getLineIndexOfChar( )* (получить индекс строки символа) возвращает индекс строки, содержащей символ с заданным индексом. Метод *getLineLength( )* (получить длину строки) возвращает число символов в строке с заданным индексом строки. Метод *getLineText( )* (получить строку текста) возвращает текст, содержащийся в строке с заданным индексом. Метод *getLineOffset( )* (получить смещение строки) возвращает индекс первого символа строки с заданным индексом строки. Метод *getLineMetrics( )* (получить метрики строки) возвращает объект *flash.text.TextLineMetrics* для строки с заданным индексом. Класс *TextLineMetrics* (метрики строки текста) определяет положение линий верхних и нижних выносных элементов, высоту, ширину, межстрочный интервал и другие свойства, описывающие строку текста.

Есть два метода для получения информации об абзацах. Метод *getFirstCharInParagraph( )* (получить первый символ абзаца) возвращает индекс первого символа абзаца, в котором содержится символ с заданным индексом. Метод *getParagraphLength( )* (получить длину абзаца) возвращает число символов в абзаце, содержащем символ с заданным индексом.

В следующем примере задействовано большинство методов данного рецепта. Здесь пользователь щелчком по символу выделяет весь абзац, в котором располагается данный символ:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.events.MouseEvent;
  import flash.geom.Rectangle;
  import flash.text.TextLineMetrics;

  public class Text extends Sprite {

    private var _field:TextField;
    private var _highlight:Sprite;

    public function Text() {
      _field = new TextField();
      _field.border = true;
      _field.background = true;
```

```
            _field.multiline = true;
            _field.wordWrap = true;
            _field.selectable = false;
            _field.width = 400;
            _field.height = 400;
            addChild(_field);
            _field.text = "Lorem ipsum dolor sit amet, consectetuer adipiscing
            elit. Morbi tortor purus, aliquet a, ornare ac, suscipit a, est. Nullam
            hendrerit molestie erat. Nunc nulla tortor, ullamcorper et, elementum
            vel, fringilla sed, dui. Praesent fermentum interdum orci.";
            _field.addEventListener(MouseEvent.CLICK, onDoubleClick);
            _highlight = new Sprite();
            addChild(_highlight);
        }

        private function onDoubleClick(event:MouseEvent):void {
            var index:int = _field.getCharIndexAtPoint(mouseX, mouseY);
            var startIndex:int = _field.getFirstCharInParagraph(index);
            var stopIndex:int = startIndex + _field.getParagraphLength(index);
            var startLine:int = _field.getLineIndexOfChar(startIndex);
            var stopLine:int = _field.getLineIndexOfChar(stopIndex - 1);
            var metrics:TextLineMetrics;
            var lineCharacter:int;
            var rectangle:Rectangle;
            _highlight.graphics.clear();
            _highlight.graphics.lineStyle(0, 0, 0);
            for(var i:int = startLine; i <= stopLine; i++) {
                lineCharacter = _field.getLineOffset(i);
                rectangle = _field.getCharBoundaries(lineCharacter);
                metrics = _field.getLineMetrics(i);
              _highlight.graphics.beginFill(0x00FFFF, .25);
             _highlight.graphics.drawRect(rectangle.x, rectangle.y, metrics.width,
              metrics.height);
              _highlight.graphics.endFill();
            }
        }

    }
}
```

9.28. Применение особого сглаживания текста

Задача

Требуется более точно управлять сглаживанием текста.

Решение

Встройте шрифт в ролик, задайте свойству *antiAliasType* текстового поля значение *flash.text.AntiAliasType.ADVANCED* и значения свойств *gridTypeFit* и *sharpness*.

Обсуждение

По умолчанию текст отображается с обычными настройками сглаживания. Для большинства шрифтов размером 10 и более такие настройки обеспечивают разборчивый текст. Однако для некоторых шрифтов и шрифтов меньшего размера обычное сглаживание не годится. В этих случаях для текстового поля можно задать расширенный тип сглаживания и использовать свойства *gridFitType* (тип привязки к сетке) и *sharpness* (резкость), чтобы обеспечить более строгий контроль над рендерингом текста.

Задавать расширенные настройки сглаживания можно только для встроенных шрифтов. Как встраивать шрифты, подробно рассказано в рецепте 9.17.

Свойство *TextField.antiAliasType* принимает одну из констант *flash.text.AntiAliasType*: `NORMAL` (по умолчанию) или `ADVANCED`. Если требуется включить более точные настройки сглаживания для текстового поля, свойству задается значение `ADVANCED`:

```
field.antiAliasType = AntiAliasType.ADVANCED;
```

Свойство *gridFitType* определяет, как контуры шрифта заполняют пикселы на экране. Возможные значения – константы `NONE`, `PIXEL` и `SUBPIXEL` класса *flash.text.GridFitType*. Значение по умолчанию, `NONE`, означает, что отображение шрифта не привязано к пикселам. Это может привести к тому, что при малых размерах шрифта текст будет размытым. Если задано значение `PIXEL`, горизонтальные и вертикальные линии контуров шрифта привязаны к целым пикселам. Это применимо только для выровненного по левому краю шрифта. Если требуется обеспечить такой рендеринг для шрифта, выровненного по центру или по правому краю, используется значение `SUBPIXEL`.

```
field.gridFitType = GridTypeFit.PIXEL;
```

Диапазон возможных значений свойства *sharpness* от –400 до 400. Значение по умолчанию равно 0. Это свойство определяет четкость отображаемого шрифта. Чем меньше число, тем хуже отображается шрифт. Чем больше число, тем более резко очерчен шрифт при рендеринге. Если шрифт выглядит размытым, даже несмотря на то, что для *gridFitType* задано `PIXEL` или `SUBPIXEL`, необходимо увеличивать значение резкости.

См. также

Рецепт 9.17.

9.29. Замена текста

Задача

Требуется заменить текст.

Решение

Для замены выделенного текста предназначен метод *replaceSelectedText( )*, а метод *replaceText( )* – для замены текстовой строки по заданному диапазону индексов.

Обсуждение

Метод *replaceSelectedText( )* (заменить выбранный текст) позволяет заменять выбранный текст текстового поля. В этот метод просто передается строка, которая должна быть вставлена вместо выбранного текста. Чтобы метод работал, фокус должен быть на текстовом поле:

```
_field.replaceSelectedText("новый текст");
```

Метод *replaceText( )* (заменить текст) может быть использован для замены текста в рамках строки, обозначенной начальным и конечным индексами. Следующий код заменяет текст от индекса 100 до индекса 150. Новая строка передается как третий параметр:

```
_field.replaceText(100, 150, "новый текст");
```

9.30. Получение списка системных шрифтов

Задача

Требуется получить список шрифтов, установленных в системе пользователя.

Решение

Обратитесь к статическому свойству *TextField.fontList*.

Обсуждение

Если необходимо работать с системными шрифтами (чтобы не встраивать шрифты или не обращаться к группе шрифтов), надо выяснить, какие шрифты установлены у пользователя. Массив системных шрифтов компьютера пользователя можно получить посредством свойства *TextField.fontList*.

```
trace(TextField.fontList);
```

На запрос списка доступных системных шрифтов возвращается массив строк. Чтобы применить шрифт к тексту, используется одна из методик, обсуждаемых в рецепте 9.16.

10

Фильтры и трансформации

10.0. Введение

ActionScript позволяет применять к отображаемым объектам различные преобразования и фильтры для изменения их цвета, формы, угла поворота, размера и обеспечения специальных эффектов. Трансформации – это изменения цвета, формы, угла поворота и размера. Все остальные специальные эффекты, обсуждаемые в данной главе, реализуются с помощью фильтров. Фильтры – это неотъемлемая часть Flash Player; они обеспечивают возможность реализовывать множество различных эффектов, от размытия до выдавливания.

10.1. Изменение цвета

Задача

Требуется закрасить отображаемый объект в определенный цвет.

Решение

В качестве значения свойства *transform.colorTransform* отображаемого объекта задайте объект *flash.geom.ColorTransform*.

Обсуждение

У всех отображаемых объектов есть свойство *transform.colorTransform*. Свойство *colorTransform* (преобразование цвета) – это объект *flash.geom.ColorTransform*, определяющий, какие преобразования цвета применяются к объекту. Свойство всегда возвращает копию объекта *ColorTransform*, фактически примененного к отображаемому объекту. Это означает, что напрямую изменять свойства объекта *colorTransform* нельзя. Однако есть возможность получить копию фактического объек-

та *ColorTransform*, присваивая переменной значение свойства *colorTransform*. После этого можно менять свойства этой копии и повторно присваивать их свойству *colorTransform*, как показано в следующем примере:

```
var color:ColorTransform = sampleSprite.transform.colorTransform;
color.color = 0xFFFFFF;
sampleSprite.transform.colorTransform = color;
```

Класс *ColorTransform* определяет два способа изменения чистого цвета отображаемого объекта. Можно задавать значение свойства `rgb`, как в предыдущем примере. Свойство `color` ожидает значение типа *uint*. Обычно значение удобно определять в шестнадцатеричном представлении (`0xRRGGBB`), как в предыдущем примере. Также с помощью свойств `redOffset` (смещение красного), `greenOffset` (смещение зеленого), `blueOffset` (смещение синего) и `alphaOffset` (смещение альфа) можно работать с каждым смещением в отдельности. Диапазон значений смещений от –255 до 255:

```
var color:ColorTransform = sampleSprite.transform.colorTransform;
color.redOffset = 255;
color.greenOffset = 255;
color.blueOffset = 255;
sampleSprite.transform.colorTransform = color;
```

В этом коде для преобразования каждому цвету задается смещение 255, что обеспечивает заливку белым цветом, и затем это преобразование применяется к объекту `sampleSprite`. Если изменить и значение `alphaOffset`, то изменится прозрачность отображаемого объекта, который подвергается преобразованиям. Прозрачность можно менять как через свойство `alphaOffset` класса *ColorTransform*, так и через свойство `alpha` отображаемого объекта; разницы никакой нет, кроме того что `alphaOffset` принимает значения в диапазоне от 0 до 255, а `alpha` – от 0 до 100.

См. также

Рецепт 10.2.

10.2. Применение оттенков цветов

Задача

Требуется применить оттенок к отображаемому объекту, а не закрасить его чистым цветом.

Решение

Обратитесь к свойству *transform.colorTransform* отображаемого объекта, но меняйте значения свойств-множителей, а не смещений.

Обсуждение

В рецепте 10.1 показано, как закрашивать отображаемые объекты чистым цветом. При таком закрашивании все детали рисунка отображаемого объекта станут неразличимыми. Закрашивание чистым цветом сродни применению заливки ко всей форме отображаемого объекта. Это означает, что всем пикселам отображаемого объекта присваивается одно и то же значение цвета.

При использовании оттенков цвет каждого пиксела меняется в зависимости от его исходного значения, таким образом, сохраняется контрастность. Применение оттенков к отображаемым объектам практически аналогично применению чистого цвета. В обоих случаях задействуется свойство *transform.colorTransform* объекта. Разница лишь в том, что при работе с оттенками свойствам смещений задаются значения 0 (значения по умолчанию) и изменяются значения свойств-множителей.

Допустимые значения свойств-множителей (`redMultiplier`, `greenMultiplier`, `blueMultiplier` и `alphaMultiplier`) лежат в диапазоне от 0 до 1. Значения множителей определяют отображаемую долю красного, зеленого, синего и альфа-компонентов исходного цвета пиксела. По умолчанию эти свойства равны 1. Это означает, что по умолчанию каждый пиксел отображает 100% красного, зеленого, синего и альфа-компонентов исходного значения пиксела. Следующий код задает зеленый оттенок отображаемого объекта, сохраняя для `greenMultiplier` значение 1 и присваивая свойствам `redMultiplier` и `blueMultiplier` значение 0.

```
var color:ColorTransform = sampleSprite.transform.colorTransform;
color.redMultiplier = 0;
color.blueMultiplier = 0;
sampleSprite.transform.colorTransform = color;
```

См. также

Рецепт 10.1.

10.3. Восстановление исходного цвета

Задача

Требуется восстановить цвет по умолчанию отображаемого объекта.

Решение

Создайте новый объект *ColorTransform*, свойства которого имеют значения по умолчанию, и присвойте его свойству *transform.colorTransform* отображаемого объекта.

Обсуждение

Значения по умолчанию объекта *flash.geom.ColorTransform* восстановят исходный цвет отображаемого объекта. Создается объект *ColorTransform*, свойства которого имеют значения по умолчанию, путем вызова конструктора без параметров. Следующий пример восстанавливает настройки цвета для `sampleSprite`:

```
sampleSprite.transform.colorTransform = new ColorTransform();
```

См. также

Рецепты 10.1 и 10.2.

10.4. Сдвиг

Задача

Требуется сдвинуть (наклонить) отображаемый объект.

Решение

Создайте объект *Matrix* со значениями по умолчанию, кроме свойств `b` и `c`. Свойства `b` и `c` управляют сдвигом относительно осей y и x соответственно. Используйте объект *Matrix* в качестве значения свойства *transform.matrix* отображаемого объекта.

Обсуждение

При сдвиге (или наклоне) на плоскости фиксируется некоторая точка, и все пикселы по одну сторону от нее сдвигаются в одном направлении, а все пикселы с другой стороны – в противоположном. В результате прямоугольник превращается в параллелограмм.

Класс *flash.geom.Matrix* включает свойства `a`, `b`, `c`, `d`, `tx` и `ty`. Свойства `b` и `c` определяют величину сдвига (тогда как `a` и `d` определяют коэффициент масштабирования, а `tx` и `ty` – смещение относительно осей `x` и `y`). Свойство `b` определяет сдвиг по оси y, а свойство `c` – сдвиг по оси x. По умолчанию `b` и `c` равны 0. Чем больше значения, тем сильнее объект сдвинут вправо и вниз. Отрицательные значения обусловливают сдвиг влево и вверх. Следующий код отрисовывает квадрат и посредством сдвига превращает его в параллелограмм, сдвинутый вдоль оси y. Объект *Matrix* использует для всех свойств значения по умолчанию, кроме `b`, которому задано значение 1. (Значения по умолчанию: `a=1`, `b=0`, `c=0`, `d=1`, `tx=0` и `ty=0`.)

```
var box:Sprite = new Sprite();
box.graphics.lineStyle();
box.graphics.drawRect(0, 0, 100, 100);
addChild(box);
box.transform.matrix = new Matrix(1, 1, 0, 1, 0, 0);
```

10.5. Применение базовых фильтров

Задача

Требуется применить основные фильтры, такие как тени, размытие, свечение и скосы.

Решение

Создайте новый объект фильтра и внесите его в массив фильтров отображаемого объекта.

Обсуждение

Пакет *flash.filters* содержит следующие классы базовых фильтров: *DropShadowFilter* (фильтр падающая тень), *BlurFilter* (фильтр размытие), *GlowFilter* (фильтр свечение), *BevelFilter* (фильтр скос), *GradientGlowFilter* (фильтр градиентное свечение) и *GradientBevelFilter* (фильтр градиентный скос). Перечисленные фильтры считаются базовыми, потому что они не требуют применения дополнительного отображаемого объекта в качестве части изображения поверхности или матриц для сложных преобразований. Каждый класс базового фильтра состоит из довольно простых свойств, которые обеспечивают простые эффекты при применении фильтра к отображаемым объектам. Например, свойства (и передаваемые в конструктор параметры) класса *DropShadowFilter* обеспечивают возможность менять смещение тени, ее цвет и степень размытия.

Базовые фильтры рассматриваются в документации Flash и Flex, поэтому не будем тратить время на детали создания и изменения основных свойств этих классов.

Объект созданного фильтра можно применить как значение свойства *filters* отображаемого объекта. Свойство *filters* (фильтры) – это массив объектов фильтров. Следующая строка кода обусловливает появление тени у отображаемого объекта `sampleSprite`:

```
sampleSprite.filters = [new DropShadowFilter()];
```

Когда свойству *filters* отображаемого объекта присваивается значение. происходит копирование массива фильтров, а не ссылки на него. Это означает, что любые изменения массива или фильтров массива не возымеют действия до тех пор, пока не будет осуществлено повторное присваивание, как показано в следующем примере:

```
var dropShadow:DropShadowFilter = new DropShadowFilter();
var sampleFilters:Array = [dropShadow];

// Применяем тень.
sampleSprite.filters = sampleFilters;

// Меняем цвет тени на белый. Однако поскольку применяемый
```

```
// к sampleSprite фильтр является копией dropShadow,
// это не оказывает никакого влияния на sampleSprite.
dropShadow.color = 0xFFFFFF;

// Добавляем фильтр свечения в массив. Однако поскольку массив,
// присвоенный sampleSprite, является копией sampleFilters,
// это не оказывает никакого влияния на sampleSprite.
sampleFilters.push(new GlowFilter());

// Повторно присваиваем sampleFilters как значение свойства filters
// объекта sampleSprite. Теперь обновления тени и массива
// (добавление фильтра свечения) оказывают влияние на отображаемый объект.
sampleSprite.filters = sampleFilters;
```

Аналогично при чтении значения свойства *filters* отображаемого объекта всегда возвращается копия массива фильтров. Это означает, что для добавления или удаления фильтров нельзя применять методы класса *Array* непосредственно к свойству *filters*.

```
var dropShadow:DropShadowFilter = new DropShadowFilter();

// Применяем тень.
sampleSprite.filters = [dropShadow];

// Это не добавляет фильтр свечения.
sampleSprite.filters.push(new GlowFilter());

// Надо скопировать текущий массив фильтров, добавить
// новый фильтр и повторно присвоить массив.
var sampleFilters:Array = sampleSprite.filters;
sampleFilters.push(new GlowFilter());
sampleSprite.filters = sampleFilters;
```

Свойство `filters` имеет кумулятивный эффект, т. е. эффект последовательного применения всех фильтров массива. Например, если массив включает два элемента, фильтры тени и свечения, свечение применяется к объекту, к которому уже применен фильтр тени (поскольку фильтр свечения стоит вторым в массиве).

Если требуется обеспечить раздельный эффект, приходится применять каждый фильтр к новой копии отображаемого объекта. Рассмотрим следующий пример, в котором к одному объекту применяются фильтры тени и свечения. Фильтр свечения применяется ко всей поверхности, включая тень. Как правило, это нежелательный эффект, поскольку в большинстве случаев требуется видоизменить исходную форму и получить раздельный, а не кумулятивный эффект.

```
var box:Sprite = new Sprite();
box.graphics.lineStyle();
box.graphics.beginFill(0xFFFFFF);
box.graphics.drawRect(0, 0, 100, 100);
box.graphics.endFill();
addChild(box);
box.filters = [new DropShadowFilter(10), new GlowFilter()];
```

Следующий фрагмент кода отрисовывает два прямоугольника, к одному из которых применяется свечение, а к другому – фильтр тени, свойству *knockout* (прозрачное окно) которого задано значение `true`:

```
var box:Sprite = new Sprite();
box.graphics.lineStyle();
box.graphics.beginFill(0xFFFFFF);
box.graphics.drawRect(0, 0, 100, 100);
box.graphics.endFill();
var boxShadow:Sprite = new Sprite();
boxShadow.graphics.lineStyle();
boxShadow.graphics.beginFill(0xFFFFFF);
boxShadow.graphics.drawRect(0, 0, 100, 100);
boxShadow.graphics.endFill();
addChild(boxShadow);
addChild(box);
box.filters = [new GlowFilter()];
boxShadow.filters = [new DropShadowFilter(10, 45, 0, 1, 4, 4, 1, 1,
                                          false, true)];
```

Свойство *knockout* доступно в большинстве базовых фильтров. Оно обеспечивает отображение фильтра, но при этом скрывает исходную форму. Например, если для фильтра тени активировать свойство *knockout*, в результате будет отрисована только тень без исходной формы, к которой фильтр применен. Это полезно во многих случаях, включая применение нескольких фильтров без кумулятивного эффекта, как в предыдущем примере, где свойству *knockout* задано значение true в конструкторе DropShadowFilter. Параметры конструкторов всех базовых фильтров можно найти в документации.

Убрать фильтр с отображаемого объекта можно, присваивая пустой массив или значение `null` свойству `filters` объекта:

```
sampleSprite.filters = [];
```

10.6. Применение дополнительных эффектов (выдавливание и т. д.)

Задача

Требуется применить дополнительные эффекты, такие как выдавливание, выделение краев и т. д.

Решение

Обратитесь к объекту *ConvolutionFilter*.

Обсуждение

Класс *flash.filters.ConvolutionFilter* позволяет применять множество эффектов, начиная от изменения яркости и контрастности до более

сложных, таких как выдавливание, размытие, выделение краев и повышение резкости.

Фильтрам свертки необходим массив значений, определяющий матрицу, служащую для отображения каждого пиксела в новое растровое изображение путем комбинирования значений смежных пикселов. (Используемая здесь линейная алгебра выходит за рамки данной книги.) Класс *ConvolutionFilter* (фильтр свертки) упрощает применение таких эффектов, так что разработчику не надо уметь перемножать матрицы или знать, как применять отображение пикселов. Достаточно иметь базовое представление об эффектах, создаваемых с помощью массивов элементов матрицы.

Конструктор *ConvolutionFilter* для всех параметров определяет значения по умолчанию. Однако как только эффект применен, понадобится задать другие значения (не применяемые по умолчанию), как минимум, для первых трех параметров. Первые два параметра определяют размерность матрицы, третий – массив элементов матрицы.

Первый параметр определяет число столбцов, второй – число строк. Ниже приведен пример матрицы с четырьмя столбцами и двумя строками:

```
1  2  3  4
5  6  7  8
```

Можно определять матрицы практически любой размерности, но все рассмотренные в данной книге эффекты ориентированы на квадратные матрицы (в которых строк и столбцов поровну).

Значения третьего параметра (массив элементов матрицы) задаются в порядке слева направо и сверху вниз. Например, матрица из предыдущего примера описывается следующим массивом:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Квадратная матрица с единицей, окруженной нулями, оказывает нейтральный эффект. В следующем примере кода к отображаемому объекту применяется именно такая матрица:

```
sampleSprite.filteres = [ConvolutionFilter(3, 3, [0, 0, 0, 0, 1, 0, 0, 0, 0])];
```

Если сумма элементов матрицы равна 1, как в данном примере, то яркость не меняется. Большие значения суммы обусловливают увеличение яркости, меньшие – уменьшение. В следующем примере увеличивается только яркость отображаемого объекта, никаких дополнительных эффектов:

```
sampleSprite.filteres = [ConvolutionFilter(3, 3, [0, 0, 0, 0, 2, 0, 0, 0, 0])];
```

А в данном примере к отображаемому объекту применяется эффект размытия. Однако поскольку сумма элементов матрицы больше 1, объект становится ярче. В данном случае яркость изменяется настолько сильно, что размытия практически невозможно заметить:

```
sampleSprite.filters = [new ConvolutionFilter(3, 3,
                                              [1, 1, 1, 1, 1, 1, 1, 1, 1])];
```

Непреднамеренное изменение яркости можно скорректировать с помощью делителя. Делитель не влияет на предполагаемый эффект (например, размытие или резкость), но при этом корректирует непредусмотренное изменение яркости. Для восстановления исходной яркости объекта служит делитель, равный по величине сумме элементов матрицы. Делитель задается как четвертый параметр. В следующем примере осуществляется такое же размытие, как в предыдущем случае, но исходная яркость восстанавливается с помощью делителя:

```
sampleSprite.filters = [new ConvolutionFilter(3, 3,
                                            [1, 1, 1, 1, 1, 1, 1, 1, 1], 9)];
```

Действие всех обсуждаемых в данной главе эффектов увеличивается при уменьшении центрального элемента матрицы и уменьшается при его увеличении. Например, предыдущий эффект размытия снижается, если центральный элемент равен 2. Обратите внимание: для сохранения яркости необходимо менять и делитель.

Более глубокое обсуждение фильтров свертки выходит за рамки данной книги. Подробнее изучить теорию этих фильтров, чтобы разрабатывать собственные матрицы для создания направленных эффектов, можно по адресу *http://www.cee.hw.ac.uk/hipr/html/convolve.html*.

Кроме того, все примеры данной главы являются ненаправленными. Например, размытие в предыдущем примере ненаправленное. Это означает, что оно не выглядит как размытие вследствие движения или радиальное размытие.

См. также

Рецепты 10.7, 10.8 и 10.9.

10.7. Выдавливание

Задача

Требуется сделать отображаемый объект выпуклым.

Решение

Примените фильтр свертки с матрицей выдавливания.

Обсуждение

Эффект выдавливания делает края поверхности выпуклыми или вдавленными на вид. Центральное значение матрицы выдавливания положительное, значения справа и слева симметричны. Они определяют степень выдавливания. Чем больше значение, тем более выпуклым бу-

дет изображение. Значения в столбце над и под центральным значением определяют, каким будет эффект (вдавливания или выдавливания), они также должны быть симметричными. Ниже представлен общий вид матрицы выдавливания:

```
a   d   -c
b   e   -b
c  -d   -a
```

Вот пример базовой матрицы выдавливания:

```
-1   1   1
-1   1   1
-1  -1   1
```

Следующая матрица 3×3 в сочетании с фильтром свертки обеспечивает хороший общий эффект выдавливания:

```
-2  -1   0
-1   1   1
 0   1   2
```

В следующем примере к отображаемому объекту применяется приведенный выше эффект выдавливания. Обратите внимание, что делитель не нужен, потому что значения симметричны и центральное значение равно 1:

```
sampleSprite.filters = [new ConvolutionFilter(3, 3,
                                    [-2, -1, 0, -1, 1, 1, 0, 1, 2])];
```

См. также

Рецепт 10.6.

10.8. Выделение краев

Задача

Требуется выделить края отображаемого объекта.

Решение

Используйте матрицу выделения краев с фильтром свертки.

Обсуждение

Для применения эффекта выделения краев с фильтром свертки служит матрица, отрицательное центральное значение которой окружено симметричным множеством положительных значений, как можно увидеть в следующем общем представлении матрицы:

```
a   b   c
d   e   d
c   b   a
```

Приведенная ниже матрица реализует общий эффект выделения краев:

```
0   1   0
1  -3   1
0   1   0
```

Здесь предыдущий эффект выделения краев применяется к отображаемому объекту:

```
sampleSprite.filters = [new ConvolutionFilter(3, 3,
                            [0, 1, 0, 1, -3, 1, 0, 1, 0])];
```

Чем больше число в центре, тем слабее эффект выделения краев. В примере ниже используется центральное значение –1 с делителем:

```
sampleSprite.filters = [new ConvolutionFilter(3, 3,
                            [0, 1, 0, 1, -1, 1, 0, 1, 0], 3)];
```

См. также

Рецепт 10.6.

10.9. Повышение резкости

Задача

Требуется применить эффект повышения резкости к отображаемому объекту.

Решение

Примените матрицу повышения резкости совместно с фильтром свертки.

Обсуждение

Повысить резкость объекта можно с помощью матрицы, структура которой очень похожа на матрицу, выделяющую края. Единственная разница: в матрице выделения краев отрицательное значение в центре окружено положительными значениями, а матрица повышения резкости использует положительное значение в центре, окруженное отрицательными значениями. Следующий пример обеспечивает повышение резкости отображаемого объекта:

```
sampleSprite.filter = [new ConvolutionFilter(3, 3,
                            [0, -1, 0, -1, 5, -1, 0, -1, 0])];
```

Увеличивая центральное значение и применяя делитель, можно ослабить эффект повышения резкости:

```
sampleSprite.filter = [new ConvolutionFilter(3, 3,
                            [0, -1, 0, -1, 10, -1, 0, -1, 0], 5)];
```

Уменьшая центральное значение и при этом увеличивая окружающие значения или используя делитель, можно усилить эффект повышения резкости:

```
sampleSprite.filter = [new ConvolutionFilter(3, 3,
                            [0, -1, 0, -1, 1, -1, 0, -1, 0], -3)];
```

См. также

Рецепт 10.6.

10.10. Создание цифрового негатива

Задача

Требуется превратить отображаемый объект в цифровой негатив.

Решение

Задача решается при помощи матрицы преобразования в цифровой негатив в сочетании с объектом *ColorMatrixFilter*.

Обсуждение

Применяется матрица преобразования в цифровой негатив посредством объекта *flash.filters.ColorMatrixFilter*, которая выглядит так:

```
-1  0  0  0  255
 0 -1  0  0  255
 0  0 -1  0  255
 0  0  0  1  0
```

В следующем примере демонстрируется применение эффекта цифрового негатива к отображаемому объекту:

```
sampleSprite.filters = [new ColorMatrixFilter([-1, 0, 0, 0, 255, 0, -1, 0,
                            0, 255, 0, 0, -1, 0, 255, 0, 0, 0, 1, 0])];
```

Также можно использовать константу `ascb.filters.ColorMatrixArrays.DIGITAL_NEGATIVE`, как показано здесь:

```
sampleSprite.filters = [new
ColorMatrixFilter(ColorMatrixArrays.DIGITAL_NEGATIVE)];
```

10.11. Применение эффекта полутонов

Задача

Требуется сделать изображение монохромным.

Решение

Это делается при помощи матрицы преобразования в монохромное изображение посредством объекта *ColorMatrixFilter*.

Обсуждение

Эффекта полутонов можно добиться, преобразовав все цвета в их эквиваленты яркости. В упрощенном повседневном смысле яркость показывает, насколько светлым или темным является изображение. Преобразовать цвет в эквивалент яркости можно путем умножения цветов на константы яркости красного, зеленого и синего. Константы, используемые в компьютерной графике, отличаются от стандарта видеозаписи и телевещания NTSC. В компьютерной графике для RGB-значений применяются константы 0,3086, 0,694 и 0,0820 соответственно. Следующая матрица описывает эффект полутонов:

```
0.3086  0.6094  0.0820  0  0
0.3086  0.6094  0.0820  0  0
0.3086  0.6094  0.0820  0  0
0       0       0       1  0
```

Следующий код применяет эффект полутонов к отображаемому объекту:

```
sampleSprite.filters = [new ColorMatrixFilter([0.3086, 0.6094, 0.0820, 0, 0,
0.3086, 0.6094, 0.0820, 0, 0, 0.3086, 0.6094, 0.0820, 0, 0, 0, 0, 0, 1, 0])];
```

Можно использовать константу `ascb.filters.ColorMatrixArrays.GRAYSCALE`:

```
sampleSprite.filters = [new ColorMatrixFilter(ColorMatrixArrays.GRAYSCALE)];
```

10.12. Изменение насыщенности

Задача

Требуется изменить насыщенность отображаемого объекта.

Решение

Примените матрицу насыщенности в сочетании с объектом *ColorMatrixFilter*.

Обсуждение

Насыщенность можно изменить с помощью матрицы:

```
a  b  c  0  0
d  e  f  0  0
g  h  i  0  0
0  0  0  1  0
```

В приведенной выше матрице значения от `a` до `i` определяются по следующим уравнениям, где `red`, `blue` и `green` – константы яркости, а `value` (значение) – значение насыщенности:

```
a = (1 - value) * red + value
b = (1 - value) * green
```

```
c = (1 - value) * blue
d = (1 - value) * red
e = (1 - value) * green + value
f = (1 - value) * blue
g = (1 - value) * red
h = (1 - value) * green
i = (1 - value) * blue + value
```

Если значение насыщенности равно 0, то это матрица преобразования в монохромное изображение.

Для создания массива матрицы насыщенности по одному заданному значению насыщенности можно использовать метод *ascb.filters.ColorMatrixArrays.getSaturationArray( )*.

```
sampleSprite.filters = [new ColorMatrixFilter
                            (ColorMatrixArrays.getSaturationArray(2))];
```

10.13. Изменение яркости

Задача

Требуется изменить яркость отображаемого объекта.

Решение

Используйте матрицу посредством объектов *ColorMatrixFilter*. Изменить яркость можно и с помощью объекта *ConvolutionFilter*.

Обсуждение

Можно изменить яркость посредством объекта *ColorMatrixFilter*, передав в него матрицу, которая обеспечивает одинаковое масштабирование или смещение красного, зеленого и синего. Ниже показано общее представление матрицы, обеспечивающей равное масштабирование красного, зеленого и синего:

```
a  0  0  0  0
0  a  0  0  0
0  0  a  0  0
0  0  0  1  0
```

Следующая матрица обеспечивает равное смещение красного, зеленого и синего:

```
1  0  0  0  a
0  1  0  0  a
0  0  1  0  a
0  0  0  1  0
```

Данный пример обеспечивает повышение яркости отображаемого объекта путем двукратного увеличения исходных значений цветов:

```
sampleSprite.filters = [new ColorMatrixFilter([2, 0, 0, 0, 0, 0, 2, 0, 0,
                            0, 0, 0, 2, 0, 0, 0, 0, 0, 1, 0])];
```

Яркость также можно изменять с помощью объекта *ConvolutionFilter*, как рассказывалось в рецепте 10.6.

См. также

Рецепт 10.6.

10.14. Изменение контрастности

Задача

Требуется настроить контрастность отображаемого объекта.

Решение

Используйте матрицу контрастности посредством объекта *ColorMatrixFilter*.

Обсуждение

Как упоминалось в рецепте 10.13, настроить яркость объекта можно путем масштабирования или смещения значений цветов. Изменить контрастность можно путем *одновременного* масштабирования и смещения значений цветов:

```
a  0  0  0  b
0  a  0  0  b
0  0  a  0  b
0  0  0  1  0
```

Следующие выражения позволяют вычислить коэффициент масштабирования и величину смещения по одному значению контрастности:

```
a = value * 11
b = 63.5 - (value * 698.5)
```

Посредством метода *ascb.filters.ColorMatrixArrays.getContrastArray( )* можно создать массив контрастности по заданному значению контрастности. Рабочая область значений контрастности от 0 до 1:

```
sampleSprite.filters = [new
ColorMatrixFilter(ColorMatrixArrays.getContrastArray(1))];
```

См. также

Рецепт 10.13.

11

Программная анимация

11.0. Введение

Анимацию можно определить как любое видимое изменение во времени. Даже если изображение не меняется с течением времени, нельзя с уверенностью сказать, что это неподвижное изображение. Существует множество свойств, манипулируя которыми можно обеспечить изменение изображения и, таким образом, создание анимации. Самое очевидное – это изменение положения объекта, что создает эффект его перемещения. Также можно менять размер, форму, угол поворота, прозрачность или цвет и многие другие параметры – все это тоже обеспечивает эффект анимации.

В более ранних версиях Flash анимация в основном реализовывалась с помощью *промежуточных кадров* (*tweens*). Объект помещался в ключевой кадр, создавался другой ключевой кадр, в котором этот объект был некоторым образом изменен, а Flash *автоматически* вставлял дополнительные кадры между ключевыми, отсюда и термин *автоматическая* анимация.[1] ActionScript позволяет создавать намного более динамичную и интерактивную анимацию.

Что касается объектов анимирования, то клип или спрайт отлично подходят для этого. Они могут содержать графические элементы, их методы и свойства обеспечивают возможность перемещения, изменения размера, поворота и любых других трансформаций. Клипы обычно используются только в среде разработки Flash, где вводятся дополнительные кадры, как при автоматической анимации.

---

[1] В англоязычной литературе оперируют понятием «*tween*-animation». Flash вставляет промежуточные кадры между (*in between*) ключевыми, отсюда и возник термин *tween*. – *Примеч. перев.*

Наконец, нужно как-то обеспечить, чтобы изменения происходили с течением времени. Лучше всего сделать это с помощью обработчика события *enterFrame* или таймера. Выражения ActionScript могут служить для изменения свойств анимированного объекта в каждом кадре или такте таймера, если таковой используется. Поскольку движение – самый наглядный вид анимации, примеры данной главы начинаются с перемещения объектов. В ней также описывается применение аналогичных техник к другим свойствам, например создание анимации за счет изменения размера и ориентации объекта.

11.1. Перемещение объекта

Задача

Требуется создать эффект перемещения некоторого графического объекта.

Решение

Определитесь с проекциями вектора скорости перемещения на ось x или y (или обе) и добавляйте эти значения к координатам объекта в каждом кадре или через заданный интервал времени.

Обсуждение

Вектор скорости часто путают со скалярной скоростью. Однако вектор скорости включает фактор направления. Например, «10 миль в час» – это скорость, а «10 миль в час точно на север» – это уже вектор скорости. Очень важно учитывать направление при проецировании вектора скорости на оси x или y. Положительный относительно оси x вектор скорости направлен вправо, отрицательный – влево. Аналогично положительный относительно оси y вектор скорости направлен вниз, отрицательный – вверх.

В первом примере определяется проекция вектора скорости на ось x, `_vx`, которой присваивается значение 3. Поскольку здесь анимация реализуется посредством события *enterFrame*, объект будет перемещаться на три пиксела вправо в каждом кадре:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Velocity extends Sprite {
        private var _sprite:Sprite;
        private var _vx:Number = 3;

        public function Velocity() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
```

```
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 50;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _sprite.x += _vx;
        }
    }
}
```

Если _vx задать значение –3, спрайт будет перемещаться в противоположном направлении. Также можно ввести проекцию вектора скорости на ось *y*, задав переменную _vy, присвоить ей значение и изменить метод onEnterFrame следующим образом:

```
public function onEnterFrame(event:Event):void {
    _sprite.x += _vx;
    _sprite.y += _vy;
}
```

Если вы не относитесь к ярым приверженцам покадровой анимации (она создается в предыдущем примере), можете применить управляемую таймером функцию, как показано в следующем фрагменте кода (выделено жирным шрифтом):

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class Velocity extends Sprite {
        private var _sprite:Sprite;
        private var _vx:Number = 3;
        private var _vy:Number = 2;
        private var _timer:Timer;

        public function Velocity() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 50;
            _sprite.y = 100;
            addChild(_sprite);
            _timer = new Timer(30);
            _timer.addEventListener("timer", onTimer);
            _timer.start();
        }
```

```
        public function onTimer(event:TimerEvent):void {
            _sprite.x += _vx;
            _sprite.y += _vy;
        }
    }
}
```

См. также

Рецепт 11.2, в котором рассказывается, как перемещать объект в заданном направлении с заданной скоростью.

11.2. Перемещение объекта в заданном направлении

Задача

Требуется переместить объект в заданном направлении с заданной скоростью.

Решение

Преобразуйте скорость и угол перемещения в проекции вектора скорости на оси x и y и прибавляйте эти значения к текущим координатам x и y объекта в каждом кадре или через заданный промежуток времени.

Обсуждение

В рецепте 11.1 описывалось, как переместить что-то вдоль осей x и y, зная вектор скорости. А что делать, если известны только угол и скорость перемещения объекта? Например, необходимо перемещать объект под углом 135 градусов со скоростью 4 пиксела за кадр.

С помощью основных тригонометрических функций можно преобразовать заданный угол и скорость в проекции вектора скорости на оси x и y. Во-первых, надо убедиться, что угол задан в радианах. Если он задан в градусах, преобразование в радианы вычисляется так:

```
radians = degrees * Math.PI / 180;
```

Обратное преобразование вычисляется по формуле:

```
degrees = radians * 180 / Math.PI;
```

Имея величину угла в радианах, с помощью функций *Math.sin* и *Math.cos* и скалярной скорости находим проекции вектора скорости на оси x и y:

```
vx = Math.cos(angle) * speed;
vy = Math.sin(angle) * speed;
```

После этого можно просто перемещать объект, как это было показано в рецепте 11.1. Вот пример перемещения под углом 135 градусов со скоростью 4 пиксела за кадр:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class AngularVelocity extends Sprite {
        private var _sprite:Sprite;
        private var _angle:Number = 135;
        private var _speed:Number = 4;
        private var _timer:Timer;

        public function AngularVelocity () {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 200;
            _sprite.y = 100;
            addChild(_sprite);
            _timer = new Timer(30);
            _timer.addEventListener("timer", onTimer);
            _timer.start();
        }

        public function onTimer(event:TimerEvent):void {
            var radians:Number = _angle * Math.PI / 180;
            var vx:Number = Math.cos(radians) * _speed;
            var vy:Number = Math.sin(radians) * _speed;
            _sprite.x += vx;
            _sprite.y += vy;
        }
    }
}
```

Конечно, в таком простом примере нет смысла пересчитывать проекции вектора скорости на оси x и y в каждом интервале, поскольку они не меняются. Поэтому их значения вычисляются один раз, результат сохраняется и используется в каждом кадре. Однако во многих случаях скорость и направление перемещения будут постоянно меняться, поэтому их придется пересчитывать для каждого кадра.

См. также

Рецепт 11.1.

11.3. Замедление

Задача

Требуется, чтобы объект, плавно замедляясь, перемещался в заданную точку и останавливался по ее достижении.

Решение

Используйте формулу замедления.

Обсуждение

Сначала рассмотрим концепцию простого *замедления* (*easing*). Пусть в определенной точке имеется объект и требуется, чтобы он плавно, замедляя ход, переместился в другую точку. Берем расстояние между этими двумя точками и перемещаем объект на какую-то часть данного расстояния, допустим, половину, одну треть или меньше. В следующей итерации находим новое расстояние и перемещаем объект уже на часть этого, нового, расстояния. Продолжаем таким же образом до тех пор, пока объект не будет так близок к цели, что можно считать, что он ее достиг.

Первые несколько переходов довольно большие, постепенно они становятся все меньше и меньше до тех пор, пока не возникает впечатление, что объект вообще не движется. В начале перемещения вектор скорости имеет какое-то значение, а к концу перехода стремится к нулю. Иначе говоря, вектор скорости зависит от расстояния: чем больше расстояние, тем выше скорость.

В следующем фрагменте кода показан простой пример замедления. Положение цели задано значениями переменных `_targetX` и `_targetY`. Значение `_easingSpeed` определяет, на какую часть расстояния до цели перемещается объект каждый раз. Здесь это 0,1, т. е. объект перемещается на десятую часть расстояния до цели за каждый интервал анимации:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class Easing extends Sprite {
        private var _sprite:Sprite;
        private var _easingSpeed:Number = 0.1;
        private var _targetX:Number = 400;
        private var _targetY:Number = 200;
        private var _timer:Timer;

        public function Easing() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
```

```
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 50;
            _sprite.y = 50;
            addChild(_sprite);
            _timer = new Timer(30);
            _timer.addEventListener("timer", onTimer);
            _timer.start();
        }

        public function onTimer(event:TimerEvent):void {
            var vx:Number = (_targetX - _sprite.x) * _easingSpeed;
            var vy:Number = (_targetY - _sprite.y) * _easingSpeed;
            _sprite.x += vx;
            _sprite.y += vy;
        }
    }
}
```

Нехорошо в данном случае только то, что таймер продолжает выполнение, даже когда объект уже очень близко к цели, т. е. практически, можно считать, достиг ее. Чтобы выйти из ситуации, найдем расстояние до цели и, если оно меньше определенного значения, просто выключим таймер. Это иллюстрирует выделенный жирным шрифтом код в следующем примере:

```
public function onTimer(event:TimerEvent):void {
    var dx:Number = _targetX - _sprite.x;
    var dy:Number = _targetY - _sprite.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < 1)
    {
        _sprite.x = _targetX;
        _sprite.y = _targetY;
        _timer.stop();
    }
    else
    {
        var vx:Number = dx * _easingSpeed;
        var vy:Number = dy * _easingSpeed;
        _sprite.x += vx;
        _sprite.y += vy;
    }
}
```

Здесь сначала вычисляются проекции расстояния на обе оси и общее расстояние. Если оно меньше 1, объект помещается на цель, таймер уничтожается. В противном случае перемещение продолжается в обычном режиме.

Однако иногда требуется прервать процесс замедления, например в случае с движущейся мишенью. В следующем примере объект за-

медляется при приближении к курсору мыши. Иначе говоря, здесь `_targetX` и `_targetY` заменяются на `mouseX` и `mouseY`:

```
public function onTimer(event:TimerEvent):void {
    var vx:Number = (mouseX - _sprite.x) * _easingSpeed;
    var vy:Number = (mouseY - _sprite.y) * _easingSpeed;
    _sprite.x += vx;
    _sprite.y += vy;
}
```

Это самая простая форма замедления, которой будет достаточно в большинстве случаев. Роберт Пеннер (Robert Penner) (*http://www.robertpenner.com*), широко известный и высокоуважаемый Flash-программист, вывел ряд намного более сложных формул замедления, ставших в некотором роде стандартными для приложений, в которых оно используется. Они даже были включены в стандартные классы ActionScript, поставляемые с Flash и Flex. В данной книге используются уравнения, написанные для ActionScript 1.0 и 2.0, но их можно без труда адаптировать для ActionScript 3.0. Эти уравнения с помощью множества разных формул делают такие вещи, как приближение с замедлением, удаление с ускорением, замедление при приближении *и* удалении или замедление на основании заданного интервала времени или количества кадров.

См. также

О векторе скорости рассказано в рецептах 11.1 и 11.2.

11.4. Ускорение

Задача

Требуется, чтобы объект начал движение медленно и постепенно ускорился.

Решение

Применяйте технику ускорения.

Обсуждение

Многие считают, что ускорение – это простое увеличение скорости. Более научное определение гласит, что ускорение – это векторная величина, определяющая приращение скорости. Хотя, конечно, все сводится к изменению скорости объекта, но это может быть и уменьшение скорости, и изменение ее направления.

Для работы с ускорением необходимо понимать суть вектора скорости, который рассматривается в рецептах 11.1 и 11.2. Ускорение тоже характеризуется *величиной* и *направлением*, которое может быть пред-

ставлено проекциями на оси x и y. В каждом кадре или интервале анимации величина проекции ускорения на каждую из осей добавляется к величине проекции скорости на эту ось, и затем величины проекций скорости прибавляются к координатам объекта, как в рецепте 11.1.

В следующем примере ускорение определяется величинами его проекций на оси x и y. Для этого служат переменные `_ax` и `_ay`, а переменные `_vx` и `_vy` задают скорость:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Accel extends Sprite {
        private var _sprite:Sprite;
        private var _ax:Number = .3;
        private var _ay:Number = .2;
        private var _vx:Number = 0;
        private var _vy:Number = 0;

        public function Accel() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 50;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _vx += _ax;
            _vy += _ay;
            _sprite.x += _vx;
            _sprite.y += _vy;
        }
    }
}
```

Как видите, сначала спрайт неподвижен, и по мере продвижения по сцене постепенно набирает скорость. Обычно, поскольку ускорение аддитивно, необходимо начинать с малых значений; скорость быстро нарастает со временем.

Кроме того, аналогично рецепту 11.2 ускорение может быть задано направлением и величиной ускоряющей силы:

```
var angle:Number = 45;
var accel:Number = .5;
```

Затем эти данные преобразовываются в проекции ускорения на каждую из осей:

```
var radians:Number = angle * Math.PI / 180;
_ax = Math.cos(radians) * accel;
_ay = Math.sin(radians) * accel;
```

Теперь у нас есть значения, которые можно добавлять к проекциям скорости.

Нелишним будет отметить, что гравитация – это просто проекция ускорения на ось *y*. Можно создать переменную гравитации, присвоить ей постоянное значение и добавлять к *y*-составляющей скорости в каждом кадре, получая реалистичный эффект гравитации.

См. также

Информацию о векторе скорости можно найти в рецептах 11.1 и 11.2.

11.5. Пружины

Задача

Требуется воспроизвести движение объекта, прикрепленного к пружине или резинке.

Решение

Используйте *Закон Гука*, формулу для пружины.

Обсуждение

Закон Гука описывает силы, действующие в пружине. Говоря простым языком, он гласит, что сила, приложенная к пружине (ускорение), пропорциональна ее растяжению. Это легко проверить. Если немного оттянуть резинку и отпустить ее, она лишь слегка ударит по руке. Но если оттянуть ее на максимально возможное расстояние, то удар будет болезненным.

Конечно, все пружины обладают разной «пружинистостью», или упругостью. Некоторые растягиваются легко и не слишком сильно сжимаются. Другие требуют намного большего усилия для сжатия и распрямляются с такой же силой. Упругость любой пружины можно охарактеризовать численно. Константу, описывающую упругие свойства пружин, представляет переменная `_k`. Ее значение обычно составляет доли единицы, например 0,1 или 0,2.

При моделировании пружины в ActionScript также необходимо задать конечную точку, в которую пружина перенесет объект. Наконец, требуется учесть затухание или трение. На практике при каждом колебании пружина понемногу теряет энергию и, в конце концов, останавливается в некоторой точке. Если не учесть этого в коде, объект будет качаться на пружине бесконечно. Затухание колебаний моделируется путем умножения значений вектора скорости на некоторый множи-

тель, например 0,95. Тем самым в каждом кадре скорость уменьшается на 5%, и в конце концов объект останавливается. Вот пример, в котором учтены все эти принципы:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Spring extends Sprite {
        private var _sprite:Sprite;
        private var _vx:Number = 20;
        private var _vy:Number = 0;
        private var _k:Number = .1;
        private var _damp:Number = .94;
        private var _targetX:Number = 200;
        private var _targetY:Number = 200;

        public function Spring() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 0;
            _sprite.y = 0;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            var ax:Number = (_targetX - _sprite.x) * _k;
            var ay:Number = (_targetY - _sprite.y) * _k;
            _vx += ax;
            _vy += ay;
            _sprite.x += _vx;
            _sprite.y += _vy;
            _vx *= _damp;
            _vy *= _damp;
        }
    }
}
```

Здесь переменная `_k` представляет жесткость пружины, ей задано значение 0,1. Переменная `_damp` равна 0,94 и контролирует затухание или трение. Конечная точка `200, 200` задана переменными `_targetX` и `_targetY`.

В методе `onEnterFrame` получаем расстояние от конечной точки до текущего положения объекта, тем самым узнаем растяжение пружины. Умножаем это значение на `_k`, жесткость пружины, получаем прикладываемую силу (или ускорение). Прибавляем это значение к проекциям скорости и полученные значения скорости – к координатам.

Наконец, учитываем затухание, умножая скорость на переменную затухания.

При выполнении этого кода мы должны видеть быстрые затухающие колебания пружины относительно конечной точки. В конце концов колебания затухают полностью и пружина останавливается в заданной конечной точке.

Конечная точка не обязательно неподвижна. Можно без труда изменить две строки предыдущего примера, в которых определяется ускорение, и использовать в качестве координат конечной точки координаты курсора мыши:

```
var ax:Number = (mouseX - _sprite.x) * _k;
var ay:Number = (mouseY - _sprite.y) * _k;
```

Получаем очень плавную интерактивную пружину.

Поэкспериментируйте со значениями `_k` и `_damp` и посмотрите, как можно изменять свойства пружины.

См. также

Рецепты 11.1, 11.2 и 11.4, где можно найти дополнительную информацию о векторе скорости и ускорении.

11.6. Тригонометрические выражения

Задача

Требуется реализовать сложное движение с использованием вращения, кругового или возвратно-поступательного движения.

Решение

К решению привлекаются встроенные математические функции *Math.sin( )*, *Math.cos( )* и *Math.atan2( )*.

Обсуждение

В рецептах 11.2 и 11.4 затрагивалась работа с функциями вычисления синуса и косинуса, но они могут быть полезны при реализации множества других эффектов, таких как движение объектов по круговой или овальной траекториям, плавные колебания вокруг определенной точки или поворот на заданный угол. Функции *Math. sin( )* и *Math.cos( )* основываются на свойствах прямоугольного треугольника (треугольника, один из углов которого равен 90 градусам). Не будем вникать в тонкости тригонометрии, отметим лишь, что при постепенном увеличении аргумента эти функции возвращают значения, плавно изменяющиеся в диапазоне от –1 до 0, 1, 0, назад до –1 и т. д. Это демонстрирует следующий фрагмент кода:

```
for(var i:Number = 0; i < 10; i += 0.1) {
    trace(Math.sin(i));
}
```

В результате выводится длинный список чисел. Если внимательно присмотреться, значения в нем начинаются с 0, возрастают до 0,999, уменьшаются до –0,999, опять возрастают и т. д. Теперь можно умножить эту функцию на любое число, скажем, 40 и получить список значений от –40 до 40. Если включить данное выражение в обработчик *enterFrame* или управляемый таймером метод и применить результат для определения положения объекта, можно получить движение назад-вперед или вверх-вниз, как показано в следующем примере:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class Oscillation extends Sprite {
        private var _sprite:Sprite;
        private var _angle:Number = 0;
        private var _radius:Number = 100;

        public function Oscillation () {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 0;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _sprite.x = 200 + Math.sin(_angle) * _radius;
            _angle += .05;
        }
    }
}
```

Здесь в переменной `_angle` хранится возрастающий аргумент *Math.sin( )*. Результат умножается на переменную `_radius`, которой задано значение 100. Это заставляет перемещаться спрайт вперед-назад на 100 пикселов.

Если использовать функцию *Math.cos( )* и сделать то же самое с координатой *y* спрайта, получим круговое движение:

```
public function onEnterFrame(event:Event):void {
    _sprite.x = 200 + Math.sin(_angle) * _radius;
    _sprite.y = 200 + Math.cos(_angle) * _radius;
    _angle += .05;
}
```

Чтобы сделать траекторию овальной, достаточно задать разные радиусы для каждой оси. Например, зададим для `_xRadius` значение 100, а для `_yRadius` значение 50 и сделаем следующее:

```
public function onEnterFrame(event:Event):void {
    _sprite.x = 200 + Math.sin(_angle) * _xRadius;
    _sprite.y = 200 + Math.cos(_angle) * _yRadius;
    _angle += .05;
}
```

Теперь, если задавать для каждой проекции разные углы и разные коэффициенты, можно получить хаотичное движение. Прежде всего создаем отдельные переменные для проекций всех величин на каждую из осей:

```
private var _xAngle:Number = 0;
private var _yAngle:Number = 0;
private var _xSpeed:Number = .13;
private var _ySpeed:Number = .09;
private var _xRadius:Number = 100;
private var _yRadius:Number = 50;
```

Вводим это в код моделирования движения:

```
public function onEnterFrame(event:Event):void {
    _sprite.x = 200 + Math.sin(_xAngle) * _xRadius;
    _sprite.y = 200 + Math.cos(_yAngle) * _yRadius;
    _xAngle += _xSpeed;
    _yAngle += _ySpeed;
}
```

Одно из возможных прикладных применений данного примера – моделирование полета мухи по комнате.

Еще одна очень полезная тригонометрическая функция – *Math.atan2( )*. Чаще всего с ее помощью реализуются объекты (спрайт, например), указывающие на мышь. Проекция расстояния на ось *y* равна `mouseY – _sprite.y`, а проекция расстояния на ось *x* равна `mouseX – _sprite.x`. *Math.atan2( )* возвращает угол. Преобразуем его в градусы и задаем как значение `_sprite.rotation`. Конечно, в спрайте надо реализовать какой-то графический элемент, который будет показывать направление вращения. В следующем примере создаются классические «следящие глаза» – маленькая настольная игрушка, следящая за перемещениями мыши по экрану и реализованная практически во всех графических операционных системах (на самом деле в данном примере создается всего один глаз, но этого достаточно, чтобы продемонстрировать принцип):

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    public class FollowingEye extends Sprite {
        private var _sprite:Sprite;

        public function FollowingEye() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0xffffff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
```

```
            _sprite.graphics.endFill();
            _sprite.graphics.beginFill(0x000000, 100);
            _sprite.graphics.drawCircle(20, 0, 5);
            _sprite.graphics.endFill();
            _sprite.x = 100;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            var dx:Number = mouseX - _sprite.x;
            var dy:Number = mouseY - _sprite.y;
            var radians:Number = Math.atan2(dy, dx);
            _sprite.rotation = radians * 180 / Math.PI;
        }
    }
}
```

Данный код отрисовывает два круга, маленький располагается в правой части большого. При реализации такого вращения графические элементы лучше выстраивать так, чтобы в начальном положении, при «нулевом угле», они «смотрели» вправо. Обработчик вычисляет две проекции расстояния и результирующий угол, преобразовывает радианы в градусы и задает полученное значение как угол поворота глаза.

11.7. Применение методов анимации к другим свойствам

Задача

Требуется применить методики, обсуждаемые в рецептах данной главы, для создания других эффектов, не только движения объекта.

Решение

Описанные методики применяются без изменений, но не к координатам объекта x и y, а к каким-то другим его свойствам.

Обсуждение

Хотя изменение положения объекта – самый наглядный метод анимации, все обсуждаемые здесь техники могут применяться практически ко всем свойствам клипа или спрайта. В данном рецепте дается всего несколько примеров, но возможности настолько безграничны, что перечислить их все было бы просто нереально.

Сначала попробуем применить скорость к свойству `rotation`; скорость вращения хранится в переменной `_vr`:

```
package {
```

```
    import flash.display.Sprite;
    import flash.events.Event;

    public class AnimatingRotation extends Sprite {
        private var _sprite:Sprite;
        private var _vr:Number = 4;

        public function AnimatingRotation () {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0xffffff, 100);
            _sprite.graphics.drawRect(-50, -20, 100, 40);
            _sprite.graphics.endFill();
            _sprite.x = 100;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
             _sprite.rotation += _vr;
        }
    }
}
```

В данном примере вместо круга фигурирует прямоугольник, так что можно увидеть вращение в действии. Переменной `_vr` задается значение 4, и в каждом кадре спрайт поворачивается на эту величину.

В следующем примере формула сжатия применяется для изменения размера спрайта. Обработчик щелчка кнопки мыши задает случайный масштаб, а обработчик события *enterFrame* реализует эффект расширения-сжатия. В результате, если щелкнуть по спрайту, его размер изменится:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class AnimatingProperties extends Sprite {
        private var _sprite:Sprite;
        private var _k:Number = 0.1;
        private var _damp:Number = 0.9;
        private var _scaleVel:Number = 0;
        private var _targetScale:Number = 1;

        public function AnimatingProperties () {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0xffffff, 100);
            _sprite.graphics.drawRect(-50, -50, 100, 100);
            _sprite.graphics.endFill();
            _sprite.x = 100;
            _sprite.y = 100;
            addChild(_sprite);
```

```
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            _sprite.addEventListener(MouseEvent.CLICK, onClick)
        }

        public function onEnterFrame(event:Event):void {
            _scaleVel += (_targetScale - _sprite.scaleX) * _k
            _sprite.scaleX += _scaleVel;
            _sprite.scaleY = _sprite.scaleX;
            _scaleVel *= _damp;
        }

        public function onClick(event:MouseEvent):void {
            _targetScale = Math.random() * 2 - .5;
        }
    }
}
```

Аналогичную функциональность можно было бы реализовать, применив замедление (рецепт 11.3), а не сжатие. Такая техника часто задействуется в приложениях создания фотогалерей для отображения фотографий различного размера. Область растягивается до нового размера, а затем до ее размеров увеличивается фото.

Более сложное применение кода движения – плавное изменение цвета. Лучше всего сделать это с помощью замедления. Начинаем с одного цвета и постепенно переходим в другой.

В следующем примере задается два набора значений цветов: `_red1`, `_green1`, `_blue1` и `_red2`, `_green2`, `_blue2`. Каждому значению поставлено в соответствие число от 0,0 до 1,0. В обработчике *enterFrame* эти значения выступают в качестве множителей, участвующих в вычислении красного, зеленого и синего каналов объекта *colorTransform* (изменение цвета), который применяется к спрайту. Все значения преобразуются с помощью техники замедления, таким образом, осуществляется плавный переход от одного значения к другому. Также здесь присутствует обработчик щелчка спрайта, который случайным образом задает три новых значения множителей. Квадрат плавно меняет цвет при каждом щелчке по нему:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.geom.ColorTransform;

    public class AnimatingColor extends Sprite {
        private var _sprite:Sprite;

        private var _red1:Number = 1;
        private var _green1:Number = 0;
        private var _blue1:Number = 0;

        private var _red2:Number = 0;
```

```
        private var _green2:Number = .5;
        private var _blue2:Number = 1;

        private var _easingSpeed:Number = 0.05;

        public function AnimatingColor () {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0xffffff, 100);
            _sprite.graphics.drawRect(-50, -50, 100, 100);
            _sprite.graphics.endFill();
            _sprite.x = 100;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            addEventListener(MouseEvent.CLICK, onClick);
        }

        public function onEnterFrame(event:Event):void {
            _red1 += (_red2 - _red1) * _easingSpeed;
            _green1 += (_green2 - _green1) * _easingSpeed;
            _blue1 += (_blue2 - _blue1) * _easingSpeed;
            _sprite.transform.colorTransform =
                    new ColorTransform(_red1, _green1, _blue1);
        }

        public function onClick(event:MouseEvent):void {
            _red2 = Math.random();
            _green2 = Math.random();
            _blue2 = Math.random();
        }
    }
}
```

См. также

Рецепт 11.3.

12

Строки

12.0. Введение

Строки – это основной текстовый элемент в ActionScript. Строка представляет собой множество, состоящее из нуля или более символов, заключенных в одинарные или двойные кавычки. В отличие от некоторых других языков программирования, ActionScript не разделяет символы и строки. И символы, и строки объединены в один тип данных – *String* (строка). Например:

```
var exampleA:String = "это строка";
var exampleB:String = 'это тоже строка';
var exampleC:String = "строки могут содержать символы, например -(*+5~";
var exampleD:String = "";  // Пустая строка
var exampleE:String = "x"; // Отдельный символ
var exampleF:String;       // По умолчанию, если не задано значение,
                           // принимает значение null
```

Строковые значения всегда заключаются в кавычки. Можно ставить как двойные, так и одинарные кавычки, только нельзя их смешивать: открывающие и закрывающие кавычки строки должны быть однотипными.

```
// Обе эти строки приводят к ошибке из-за несоответствия кавычек.
var exampleA:String = "неправильная строка';  // Закрывающая кавычка
                                              // должна быть двойной
var exampleB:String = 'еще одна неправильная строка"; // Закрывающая кавычка
                                                      // должна быть одинарной
```

ActionScript предоставляет широкие функциональные возможности работы со строками и символами. В ActionScript 3.0 появилась встроенная поддержка регулярных выражений (сопоставление с шаблоном), которая рассматривается в главе 13.

12.1. Объединение строк

Задача

Требуется объединить (конкатенировать) две или более строк в одно значение.

Решение

Используйте оператор конкатенации строк, +, комбинированный оператор конкатенации с присваиванием, +=, или метод *String.concat( )*.

Обсуждение

Строки (строковые операнды) можно объединить посредством оператора конкатенации +, если поставить его между ними:

```
// В результате получаем одно значение
// "Этоработает" (без пробела)
var example:String = "Это" + "работает";
```

Для того чтобы объединить более двух строк, нужны несколько операторов конкатенации и строковые операнды в соответствующем порядке:

```
// В результате получаем одно значение
// "Это работает" (с пробелом)
var example:String = "Это" + " " + "работает";
```

В предыдущих примерах не было особой причины задавать несколько строковых литералов, а затем объединять их. Достаточно было присвоить одно строковое значение (`"Это работает"` вместо `"Это" + " " + "работает"`). Однако тем самым была продемонстрирована техника работы с динамическими значениями. Оператор конкатенации может работать не только со строковыми литералами, но и с переменными, содержащими строковые значения (или значениями, которые могут быть преобразованы в строки). Например:

```
var attendance:int = 24;
// В результате получаем единое значение
// "Присутствует 24 человека"
var output:String = "Присутствует " + attendance + " человека";
```

Оператор конкатенации автоматически преобразовывает любые нестроковые значения в строки, если хотя бы один из операндов выражения является строкой. В предыдущем примере числовое значение 24 перед объединением с другими строками автоматически преобразуется в строковое значение 24. Однако если все операнды являются числами, интерпретатор ActionScript рассматривает оператор + как оператор сложения, а не оператор конкатенации:

```
var first:int = 24;
var second:int = 42;
// В результате возникает ошибка компиляции,
```

```
// "Implicit coercion of a value type 'Number' to an
// unrelated type 'String'"[1]
var result:String = first + second;
```

Числа можно объединять (не складывать) несколькими способами. Один из них – первым операндом в выражении конкатенации поставить пустую строку:

```
var first:int = 24;
var second:int = 42;
// В результате получаем строковое значение "2442"
var result:String = "" + first + second;
```

Пустая строка должна стоять в выражении первой, потому что в противном случае, если первыми будут идти два числа, над ними будет произведена операция сложения, а не конкатенации. А вот окончательно значение будет преобразовано в строку:

```
var first:int = 24;
var second:int = 42;
// В результате получаем строковое значение "66"
var result:String = first + second + "";
```

Другой вариант – использовать функцию преобразования *String( )*, чтобы гарантировать, что, по крайней мере, одно из чисел приводится к строке перед операцией конкатенации:

```
var first:int = 24;
var second:int = 42;
// В результате получаем строковое значение "2442"
var result:String = String( first ) + second;
```

Если данная техника применяется в операции, в которой принимают участие только два числа, какое из них (или оба) будет преобразовано в строку, не имеет значения. Но если объединяется более двух чисел, в строку должно быть преобразовано первое или второе из них. В противном случае числа, стоящие раньше преобразуемого в строку значения, будут суммированы, а не объединены:

```
var first:int = 24;
var second:int = 42;
var third:int = 21;
// В результате получаем строковое значение "6621"
var result:String = first + second + String( third );
```

Еще один вариант – использовать метод *toString( )*, доступный в большинстве встроенных объектов. В обоих типах, *Number* и *int*, есть метод *toString( )*:

```
var first:int = 24;
```

[1] Неявное приведение значения типа *Number* к неродственному типу *String*. – *Примеч. перев.*

```
var second:int = 42;
var third:int = 21;
// В результате получаем строковое значение "244221"
var result:String = first.toString() + second + third;
```

Если требуется сложить, а не объединить, два числовых значения в середине выражения конкатенации строк, они должны быть взяты в скобки. Тем самым меняется порядок операций, и внутреннее выражение вычисляется первым и рассматривается как операция сложения, а не конкатенации:

```
var first:int = 24;
var second:int = 42;
// В результате получаем "Присутствуют 66 человек"
var result:String = "Присутствуют " + ( first + second ) + " человек";
```

Оператор конкатенации с присваиванием += позволяет добавлять текст в конец существующей строки:

```
var attendance:int = 24;
var example:String = "Присутствуют ";
example += attendance;
// В результате получаем строковое значение
// "Присутствуют 24 человека"
example += " человека";
```

Это может пригодиться в нескольких случаях. Прежде всего при объединении длинных строковых значений. Код будет более удобным для чтения, если разбить их на несколько строк:

```
var example:String = "Это первое предложение длинного абзаца.";
example += "Добавляя строку за строкой к строковой переменной, вы делаете ";
example += "свой код более удобочитаемым.";
```

Однако предыдущий код можно сделать более выразительным, если вместо трех отдельных выражений разумно использовать пробелы и разрывы строк с обычным оператором +. Обратите внимание, что здесь только одна ;, таким образом, все выражение присваивания занимает три строки:

```
var example:String = " Это первое предложение длинного абзаца. "
    + "Разбивая длинную строку на меньшие части, с которыми намного "
    + "удобнее работать, и используя оператор +, "
    + "можно сделать ваш код более удобным для чтения.";
```

На практике конкатенация с присваиванием чаще всего применяется для добавления текста через некоторое время. Например, имеется окно истории чата, которое обновляется при вводе текста любым из участников беседы. Новый текст можно добавлять к существующей строке посредством оператора конкатенации с присваиванием:

```
// Метод, который будет добавлять имя пользователя
// и сообщение в строковую переменную истории чата
```

```
private function updateChatHistory( message:String, username:String ):void {
  // Предположим, что history – это строковая переменная,
  // содержащая реплики предыдущей беседы.
  // Добавим в конец существующей строки имя пользователя,
  // сообщение и новую строку.
  _history += username + ":" + message + '\n';
};
```

Добавлять новые значения в конец существующей строки можно также посредством метода *String.concat( )*. Он не меняет существующую строку, а возвращает новую, содержащую результат конкатенации:

```
var original:String = "исходная строка.";

// Присваиваем переменной modified значение
// "исходная строка.теперь модифицирована."
// Значение original не изменяется.
var modified:String = original.concat( "теперь модифицирована." );
```

Во Flash 4 для конкатенации строк служил оператор &. При обновлении файлов Flash 4 в формат Flash 5 или Flash 6 Flash преобразовывает оператор & в оператор *сложения*. Оператор + является предпочтительным оператором конкатенации строк во Flash 5 и более поздних версиях.

См. также

Рецепт 12.6.

12.2. Кавычки и апострофы в строках

Задача

Требуется использовать кавычки или апострофы в строковом значении.

Решение

Чтобы кавычки или апострофы, содержащиеся в строке, рассматривались как обычные символы, а не как символы завершения строки, перед ними ставится обратный слэш или внутри двойных кавычек ставятся одинарные, и наоборот.

Обсуждение

Компилятор ActionScript при обработке строковых литералов сопоставляет кавычки одного типа (одинарные кавычки с одинарными и двойные с двойными). Поэтому, если строковый литерал заключен в кавычки одного типа и в этом строковом значении встречаются такие же кавычки, код не будет скомпилирован правильно.

Следующее выражение приводит к формированию ошибки из-за несоответствия кавычек. Здесь строка начинается двойной кавычкой, по-

этому символ двойной кавычки перед буквой Д (¨Да) сигнализирует компилятору об окончании строки, и он не знает, что делать с остальными символами:

```
var error:String = ¨Он сказал, ¨Да.¨¨;  // Неправильно.
```

Одно из возможных решений – использовать одинарные кавычки для обозначения строкового литерала, заключенного в двойные кавычки, или двойные кавычки для строкового литерала, содержащего одинарные кавычки:

```
// Это выражение работает. В результате получаем строку:
// Он сказал: ¨Да.¨
var exampleA:String = 'Он сказал: ¨Да.¨';

// Это выражение также работает. В результате
// получаем строку: Он сказал: 'Да.'
var exampleB:String = ¨Он сказал: 'Да.'¨;
```

Однако если строковое значение содержит и одинарные, и двойные кавычки, эта техника не работает. Более того, при создании строк необходимо очень внимательно следить за типом кавычек, и если он все время меняется, то недолго и запутаться. Альтернативное решение, которое подходит во всех случаях, состоит в том, чтобы ставить перед кавычками или апострофами обратный слэш (\), который экранирует эти символы:

```
// Это выражение работает. В результате
// получаем строку: Он сказал, ¨Да.¨
var sExample:String = ¨Он сказал, \¨Да.\¨¨;
```

Встретив обратный слэш, компилятор рассматривает следующий символ буквально, не учитывая особого значения, которое он обычно имеет. Следовательно, когда кавычкам в строке предшествует обратный слэш, компилятор понимает, что кавычка не является сигналом окончания строкового значения, а должна быть интерпретирована просто как один из символов строки.

12.3. Как вставить специальные пробельные символы

Задача

Требуется добавить в строку пробельные символы, такие как табуляция или символ новой строки.

Решение

Это делается при помощи экранирующих последовательностей для задания специальных символов.

Обсуждение

Существует пять специальных пробельных символов и соответствующих им экранирующих последовательностей, как показано в табл. 12.1.

Таблица 12.1. Последовательности переключения кода для пробельных символов

Пробельный символ	Экранирующая последовательность
Перевод строки	\n
Табуляция	\t
Возврат	\b
Перевод страницы	\f
Возврат каретки	\r

Эти последовательности могут помещаться в строку. Они особенно полезны при отображении строкового значения в текстовом поле:

```
// В результате получаем строковое значение:
// эти    слова    разделены    символами    табуляции
var example:String = "эти\tслова\tразделены\tсимволами\ttабуляции";

/* В результате получаем строковое значение:
эти
слова
разделены
символами
перевода
строки
*/
var example:String = "эти\nслова\nразделены\nсимволами\nперевода\nстроки";
```

В отличие от предыдущих версий, ActionScript 3.0 не поддерживает константу `newline` (новая строка). Если он встречается в коде, то ее необходимо заменить экранирующей последовательностью `\n`.

Результат аналогичен.

```
// Формируется ошибка компиляции – необходимо
// заменить newline последовательностью "\n"
var error:String = "две" + newline + "строки"; // Ошибка компиляции: "Доступ
к неопределенному свойству 'newline'"
```

Во Flash символы перевода строки, перевода страницы и возвращения каретки отображаются одинаково. Поэтому если предполагается использовать документ только во Flash, нет разницы, какой из символов применять. Однако если с содержимым будут работать другие приложения, то для корректного отображения необходимо различать эти символы.

12.4. Поиск подстроки

Задача

Требуется найти подстроку в строке.

Решение

Это делается при помощи методов *indexOf( )* или *lastIndexOf( )* класса *String*.

Обсуждение

Для выявления заданной подстроки в строке можно использовать методы *indexOf( )* и *lastIndexOf( )*. Каждый из них возвращает начальный индекс обнаруженной подстроки. Метод *indexOf( )* осуществляет поиск слева направо, тогда как метод *lastIndexOf( )* рассматривает строку справа налево. Если подстрока не найдена, возвращается значение `-1`.

Метод *indexOf( ) принимает два параметра:*

substring (подстрока)

Подстрока, которую требуется найти.

startIndex (начальный индекс)

Необязательное отсчитываемое от нуля начальное положение, с которого начинается поиск. Отсчитываемый от нуля означает, что индекс первого символа строки 0, а не 1. Если этот параметр опущен, метод начинает поиск с начала строки (с индекса 0).

Если требуется проверить наличие заданной подстроки в строке, можно вызвать метод *indexOf( )*с одним параметром. Например:

```
var example:String = "This string contains the word cool twice. Very cool.";

// Получаем индекс первого экземпляра подстроки "cool" в example.
var index:int = example.indexOf( "cool" );

// Если метод indexOf() возвращает -1, "cool" не найдена.
if ( index != -1 ) {
  // На экран выводится: " Строка содержит слово cool
  // под индексом 30", потому что первая подстрока cool
  // начинается в example с индекса 30.
  trace( " Строка содержит слово cool под индексом " + index );
}
```

Индексы последующих экземпляров подстроки получаем, задавая методу *indexOf( )* второй, необязательный, параметр. Простой и эффективный способ найти следующий экземпляр подстроки состоит в том, чтобы передать в метод в качестве начального индекса значение, на единицу большее, чем возвращенное при предыдущем поиске:

```
var example:String = "This string contains the word cool twice. Very cool.";
```

```
// Получаем индекс первого экземпляра
// подстроки "cool" в example.
var index:int = example.indexOf( "cool" );

if ( index != -1 ) {
  // На экран выводится: "Строка содержит слово cool под индексом 30"
  trace( " Строка содержит слово cool под индексом " + index );
}

// Получаем индекс второго экземпляра подстроки "cool" в example.
// Передаем в метод найденное значение index + 1. То есть поиск
// начинается после начального индекса предыдущего экземпляра,
// что обеспечивает выявление следующего экземпляра подстроки.
// Если не прибавить 1 к значению index при передаче в indexOf(),
// метод опять возвращает местоположение первого экземпляра.
index = example.indexOf( "cool", index + 1 );

if ( index != -1 ) {
  // На экран выводится: "Строка содержит слово cool
  // под индексом 47", потому что второй экземпляр подстроки
  // начинается с индекса 47 в example.
  trace( "Строка содержит слово cool под индексом " + index );
}
```

Чтобы получить все экземпляры подстроки, можно поместить метод *indexOf( )* в выражение *while*. Например:

```
var example:String = "This string contains the word cool. Very cool. Yes,
cool.";

// Задаем index начальное значение -1, чтобы выражение
// while просматривало строку с самого начала
// (индекс станет равным 0 после операции index + 1)
var index:int = -1;

// Цикл выполняется, пока indexOf() не возвратит -1.
while ( ( index = example.indexOf( "cool", index + 1 ) ) != -1 ) {

  /* На экран выводится:
     Строка содержит слово cool под индексом 30
     Строка содержит слово cool под индексом 41
     Строка содержит слово cool под индексом 52
  */
  trace( "Строка содержит слово cool под индексом " + index );
}
```

Условное выражение *while* в предыдущем фрагменте кода кажется более сложным, чем есть на самом деле. На самом деле оно состоит из простых и понятных частей. Первая присваивает переменной `index` значение, возвращенное методом *indexOf( )*:

```
index = example.indexOf( "cool", index + 1 );
```

Перед первой итерацией исходное значение `index` равно –1, потому что оно было задано до выражения *while*. Это абсолютно логично и обеспе-

чивает корректное выполнение процесса, поскольку к индексу прибавляется единица в каждой итерации цикла. Если для `index` не задано начальное значение, то берется значение по умолчанию, т. е. 0. Добавление 1 к 0 приведет к тому, что поиск начнется со второго символа строки, индекс которого равен 1. При каждом последующем вычислении условия *while* значение `index` соответствует начальному индексу найденного в предыдущий раз соответствия. Если оно не будет увеличиваться на 1, метод всегда будет возвращать индекс первого соответствия, цикл станет бесконечным. То есть метод возвращает 30 и в следующей итерации опять начинает поиск с индекса 30, находит соответствие, опять возвращает 30, и цикл повторяется.

Вторая часть выражения *while*, прежде чем снова выполнять тело цикла, проверяет результат метода *indexOf( )*. Это важно, потому что без такой проверки цикл *while* продолжался бы бесконечно, т. к. в условном выражении любое отличное от нуля числовое значение, будь оно положительным или отрицательным, интерпретируется как `true`. Если *indexOf( )* возвращает –1, значит, соответствий больше нет, и цикл *while* прерывается.

Принцип работы метода *lastIndexOf( )* во многом аналогичен *indexOf( )*, кроме направления поиска (он перебирает строку справа налево). Метод *lastIndexOf( )* возвращает начальный индекс последнего экземпляра подстроки. Возвращаемый начальный индекс – это положение начала обнаруженной подстроки, а не ее конца, хотя поиск ведется в обратном направлении.

Как и *indexOf( )*, метод *lastIndexOf( )* принимает два параметра:

substring (подстрока)

Подстрока, которую требуется найти.

startIndex

Необязательный, отсчитываемый от нуля номер позиции в строке, с которой начинается поиск. Отсчитываемый от нуля означает, что индекс первого символа строки равен 0, а не 1. Если параметр опущен, метод начинает поиск с конца строки (`index` равен `string.length - 1`).

Если метод *lastIndexOf( )* не может найти подстроку в строке, он возвращает –1:

```
var example:String = "This string contains the word cool twice. Very cool.";

// Получаем индекс последнего экземпляра
// подстроки "cool" в example.
var index:int =  example.lastIndexOf( "cool" );

if ( index != -1 ) {
  // На экран выводится: "Строка содержит слово cool
  // под индексом 47", потому что последний экземпляр
  // подстроки в example начинается с индекса 47.
```

```
  trace( "Строка содержит слово cool под индексом " + index );
}
// Получаем индекс предпоследнего экземпляра "cool" в example.
// Передаем в метод предыдущее значение index - 1.
// Тем самым поиск начинается с предыдущего индекса относительно
// найденного ранее экземпляра подстроки, т.е. поиск ведется
// справа налево. Если не вычесть 1 из значения index
// при его передаче в lastIndexOf(), метод вновь возвращает
// местоположение последнего экземпляра.
index = example.lastIndexOf( "cool", index - 1 );

if ( index != -1 ) {
  // На экран выводится: "Строка содержит слово cool под индексом 30",
  // потому что предшествующий последнему экземпляр
  // подстроки начинается в example с индекса 30.
  trace( "Строка содержит слово cool под индексом " + index );
}
```

Индексы всех экземпляров подстроки можно получить посредством той же техники, что и с методом *indexOf( )*: объединить цикл *while* с методом *lastIndexOf( )* и задать начальное значение индекса.

```
var example:String = "This string contains the word cool. Very cool. Yes,
cool.";

// Задаем index начальное значение example.length,
// чтобы выражение while начинало поиск с самого
// последнего индекса. На самом деле, последний индекс
// равен example.length - 1, но в цикле из значения
// index вычитается 1, поэтому аккуратно инициализируем index здесь.
var index:int =  example.length;

// Цикл выполняется, пока lastIndexOf() не возвратит -1.
while ( ( index = example.lastIndexOf( "cool", index - 1 ) ) != -1 ) {

  /* На экран выводится:
     Строка содержит слово cool под индексом 52
     Строка содержит слово cool под индексом 41
     Строка содержит слово cool под индексом 30
  */
  trace( "Строка содержит слово cool под индексом " + index );
}
```

Этот фрагмент кода очень похож на пример с методом *indexOf( )*, за исключением двух небольших отличий. Первое: начальное значение `index` **задается равным** `example.length`**, а не –1, как раньше, потому что *lastIndexOf( )* ведет поиск справа налево, а не слева направо, т. е. с конца строки** `example`**. Второе: в каждой итерации из значения индекса вычитается 1, чтобы поиск осуществлялся в направлении справа налево.**

Методы *indexOf( )* и *lastIndexOf( )* ведут поиск с учетом регистра символов. Например, подстрока «cool» не будет обнаружена в строке «Cool» из-за несовпадения регистров. Для проведения нечувствитель-

ного к регистру поиска вместе с *indexOf( )* или *lastIndexOf( )* служит метод *toLowerCase( )*:

```
// Создаем строку, в которой встречается
// и подстрока "cool", и подстрока "Cool".
var example:String = "Cool. This is a string with the word cool. It spells"
                   + " cool as both cool (lowercase) and Cool (capitalized).";

var search:String = "cool";

// Выводим индекс первого экземпляра "cool". Результат - 37, потому что
// метод не находит "Cool", т.к. поиск является чувствительным к регистру.
trace( example.indexOf( search ) );

// Выводим индекс первого экземпляра "cool" после преобразования
// всех символов строки в нижний регистр методом toLowerCase().
// Результат – 0, потому что метод находит подстроку "Cool"
// (которая была преобразована в cool) в самом начале строки.
trace( example.toLowerCase().indexOf( search ) );

// Выводим индекс последнего экземпляра "cool". Результат - 66,
// потому что метод не находит последнего "Cool"
// из-за чувствительности к регистру.
trace( example.lastIndexOf( search ) );

// Выводим индекс последнего экземпляра cool после преобразования
// всех символов строки в нижний регистр методом toLowerCase().
// Результат – 87, потому что метод находит последний "Cool".
trace( example.toLowerCase().lastIndexOf( search ) );

// Теперь меняем строку поиска на "Cool" (с большой буквы).
search = "Cool";

// Выводим индекс первого экземпляра "Cool" после преобразования всех
// символов строки в нижний регистр. Результат равен –1, потому что
// "Cool" не существует в строке, включающей только строчные символы.
trace( example.toLowerCase().indexOf( search ) );

// Это аналогично предыдущей строке кода, но строка поиска тоже
// переводится в нижний регистр. Тогда в результате получаем 0,
// потому что начальный индекс первого экземпляра "cool"
// (без учета регистра) - 0. Перевод обеих строк, искомой и той,
// в которой проводится поиск, в нижний регистр гарантирует
// абсолютно нечувствительный к регистру поиск.
trace( example.toLowerCase().indexOf( search.toLowerCase() ) );
```

В этом фрагменте кода для проведения нечувствительного к регистру поиска можно свободно заменить метод *toLowerCase( )* на *toUpperCase( )*. Суть в том, чтобы перед поиском перевести исходную строку и подстроку в один регистр (верхний или нижний, не важно).

См. также

Рецепт 13.3.

12.5. Извлечение подстроки

Задача

Требуется извлечь подстроку из строки.

Решение

Задача решается при помощи методов *substring( )*, *substr( )* или *slice( )*.

Обсуждение

Методы *substring( )*, *substr( )* и *slice( )* возвращают значение подстроки, не изменяя исходную строку. Они отличаются только принимаемыми параметрами.

Метод *substr( ) принимает до двух параметров:*

startIndex

Положение первого символа подстроки. Значение может быть отрицательным, в этом случае индекс вычисляется от конца строки, где –1 – это индекс последнего символа, –2 – предпоследнего и т. д.

length

Число символов в извлекаемой подстроке. Если параметр опущен, используются все символы от начала до конца.

```
var example:String = "Зайчики";
trace( example.substr( 0 ) );      // На экран выводится: Зайчики
trace( example.substr( 0, 3 ) );   // На экран выводится: Зай
trace( example.substr( 3, 3 ) );   // На экран выводится: чик
trace( example.substr( -1 ) );     // На экран выводится: и
trace( example.substr( -2, 5 ) );  // На экран выводится: ки
```

Методы *substring( )* и *slice( )* принимают одинаковые параметры.

`startIndex`

Положение первого символа извлекаемой подстроки.

`endIndex`

Положение следующего символа после последнего символа извлекаемой подстроки. То есть подстрока будет содержать символы, начиная от начального индекса до символа с индексом `endIndex`, но не включая его. Если параметр опущен, используются все символы от начального индекса до конца строки.

Методы *substring( )* и *slice( )* отличаются тем, что *substring( )* принимает только положительные значения, а отрицательные он интерпретирует как 0. Также если *endIndex* меньше *startIndex*, метод *substring( )* автоматически меняет их местами, всегда используя в качестве начального индекса меньший из двух параметров. С другой стороны, метод *slice( )* принимает отрицательные значения для обоих параметров, *startIndex* и *endIndex*; в этом случае отсчет отрицательных значений

ведется с конца строки, где –1 – это последний символ строки. Метод *slice( )* возвращает пустую строку, если *endIndex* меньше *startIndex*.

```
var example:String = "Кролики";

// Оба этих метода обеспечивают вывод всей строки,
// начиная с индекса 0 и до последнего индекса
// (значение которого – example.length - 1).
trace( example.substring( 0 ) );       // На экран выводится: Кролики
trace( example.slice( 0 ) );           // На экран выводится: Кролики

// Метод substring() не возвращает ничего, потому что
// преобразовывает отрицательные индексы в 0.
// Метод slice() возвращает "ик", т.е. подстроку
// от третьего с конца (индекс -3) до
// предпоследнего (индекс -1) символа.
trace( example.substring( -3, -1 ) );  // На экран ничего не выводится
(пустая строка)
trace( example.slice( -3, -1 ) );      // На экран выводится: ик

// Здесь в обоих случаях выводится подстрока "ро".
trace( example.substring( 1, 3 ) );    // На экран выводится: ро
trace( example.slice( 1, 3 ) );        // На экран выводится: ро

// Метод substring() возвращает подстроку "ро",
// потому что автоматически меняет порядок параметров.
// Метод slice()не возвращает ничего, потому что
// не меняет порядок параметров.
trace( example.substring( 3, 1 ) );    // На экран выводится: ро
trace( example.slice( 3, 1 ) );        // На экран выводится пустота (пустая
строка)
```

Обычно методы извлечения подстрок применяются в сочетании с методами *indexOf( )* и *lastIndexOf( )*. Сначала методы *indexOf( )* и *lastIndexOf( )* выявляют в строке подстроку, а потом методы извлечения подстроки возвращают ее.

В данном примере извлекаются расширение и имя файла (без расширения) исходя из предположения, что они разделены точкой:

```
var filename:String = "document.jpg";
// Находим положение точки.
var extensionIndex:Number = filename.lastIndexOf( '.' );

// Перед точкой находится только
// имя файла без расширения ("mydocument").
var extensionless:String = filename.substr( 0, extensionIndex );
trace( "Имя файла – " + extensionless );

// После точки находится только расширение ("jpg").
var extension:String = filename.substr( extensionIndex + 1, filename.length );
trace( "Расширение файла – " + extension );
```

Также можно использовать метод *split( )*, предполагая, что имя файла включает только одну точку:

```
var filename:String = "document.jpg";
// Разделяем строку по точке.
var nameParts:Array = filename.split(".");

// Первый элемент - "document" (все до первой точки).
var extensionless:String = nameParts[0];
trace ("Имя файла - " + extensionless);

// Второй элемент - "jpg" (все после
// первой точки и до следующей).
var extension:String = nameParts[1];
trace ("Расширение файла - " + extension);
```

Проанализируем два предыдущих примера. Что если в имени файла не будет точки, например *document*? (Подсказка: первый пример приведет к недействительному результату.) Что если в имени файла несколько точек, например *ascb_fig8.1.bmp*? (Подсказка: второй пример приведет к недействительному результату.)

Вот универсальное решение:

```
// Этот метод возвращает все символы до последней точки,
// если таковые имеются.
private function removeExtension( filename:String ):String {
  // Находим местоположение точки.
  var extensionIndex:Number = filename.lastIndexOf( '.' );
  if ( extensionIndex == -1 ) {
    // Ой, точка не обнаружена.
    // Просто возвращаем имя файла.
    return filename;
  } else {
    return filename.substr( 0, extensionIndex );
  }
}

// Этот метод возвращает все символы после
// последней точки, если таковые имеются.
private function extractExtension( filename:String ):String {
  // Находим местоположение точки.
  var extensionIndex:Number = filename.lastIndexOf( '.' );
  if ( extensionIndex == -1 ) {
    // Ой, точка не обнаружена, поэтому
    // возвращаем пустую строку.
    return "";
  } else {
    return filename.substr( extensionIndex + 1, filename.length );
  }
}
```

Вот пример использования этой функции:

```
trace( removeExtension( "document.jpg" ) ); // На экран выводится: document
trace( removeExtension( "document" ) );     // На экран выводится: document
trace( removeExtension( "document.1.jpg" ) ); // На экран выводится: document.1
```

```
trace( extractExtension( "document.jpg" ) ); // На экран выводится: .jpg
trace( extractExtension( "document" ) );  // На экран выводится пустая строка
trace( extractExtension( "document.1.jpg" ) ); // На экран выводится: .jpg
```

См. также

Рецепт 12.6.

12.6. Как разобрать строку по словам

Задача

Требуется обработать отдельные слова строки.

Решение

Обратитесь к методу *split( )*.

Обсуждение

Метод *split( )* (см. рецепт 5.6) разбивает строку на массив, используя заданный разделитель. Для разбиения строки на слова применяется метод *split( )* с пробелом в качестве разделителя.

```
// Создаем строку, состоящую из нескольких слов.
var example:String = "Это строка слов";

// Разбиваем строку на массив слов, используя
// пробел в качестве разделителя.
var words:Array = example.split( " " );

// Перебираем элементы массива и производим над каждым
// словом какую-то операцию. В данном примере просто
// выводим значения на экран.
for ( var i:int = 0; i < words.length; i++ ) {
  /* На экран выводится:
     Это
     строка
     слов
  */
  trace( words[i] );
}
```

Слова можно обрабатывать по-разному. Далее приведен готовый пример, в котором данная техника применяется для разделения строки на слова с последующим созданием спрайтов, содержащих эти слова. Пользователь может перетаскивать слова по сцене и составлять различные предложения или выражения, как в популярных магнитных наборах для составления фраз:

```
package {

 import flash.display.Sprite;
```

```
import flash.events.MouseEvent;
import flash.text.TextField;
import flash.text.TextFieldAutoSize;
import flash.display.StageAlign;
import flash.display.StageScaleMode;

public class ActionScriptPoetry extends Sprite {

  public function ActionScriptPoetry() {

    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    // Создаем строку и разделяем ее в массив слов.
    var example:String = "Это строка слов об ActionScript";
    var words:Array = example.split(" ");
    var word:Sprite;
    var wordText:TextField;

    // Перебираем все слова массива.
    for ( var i:int = 0; i < words.length; i++ ) {

      // Создаем для каждого слова новый спрайт и добавляем его
      // в список отображения, чтобы он появился на экране
      word = new Sprite();
      addChild( word );

      // Создаем в спрайте текстовое поле путем создания
      // нового экземпляра TextField и добавления его
      // в качестве дочернего элемента
      wordText = new TextField();
      word.addChild( wordText );

      // Текстовое поле должно автоматически менять
      // размеры в соответствии со своим содержимым.
      // Оно также должно иметь рамки и фон, чтобы имитировать
      // вид магнитов со словами, и необходимо отключить
      // возможность выбора текста, поскольку предполагается
      // перемещать спрайты по сцене методом drag-and-drop.
      wordText.autoSize    = TextFieldAutoSize.LEFT; // Текст выровнен влево
      wordText.border     = true;
      wordText.background = true;
      wordText.selectable = false;

      // Задаем слова массива как значения
      // свойства text текстового поля.
      wordText.text = words[i];

      // Этот спрайт можно перетягивать
      // методом drag-and-drop.
      word.addEventListener( MouseEvent.MOUSE_DOWN, handleDrag );
      word.addEventListener( MouseEvent.MOUSE_UP, handleDrop );

      // Размещаем спрайты со словами случайным образом
      var rx:Number = Math.random() * stage.stageWidth  - word.width;
      var ry:Number = Math.random() * stage.stageHeight - word.height;
```

```
            word.x = rx;
            word.y = ry;
            trace(word);
          }
        }

        // Эта функция вызывается, если пользователь нажимает
        // кнопку мыши, когда курсор находится над словом
        private function handleDrag( event:MouseEvent ):void {
          // Целью события будет TextField, поэтому,
          // чтобы получить слово,обращаемся к родителю
          var word:Sprite = event.target.parent;

          // Делаем так, чтобы выбранное слово отрисовывалось
          // поверх всего остального
          setChildIndex( word, numChildren - 1 );

          // Перемещаем слово соответственно движениям мыши
          word.startDrag();
        }

        // Эта функция вызывается, когда пользователь
        // отпускает кнопку мыши
        private function handleDrop( event:MouseEvent ):void {
          // Целью события будет TextField, родитель - Sprite
          var word:Sprite = event.target.parent;

          // Прекращаем перемещение слова методом drag-and-drop
          word.stopDrag();
        }
      }
    }
```

Предыдущий пример применения метода *split( )* – это все, что нужно, если исходная строка содержит только слова и пробелы без знаков пунктуации. Если же в строке есть знаки препинания или другие символы, их удаляют с помощью регулярного выражения, передаваемого в качестве параметра в метод *split( )*. Регулярное выражение для выявления знаков препинания должно быть таким: `/[^a-zA-Z0-9]+/`**.**

```
// Создаем строку со знаками препинания.
var example:String = "Здесь есть несколько слов. Но, кроме того, здесь есть
знаки препинания!";

// Создаем массив слов из строки, не удаляя знаки препинания.
var words:Array = example.split( ' ' );

// Выводим на экран элементы массива.
// Некоторые из них включают знаки препинания.
// Скорее всего, это нежелательно.
for ( var i:int = 0; i < words.length; i++ ) {
  /* Выводится:
     Здесь
     есть
```

```
      несколько
      слов.
      Но,
      кроме
      того,
      здесь
      есть
      знаки
      препинания!
  */
  trace( words[i] );
}

// Задаем регулярное выражения для знаков препинания
// и используем его для создания массива слов без них
words = example.split( /[^a-zA-Z0-9]+/ );

// Выводим все элементы массива слов. На этот раз массив
// включает только слова, и ни одно из них не содержит знаков препинания.
for ( i = 0; i < words.length; i++ ) {
  /* Outputs:
      Здесь
      есть
      несколько
      слов
      Но
      кроме
      того
      здесь
      есть
      знаки
      препинания
  */
  trace( words[i] );
}
```

Регулярные выражения, рассматриваемые в главе 13, являют собой новую мощную возможность ActionScript 3.0.

См. также

Главу 13; рецепты 5.6 и 12.5.

12.7. Удаление и замена символов и слов

Задача

Требуется удалить из строки символы или заменить одну подстроку другой.

Решение

Это делается при помощи метода *replace( )* или сочетания методов *split( )* и *join( )*.

Обсуждение

ActionScript 3.0 включает новый метод *String.replace( )*, которого не было в предыдущих версиях. Он позволяет заменять части строки. Этот метод принимает два параметра:

`pattern` (шаблон)

Подстрока или регулярное выражение, которое требуется найти и заменить.

`replace` (заменить)

Значение, которым заменяется подстрока, соответствующая `pattern`. Обычно это строка, но также может использоваться функция, возвращающая строку. Это полезно, когда `pattern` является регулярным выражением.

Есть два варианта этого метода, отличающиеся параметрами. Данный рецепт основное внимание уделяет использованию строки в качестве шаблона. Работа с регулярными выражениями рассматривается в рецепте 13.4.

Вот простой пример замены подстроки в предложении. Метод *replace( )* возвращает новую строку, в которой подстрока, соответствующая параметру `pattern`, заменяется подстрокой, заданной параметром `replace`. Исходная строка остается без изменений.

```
var example:String = "This is a cool sentence."

// Заменяем в example " is " на " is not ".
// На экран выводится: This is not a cool sentence.
trace( example.replace( " is ", " is not " ) );
```

В данном примере слово «is» в шаблоне окружено пробелами. Это важно, потому что в случае отсутствия пробелов шаблону соответствует и часть «is» слова «This», и тогда предложение будет начинаться с «This not is», а мы совсем не этого добивались.

Если в качестве параметра `pattern` передавать строку, а не регулярное выражение, то функциональные возможности метода *replace( )* сужаются. Два самых неприятных момента: заменяется только первая подстрока, соответствующая шаблону, и приходится реализовывать собственную подстановку, нечувствительную к регистру.

Чтобы заменить все встречающиеся в строке соответствия шаблону, необходимо вызывать метод *replace( )* в цикле:

```
// Создаем исходную строку и еще одну строку
// для хранения замещений
```

```
var example:String = "Это – птица, это – самолет, это – ActionScript-мэн!";
var replaced:String = example; // Инициализируем replaced и сохраняем
                               // в ней исходный текст

// Если находим в replaced подстроку pattern, проводим замену
while ( replaced.indexOf( "это" ) != -1 ) {
  // Замещаем первый экземпляр "это" строкой "вот это".
  replaced = replaced.replace( "это", "вот это" );
}

// Чтобы решить проблему чувствительности к регистру,
// необходимо изменить pattern и повторить процесс
// замещения. Таким образом, замещаем вариант "Это"
replaced = replaced.replace( "Это", "Вот это" );

// Выводится: Вот это – птица, вот это – самолет, вот это – ActionScript-мэн!
trace( replaced );
```

Метод *split( )* тоже может применяться для замены и удаления символов и слов строки. В отличие от метода *replace( )* со строковым шаблоном, *split( )* замещает все экземпляры слова. Но проблема чувствительности к регистру общая для обоих методов. Далее приведен пример замены HTML-тега
 символом \n для соответствующего разбиения предложения на несколько строк:

```
var example:String = "Это<br>предложение<br>на 3 строки";

// Удаляем теги <br> и замещаем их символами перевода строки ('\n')
/* На экран выводится:
   Это
   предложение
   на 3 строки
*/
trace( example.split( "<br>" ).join( '\n' ) );
```

После разделения строки возвращается массив строк. Новую строку из отдельных элементов массива можно создать посредством метода *Array.join( )*. Разделение уничтожает разделитель, что дает возможность объединить элементы в новую строку с новым разделителем, который передается в метод *join( )* как параметр. Так можно заменить все экземпляры разделителя, переданного в *split( )*.

Чтобы не заместить, а удалить символы или слова, в качестве строки замещения задают пустую строку:

```
var example:String = "Это классное предложение.";
// Удаляем из example слово "классное" и пробел после него
// На экран выводится: Это предложение.
trace( example.replace( "классное ", "" ) );
```

См. также

Рецепты 12.4 и 12.7.

12.8. Как извлекать символы по одному

Задача

Требуется извлекать из строки символы по одному.

Решение

Это делается при помощи выражения *for* и метода *String.charAt( )*. Также можно прибегнуть к методу *String.split( )* с пустой строкой в качестве разделителя для разделения строки в массив символов и затем перебирать все элементы массива в цикле *for*.

Обсуждение

Проще всего извлечь все символы строки с помощью выражения *for*, которое проходит по всей строке от нулевого индекса до `string.length - 1` с единичным шагом. Если требуется извлечь символ для последующей обработки, в теле выражения *for* можно вызывать метод *charAt( )*:

```
var example:String = "строка";

// Перебираем все символы строки.
for ( var i:int = 0; i < example.length; i++ ) {
  /* Выводим все символы по одному:
     с
     т
     р
     о
     к
     а
  */
  trace( example.charAt( i ) );
}
```

Того же результата можно добиться, сначала разделяя строку в массив символов методом *split( )*, и затем перебирая массив и обрабатывая каждый символ. Для разделения символов в методе *split( )* в качестве параметра-разделителя задают пустую строку.

```
var example:String = "строка";

// Разделяем строку в массив символов (односимвольные строки).
var characters:Array = example.split( "" );

// Перебираем все элементы массива characters.
for ( var i:int = 0; i < characters.length; i++ ) {
  /* Выводим все элементы-символы. На экран выводится:
     с
     т
     р
     о
     к
```

```
    a
  */
  trace( characters[i] );
}
```

Обе техники в целом взаимозаменяемы, хотя вторая предлагает некоторые преимущества, если требуется работать с символами, используя общие методы массивов. Например, если сначала разделить строку в массив символов, этот массив можно сортировать. Это не так просто сделать в случае применения *charAt( )*:

```
var example:String = "строка";

var characters:Array = example.split( "" );

// Сортируем массив символов в алфавитном порядке.
characters.sort();

for ( var i:int = 0; i < characters.length; i++) {
  /* На экран выводится:
     а
     к
     о
     р
     с
     т
  */
  trace( characters[i] );
}
```

Также если требуется посредством этого процесса удалить все экземпляры определенного символа, то проще работать с массивом, чем с методом *charAt( )*:

```
var example:String = "строка";

var characters:Array = example.split( "" );

for ( var i:Number = 0; i < characters.length; i++ ) {
  // Удаляем из массива все элементы "р". Не забываем
  // уменьшать i при удалении элемента; в противном
  // случае следующий элемент будет пропущен.
  if ( characters[i] == "р") {
    characters.splice( i, 1 );
    i--;
  }
}

// На экран выводится: стока
trace( characters.join( "" ) );
```

Данная техника замещения символов годится для простых случаев. Ее альтернативы для более сложных вариантов рассмотрены в рецепте 12.7.

См. также

Рецепт 12.7.

12.9. Изменение регистра

Задача

Требуется изменить регистр строки или провести нечувствительное к регистру сравнение.

Решение

Задача решается при помощи методов *toUpperCase( )* и *toLowerCase( )*.

Обсуждение

Методы *toUpperCase( )* (в верхний регистр) и *toLowerCase( )* (в нижний регистр) возвращают новую строку, в которой все символы переведены в верхний или нижний регистр, соответственно, без изменения исходной строки. Это полезно, когда требуется обеспечить единообразие регистра. Например, методы *toLowerCase( )* или *toUpperCase( )* могут применяться для проведения нечувствительного к регистру поиска, как показано в рецепте 12.4. Оба метода работают только с буквенными символами, не изменяя остальные:

```
var example:String = "Какой регистр?";

// На экран выводится: какой регистр?
trace( example.toLowerCase() );

// На экран выводится: КАКОЙ РЕГИСТР?
trace( example.toUpperCase() );

// Исходная строка остается неизменной: Какой регистр?
trace( example );
```

Оба метода возвращают новую строку. Чтобы изменить исходную строку, необходимо присвоить ей возвращаемое значение:

```
var example:String = example.toLowerCase();
```

Применяя методы *toLowerCase( )* и *toUpperCase( )* совместно, можно изменять регистры символов так, чтобы, например, слова начинались с прописной буквы. Именно это делает метод *ascb.util.StringUtilities.toInitialCap( )*. Ниже приведен фрагмент кода класса *StringUtilities*, в котором описывается метод *toInitialCap( )*:

```
public static function toInitialCap( original:String ):String {
  return original.charAt( 0 ).toUpperCase() + original.substr( 1
).toLowerCase();
}
```

Пример применения этого метода:

```
var example:String = "бРюС";
trace( StringUtilities.toInitialCap( example ) ); // На экран выводится: Брюс
```

Метод *toTitleCase( )* (в заголовок) преобразовывает строку к так называемому регистру «каждое слово с прописной» (title case). Описание метода:

```
public static function toTitleCase( original:String ):String {
  var words:Array = original.split( " " );
  for (var i:int = 0; i < words.length; i++) {
    words[i] = toInitialCap( words[i] );
  }
  return ( words.join( " " ) );
}
```

Простой пример:

```
var example:String = "сборник рецептов actionScript";

// На экран выводится: Сборник Рецептов ActionScript
trace( StringUtilities.toTitleCase( example ) );
```

См. также

Рецепт 12.4.

12.10. Как удалить лишние пробелы

Задача

Требуется удалить лишние пробелы в начале или конце строки.

Решение

Обратитесь к специальному методу *ascb.util.StringUtilities.trim( )*. А при работе в инфраструктуре Flex 2 можно использовать статический метод *mx.utils.StringUtil.trim( )*.

Обсуждение

Лишние пробелы в начале и конце строки – явление обычное и достаточно неприятное, поэтому необходим способ избавления от них. В ActionScript метод *trim( )* не реализован, поэтому приходится или писать собственный метод, или брать готовое решение.

Данная операция включает следующие основные этапы:

1. Преобразование строки в массив символов методом *split( )*.
2. Удаление всех пробельных элементов (табуляция, перевод страницы, возврат каретки, перевод строки или пробел) в начале массива.
3. Удаление пробельных элементов в конце массива.

4. Объединение элементов массива в единую строку и возвращение этого нового значения с помощью метода *join( )*.

Все это выполняет метод *ascb.util.StringUtilities.trim( )*. Ниже показано реальное описание этого метода и вспомогательного метода *isWhitespace( )*:

```
// Возвращает true, если символ является пробельным символом
public static function isWhitespace( ch:String ):Boolean {
  return ch == '\r' ||
         ch == '\n' ||
         ch == '\f' ||
         ch == '\t' ||
         ch == ' ';
}

public static function trim( original:String ):String {

  // Превращает строку в массив элементов.
  var characters:Array = original.split( "" );

  // С помощью метода splice() удаляем все пробельные
  // элементы в начале массива. По достижении символа,
  // не являющегося пробельным, осуществляется выход
  // из цикла с помощью выражения break, чтобы предотвратить
  // уничтожение пробелов в середине строки.
  for ( var i:int = 0; i < characters.length; i++ ) {
    if ( isWhitespace( characters[i] ) ) {
      characters.splice( i, 1 );
      i--;
    } else {
      break;
    }
  }

  // Перебирает элементы массива с конца, удаляя
  // пробельные элементы до тех пор, пока не встретится
  // непробельный символ. Тогда выходит из цикла
  // с помощью выражения break.
  for ( i = characters.length - 1; i >= 0; i-- ) {
    if ( isWhitespace( characters[i] ) ) {
      characters.splice( i, 1 );
    } else {
      break;
    }
  }

  // Восстанавливает строку с помощью метода join()
  // и возвращает результат.
  return characters.join("");
}
```

Пример применения метода *trim( )*.

```
// Создаем строку с лишними пробелами в начале и в конце.
var example:String = "\n\r\f\tстрока\t\t\n\n";

/* Перед вызовом метода trim() выводим на экран значение. На экран выводится:
Значение этой строки:

    строка

<end>
*/
trace( "значение этой строки: " + example + "<end>" );

// Присваиваем example значение,
// возвращенное методом trim().
example = StringUtilities.trim( example );

// Теперь, используя такое же выражение trace(),
// снова выводим на экран значение.
//На экран выводится: значение этой строки: строка<end>
trace( "значение этой строки: " + example + "<end>" );
```

Реализация метода *trim( )* в инфраструктуре Flex 2 практически аналогична. Вот пример кода:

```
// На экран выводится: строка<end>
trace( StringUtil.trim( "\n  \r\tстрока\t\t\n\n" ) + "<end>" );
```

12.11. Изменение порядка слов или символов в строке на обратный

Задача

Требуется изменить в строке порядок слов или символов на обратный.

Решение

Надо взять метод *split( )* для создания массива слов/символов и применить к этому массиву методы *reverse( )* и *join( )*.

Обсуждение

Порядок слов или символов в строке можно изменить посредством той же процедуры. Только разделители в методах *split( )* и *join( )* будут другими. В любом случае основной алгоритм таков:

1. Преобразование строки в массив методом *split( )* с использованием пробела или пустой строки в качестве разделителя символов.
2. Вызов метода *reverse( )* массива, который меняет порядок элементов на обратный.
3. Применение метода *join( )* для воссоздания строки. При изменении порядка слов разделителем будет пробел; при изменении порядка символов разделителем будет пустая строка.

Следующий код иллюстрирует этот процесс:

```
var example:String = "здравствуй дорогой читатель";

// Превращаем строку в массив слов.
var words:Array = example.split( " " );

// Меняем порядок элементов массива на обратный.
words.reverse();

// Объединяем элементы массива в строку, используя пробелы.
var exampleRevByWord:String = words.join( " " );

// На экран выводится: читатель дорогой здравствуй
trace( exampleRevByWord );

// Превращаем строку в массив символов.
var characters:Array = example.split( "" );

// Меняем порядок элементов массива на обратный.
characters.reverse();

// Объединяем элементы массива в строку, используя пустую строку.
var exampleRevByChar:String = characters.join( "" );

// На экран выводится: ьлетатич йогород йувтсвардз
trace( exampleRevByChar );
```

12.12. Преобразование строк в коды Unicode или ASCII и наоборот

Задача

Требуется преобразовать символы в соответствующие им коды Unicode (также известные как коды символов) или наоборот. Или требуется преобразовывать строки в/из кодов ASCII.

Решение

Это делается при помощи методов *String.charCodeAt( )* и *String.fromCharCode( )*.

Обсуждение

Для отображения символов, которые нельзя ввести в документ Flash напрямую, можно использовать метод *fromCharCode( )* (из символьного кода). Это статический метод. То есть он вызывается из объекта *String* верхнего уровня, а не из экземпляра строки. Он принимает целое число или множества целых чисел и преобразует коды символов в их строковые эквиваленты. Если их значения меньше 128, то *fromCharCode( )* фактически преобразует числовой код ASCII в эквивалентный символ:

```
/* На экран выводится:
   Новый параграф: &#182;
   Центы: &#162;
   Имя: Darron
*/
trace( "Новый параграф: " + String.fromCharCode( 182 ) );
trace( "Центы: " + String.fromCharCode( 162 ) );
trace( "Имя: " + String.fromCharCode( 68, 97, 114, 114, 111, 110 ) );
```

Для извлечения кода символа строки по определенному индексу предназначен метод *charCodeAt( )*. Символы, код Unicode которых меньше 128, *charCodeAt( )* преобразует в эквивалентные им с таким же кодом ASCII:

```
var example:String = "abcd";

// Выводится код, 97, первого символа, a.
trace( example.charCodeAt( 0 ) );
```

Метод *fromCharCode( )* представляет собой альтернативу отображению специальных символов посредством кодов Unicode. Однако *fromCharCode( )* может совместно с *charCodeAt(*, применяться для проверки существования специальных символов:

```
var example:String = String.fromCharCode( 191 ) + "D"
           + String.fromCharCode( 243 ) + "nde est"
           + String.fromCharCode( 225 ) + " el ba"
           + String.fromCharCode( 241 )  + "o?";

// Проверяем, соответствует ли первый символ коду 191.
// Для отображения специального символа используем код
// символа, а не метод fromCharCode( 191 ). Если да,
// на экран выводится: The string "&#191;D&#243;nde est&#225;
// el ba&#241;o?" has a &#191; at the beginning.
if ( example.charCodeAt( 0 ) == 191 ) {
  trace( "The string \"" + example + "\" has a \u00BF at the beginning." );
}
```

Для преобразования строки в коды символов и наоборот можно совместно использовать методы *charCodeAt( )* и *fromCharCode( )*.

Следующие методы пригодятся для создания игр в шифрование, но они небезопасны и не должны применяться к уязвимым данным:

```
public static function encode( original:String ):String {

  // Свойство codeMap класса StringUtilities определяется
  // при первом вызове метода encode().
  // Поэтому, если codeMap еще не определен, его надо создать.
  if ( codeMap == null ) {

    // Свойство codeMap – это ассоциативный массив, сопоставляющий
    // исходный код каждого символа с кодом другого символа.
    codeMap = new Object();
```

```
    // Создаем массив всех кодов символов от 0 до 255.
    var originalMap:Array = new Array();
    for ( var i:int = 0; i < 256 ; i++ ) {
      originalMap.push( i );
    }

    // Создаем временный массив, являющийся копией массива originalMap.
    var tempChars:Array = originalMap.concat();

    // Перебираем все коды символов в originalMap.
    for ( var i:int = 0; i < originalMap.length; i++ ) {

      //Создаем случайное число от 0 до последнего индекса tempChars.
      var randomIndex:int = Math.floor( Math.random() *
                                          ( tempChars.length - 1 ) );

      // Присваиваем codeMap значения таким образом, чтобы
      // ключи являлись исходными кодами, а значения –
      // кодами, с которыми они должны быть сопоставлены.
      codeMap[ originalMap[i] ] = tempChars[ randomIndex ];

      // Удаляем из tempChars элементы, которые только что
      // были присвоены codeMap. Это предотвращает дублирование.
      tempChars.splice( randomIndex, 1 );
    }
  }

  // Преобразуем строку в массив символов.
  var characters:Array = original.split("");

  // Заменяем каждый символ массива соответствующим значением из codeMap.
  for ( i = 0; i < characters.length; i++ ) {
    characters[i] = String.fromCharCode(
                              codeMap[ characters[i].charCodeAt( 0 ) ] );
  }

  // Возвращаем кодированную строку.
  return characters.join( "" );
}

public static function decode( encoded:String ):String {

  // Превращаем кодированную строку в массив символов.
  var characters:Array = encoded.split( "" );

  // Свойству reverseCodeMap присваивается значение
  // при первом выполнении метода decode(). Поэтому, если
  // reverseCodeMap до сих пор не определено, его надо создать.
  if ( reverseCodeMap == null ) {

    // Создаем ассоциативный массив, который меняет
    // порядок ключей и значений codeMap на обратный.
    // Это позволяет проводить поиск в обратном
    // направлении на основании кодированных символов,
    // а не оригинальных.
    reverseCodeMap = new Object();
```

```
    for ( var key in codeMap ) {
      reverseCodesMap[ codeMap[key] ] = key;
    }
  }

  // Перебираем все символы массива и заменяем их
  // соответствующими значениями reverseCodeMap, таким
  // образом, восстанавливая исходные значения символов.
  for ( var i:int = 0; i < characters.length; i++ ) {
    characters[i] = String.fromCharCode( reverseCodeMap[
characters[i].charCodeAt( 0 ) ] );
  }

  // Возвращаем дешифрованную строку.
  return characters.join( "" );
}
```

Далее приведен простой пример применения:

```
var example:String = "Peter Piper picked a peck of pickled peppers.";

// С помощью метода encode() создаем шифрованную версию example.
var encoded:String = StringUtilities.encode( example );

// Выводим значение шифрованной строки. Она будет
// генерироваться случайным образом при каждом запуске
// ролика. Это будет выглядеть примерно так:
// жT#Tm&#239;ж~*Tm&#239;*~NcT&#173;&#239;?&#239;*TNc&#239;?2&#239;*~Nc?T&#
// 173;&#23 9;*T**Tm:V
trace( encoded );

// Выводим значение, возвращенное методом decode().
// На экран выводится: Peter Piper picked a peck of pickled peppers.
trace( StringUtilities.decode( encoded ) );
```

13

Регулярные выражения

13.0. Введение

Регулярные выражения (*regular expressions*)[1] – одна из самых мощных возможностей, появившихся в ActionScript 3.0. По сути регулярные выражения представляют собой шаблоны, служащие для сопоставления строк. Читателю должны быть хорошо известны другие типы шаблонов, такие как групповые символы (например, `*` и `?`), которые могут применяться при поиске файлов. Шаблоны также упоминаются в рецепте 9.5. Кроме такого сопоставления шаблонов регулярные выражения обеспечивают и намного более сложные.

Регулярные выражения могут быть полезны во многих ситуациях. Например, шаблоны применяются при работе со строками для осуществления разнообразных задач, включая:

- Поиск подстрок, начинающихся с гласной (а, е, и, о, у, э, ю или я)
- Извлечение из строки определенных значений, например значение года из полной даты
- Проверку действительности вводимых пользователем данных, чтобы гарантировать соответствующий формат адреса электронной почты
- Удаление HTML-тегов из блока текста для устранения разметки

Шаблоны, используемые для регулярных выражений, создаются путем сочетания символов, имеющих специальное значение. Шаблоны могут быть как очень простыми:

```
[a-zA-Z]
```

[1] В англоязычной литературе часто используют сокращенные названия *regexes* или *regexps*.

так и крайне сложными и непонятными, такими как данное выражение для определения действительности IP-адреса:

```
^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-
5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])$
```

Простые шаблоны, такие как `.*`, легки для понимания, но встречаются и более замысловатые, сложные для восприятия и еще более сложные в реализации. К счастью, любое регулярное выражение можно охарактеризовать простым описанием. Например, регулярное выражение `.*` означает «любой символ, повторяющийся любое количество раз». Более сложные шаблоны, такие как `(A|a)ction(S|s)cript`, ничем не отличаются, потому что образованы простыми шаблонами. Шаблон `(A|a)ction(S|s)cript` означает «прописная или строчная 'a', за которой следует строка 'ction', за которой следует прописная или строчная 's', за которой следует строка 'cript'», и может использоваться для поиска экземпляров подстроки «ActionScript» (и ее разновидностей, таких как «actionScript») в строке.

Овладеть синтаксисом регулярных выражений непросто. Но как и в любых других случаях, изучение базовых принципов играет важную роль, а знание и понимание регулярных выражений – это мощное оружие, которое никогда не помешает иметь в своем профессиональном арсенале.

Более глубоко изучить регулярные выражения помогут книги «Mastering Regular Expressions» Джеффри Фридла (Jeffrey E.F. Friedl) (O'Reilly) или «Regular Expressions Pocket Reference» Тони Стаббелбайна (Tony Stubblebine) (O'Reilly). В этих книгах есть все, что необходимо знать о регулярных выражениях.

Регулярные выражения, какими бы сложными они ни были, очень просто применять в ActionScript. В этой главе основное внимание уделяется распространенным применениям регулярных выражений в контексте Flash и ActionScript, и она не претендует на роль исчерпывающего руководства.

13.1. Понимание шаблонов регулярных выражений

Задача

Требуется научиться понимать основные элементы регулярных выражений.

Решение

Регулярные выражения образованы сочетанием символов, имеющих специальное значение. Начинайте с изучения базовых шаблонов и затем применяйте полученные знания к более сложным.

Обсуждение

Регулярное выражение – это *шаблон*, созданный в соответствии с правилами синтаксиса регулярных выражений и обычно используемый при обработке текстов и сопоставлении шаблонов. Синтаксис включает символы, *метасимволы* и *метапоследовательности*. Символы интерпретируются буквально, тогда как метасимволы и метапоследовательности в контексте регулярных выражений имеют специальное значение. Так, регулярное выражение, состоящее из символов `привет`, соответствует строке «привет», тогда как регулярное выражение из одного метасимвола `.` означает «любой символ» и соответствует «а», «б», «1» и т. д. А регулярное выражение, образованное метапоследовательностью `\d`, соответствует любой цифре, например «1» или «9».

Прежде чем углубиться в синтаксис регулярных выражений, посмотрим, как они создаются в ActionScript 3.0. Регулярные выражения создаются с помощью класса *RegExp* и строки, описывающей шаблон, или *литерал регулярного выражения*. Литерал регулярного выражения – это шаблон регулярного выражения, ограниченный с обеих сторон прямыми слэшами, например / *шаблон*/. В следующем фрагменте кода продемонстрировано, как создать регулярное выражение шаблона `привет` с помощью как строки с конструктором *RegExp*, так и литерала регулярного выражения:

```
// Создаем шаблон 'привет', используя конструктор класса
// и передавая в него строку, описывающую шаблон
var example1:RegExp = new RegExp( "привет" );

// Создаем такой же шаблон, используя литерал регулярного выражения
var example2:RegExp = /привет/;
```

Оба регулярных выражения, `example1` и `example2`, соответствуют одному и тому же шаблону, строке «привет». В большинстве случаев шаблоны можно создавать любым из этих двух способов. Однако если шаблон регулярного выражения включает обратный слэш (\), вариант со строкой и конструктором *RegExp* становится более сложным.

Поскольку объект *RegExp* создается путем передачи строки в конструктор, все вхождения последовательности `\\` в строке должны быть экранированы, т. е. записаны как `\\\\`. Кроме того, `\\` – это еще и специальный символ шаблонов *RegExp*, поэтому для организации поиска обратного слэша в регулярных выражениях экранирование осуществляется так: `\\\\\\\\`.

Обратный слэш обозначает начало экранированной последовательности в строке (см. рецепт 12.3) и теряет свое значение в контексте регулярного выражения. То есть обратный слэш прежде всего интерпретируется как специальный символ строки. Поэтому, если требуется создать шаблон для выявления обратного слэша с помощью строки и конструктора класса, в строке должен использоваться двойной обратный слэш. В литерале регулярного выражения такой проблемы не возникает:

```
// Создаем регулярное выражение для выбора цифр
// (обратите внимание на двойной обратный слэш)
var example1:RegExp = new RegExp( "\\d" );

// Создаем регулярное выражение для выбора цифр
var example2:RegExp = /\d/;

// Создаем регулярное выражение для выбора обратного слэша
var example3:RegExp = new RegExp("\\\\");

// Создаем регулярное выражение для выбора обратного слэша
Var example4:RegExp = /\\/;
```

Создавать регулярные выражения лучше с помощью литералов. Далее в данной главе рассматривается только этот способ.

На данный момент мы знаем, что символы в шаблоне регулярного выражения интерпретируются буквально. Комбинируя метасимволы и метапоследовательности с обычными символами, можно создавать мощные сочетания, служащие для представления многих типов шаблонов. Теперь посмотрим, что означают метасимволы и как они могут применяться.

В табл. 13.1 представлены метасимволы регулярных выражений. Если метасимвол должен использоваться буквально, ему должен предшествовать обратный слэш. Например, для поиска открывающей фигурной скобки используется регулярное выражение `\{`.

Таблица 13.1. Метасимволы регулярных выражений

Выражение	Значение	Пример
`?`	Представляет предыдущий символ ноль или один раз (т.е. предыдущий символ является необязательным)	`ta?k` соответствует *tak* или *tk*, но не *tik* или *taak*
`*`	Представляет предыдущий символ ноль или более раз	`wo*k` соответствует *wok*, *wk* или *woook*, но не *wak*
`+`	Представляет предыдущий символ один или более раз	`craw+l` соответствует *crawl* или *crawwwl*, но не *cral*
`.` (точка)	Представляет любой символ, кроме символа перевода строки (если только не установлен флаг *dotall*)	`c.ow` соответствует *crow* или *clow*, но не *cow*
`^`	Соответствует началу строки (также началу строки текста, если установлен флаг *multiline*)	`^wap` соответствует *wap*, но не *swap*
`$`	Соответствует концу строки (также соответствует позиции перед символом перевода строки «`\n`», если установлен флаг *multiline*)	`ow$` соответствует *ow*, но не *owl*

Выражение	Значение	Пример
`|`	Определяет выбор левой или правой части цепочки	`one|two` соответствует *one* или *two*, но не *ten*
`\`	Экранирует специальное значение метасимвола, следующего за обратным слэшем	`\.` соответствует точке, а не «любому символу», как это было бы, если бы использовался просто метасимвол `.`, без обратного слэша
`(` и `)`	Создают группы в рамках регулярного выражения для: • Определения области действия `|` • Определения области действия `{` и `}` • Использования обратных ссылок, где `\1` ссылается на все, что соответствует образцу в первой группе, и т. д.	`l(o|a)g` соответствует *log* или *lag*, но не *lug* `a(b){1,2}` соответствует *ab* или *abb*, но не *a* `(a|b)\1` соответствует *aa* или *bb*, но не *ab* или *ba*
`[` и `]`	Представляет любой одиночный символ из числа заключенных в скобки, т.е. класс символов. • Наличие тире (−) указывает на диапазон символов • Знак вставки (^) определяет класс недопустимых символов (допускаются любые символы, кроме определенных) • Метасимволы (кроме тире и знака вставки в начале) не требуют экранирования обратным слэшем	`l[oa]g` соответствует *log* или *lag*, но не *lug* `[a-z]` соответствует любому строчному символу, например *a* или *h*, но не *1*, *2*, или *F* `l[^oa]g` соответствует *lug*, но не *lag* или *log* `[+\-]` соответствует + или -

Аналогично метасимволам в табл. 13.2 приводится список метапоследовательностей с описаниями и примерами.

Таблица 13.2. Метапоследовательности регулярных выражений

Выражение	Соответствия	Пример
`{n}`	Точно *n* экземпляров предыдущего символа или группы	`Cre{2}l` соответствует *creel*, но не *crel* или *creel*
`{n,}`	По крайней мере, *n* экземпляров предыдущего символа или группы	`Cre{2,}l` соответствует *creel* или *creeeel*, но не *crel*
`{n,m}`	По крайней мере, *n*, но не более *m* экземпляров предыдущего символа или группы	`Cre{2,3}l` соответствует *creel* или *creeel*, но не *crel* или *creel*
`\A`	В начале строки; аналогично (^)	`\Awap` соответствует *wap*, но не *swap*

Таблица 13.2 (продолжение)

Выражение	Соответствия	Пример
\b	Границы слова	\b7\b соответствует *7, но не 71* или *573*
\B	Не границы слова	\B7\B соответствует *573*, но не *71* или *7*, или *37*
\d	Любой цифровой символ; аналогично [0-9]	a\d соответствует *a1* и *a8*, но не *ab* или *ad*
\D	Любой нецифровой символ; аналогично [^0-9]	a\D соответствует *aB* и *ak*, но не *a8* или *a1*
\n	Символ новой строки	a\nb соответствует "a\nb"
\r	Символ возврата	a\rb соответствует "a\rb"
\s	Один пробельный символ (пробел, табуляция, перевод строки или перевод страницы)	King\sTut соответствует *King Tut* и *King\tTut*
\S	Один непробельный символ	\STut соответствует *gTut*, но не *Tut*
\t	Символ табуляции	a\tb соответствует "a\tb"
\unnnn	Символ Unicode, шестнадцатеричный код которого равен *nnnn*	\u000a соответствует "\n"
\w	Любая буква, цифра или символ подчеркивания; аналогично [A-Za-z0-9_]	a\wm соответствует *arm* и *a8m*, но не *a m* или *aйm*
\W	Любой символ, не являющийся буквой, цифрой или символом подчеркивания; аналогично [^A-Za-z0-9_]	a\Wm соответствует *a m* или *aйm*, но не *a7m* или *aim*
\xnn	Символ ASCII, шестнадцатеричный код которого равен *nn*	\x0a соответствует "\n"
\Z	Конец строки; сравнивается строка *до* символа разрыва строки, если таковой присутствует	ab\Z соответствует "ab\n" и ab, но не "ab\nc"
\z	Конец строки; сравнивается строка, *включая* символ разрыва строки, если таковой присутствует	ab\z соответствует ab, но не "ab\n" или "ab\nc"

В табл. 13.1 и 13.2 описаны основные синтаксические правила регулярных выражений. Сочетая символы, метасимволы и метапоследовательности, можно создавать огромное множество разнообразных шаблонов. Но это еще не все.

Регулярные выражения также могут включать определенные флаги, указывающие на необходимость дополнительной обработки шаблона. Существует пять флагов, которые представлены свойствами объекта: global (глобальный), ignoreCase (игнорировать регистр), multiline (многострочный), dotall и extended (расширенный).

Флаги должны быть заданы при создании выражения. Попытки изменить флаг уже существующего экземпляра *RegExp* приводят к ошибке компиляции:

```
// В режиме строгого контроля типов формируется ошибка
// компиляции: Свойство доступно только для чтения
example.global = true;
```

Флаги можно задать двумя способами в зависимости от того, как создается регулярное выражение. Если используется конструктор *RegExp*, в него можно передать второй параметр – строку, в которой перечислены флаги регулярного выражения. В случае применения литерала регулярного выражения флаги замыкают выражение и стоят после последнего слэша:

```
// Создаем регулярное выражение с флагами
// global и ignoreCase
var example1:RegExp = new RegExp( "привет", "gi" );

// Создаем регулярное выражение с флагами
// global и ignoreCase
var example2:RegExp = /привет/gi;
```

По умолчанию, если флаг не задан явно при создании регулярного выражения, для него устанавливается значение `false`. В табл. 13.3 перечислены различные флаги и их значения.

Таблица 13.3. Флаги регулярных выражений

Флаг	Значение	Пример
`g` (global)	Более одного соответствия	`/the/g` задает поиск всех вхождений *the*
`i` (ignoreCase)	Осуществляется сравнение `[a-z]` и `[A-Z]` без учета регистра (и специальных символов, как й)	`/a/i` соответствует *a* и *A*
`m` (multiline)	Символ `^` соответствует концу строки; символ `$` соответствует началу строки	`/^a/m` соответствует и \n*a*, и *a*
`s` (dotall)	`.` соответствует любому символу, включая и символ перевода строки `\n`	`/a./s` соответствует и *a*\n, и *ab*
`x` (extended)	В регулярном выражении допускаются пробелы, игнорируемые шаблоном, что обеспечивает возможность более понятной записи шаблона	`/a \d/x` соответствует *a2*, но не *a 2* (с пробелом между символами)

Чаще других применяются флаги `ignoreCase` и `global`, но флаг `extended` может быть полезен для понимания регулярных выражений. Если установить флаг `extended`, в регулярное выражение можно включать дополнительные пробелы для выделения различных частей, его образующих; например:

```
var example1:RegExp = /(a(b)*){2,}/

// Используем флаг extended, чтобы сделать выражение
// немного более удобным для чтения
var example2:RegExp = /(a (b)* ){2,}/x;
```

Этот код создает регулярное выражения для «a, за которой следует b, повторяющаяся любое количество раз, а все выражение должно повторяться, по крайней мере, 2 раза». Этому соответствуют строки «abba» и «abbbabbbbbbbb», но не «abbb».

Главное, необходимо запомнить, что любое регулярное выражение можно разбить на эти базовые строительные блоки. Понимание этого и знание принципов анализа сложных шаблонов поможет избежать многих неприятностей при изучении регулярных выражений. Регулярные выражения заслуживают потраченного на их изучение времени, и как только вы овладеете ими, они докажут свою ценность.

См. также

Рецепты 9.5 и 12.3. Справочную информацию по регулярным выражениям можно найти по адресу *http://www.regular-expressions.info*. В книге Джеффри Фридла (Jeffrey Friedl) «Mastering Regular Expressions» (O'Reilly)[1] вы найдете массу практических примеров применения регулярных выражений. Как хороший справочник по этой теме можно рекомендовать и книгу Тони Стаббелбайна (Tony Stubblebine) «Regular Expressions Pocket Reference» (O'Reilly).

13.2. Тестирование регулярных выражений

Задача

Требуется протестировать регулярное выражение и выяснить, можно ли найти соответствия.

Решение

Это делается при помощи методов *RegExp.test( )* и *RegExp.exec( )*.

Обсуждение

В рецепте 13.1 обсуждаются строительные блоки регулярных выражений. Предположим, регулярное выражение, описывающее шаблон того, что необходимо найти в строке, создано. Класс *RegExp* предлагает два метода, которые могут осуществлять поиск: *test( )* и *exec( )*.

Метод *test( )* экземпляра *RegExp* применяется для поиска соответствий шаблону в строке. Метод *test( )* в качестве параметра принимает

[1] Дж. Фридл «Регулярные выражения», 3-е изд. – СПб: Символ-Плюс, 2008.

тестируемую строку и возвращает Булево значение `true`, если соответствие найдено, и `false` в противном случае:

```
// Создаем шаблон для сопоставления со строкой
var example:RegExp = /abc/;

// На экран выводится: true
trace( example.test( "Строка, содержащая abc" ) );

// На экран выводится: true
trace( example.test( "abc" ) );

// На экран выводится: false
trace( example.test( "Другая строка для проверки..." ) );
```

Метод *test( )* возвращает `true`, если находит в строке соответствие регулярному выражению, но он не позволяет извлечь соответствующую часть строки или определить ее местонахождение в строке. Для получения этой информации служит метод *exec( )*.

Принцип работы метода *exec( )* во многом аналогичен методу *test( )*, за исключением того, что он возвращает не Булево значение, а массив, содержащий подстроку, соответствующую шаблону. Если соответствий не выявлено, *exec( )* возвращает `null`.

```
// Создаем шаблон для сопоставления со строкой
var example:RegExp = /abc/;

// Сохраняем результат проверки строки, содержащей
// соответствующую шаблону подстроку
var result:Array = example.exec( "Строка, содержащая abc" );

// На экран выводится: abc
trace( result );
// На экран выводится: 14
trace( result.index );

// Сохраняем результат проверки строки,
// не содержащей соответствий шаблону
result = example.exec( "Строка без соответствий" );

// На экран выводится null
trace( result );
```

По умолчанию метод *exec( )* начинает поиск с начала строки и возвращает массив, содержащий первое выявленное соответствие, или `null`, если таковое не обнаружено. Если соответствие найдено, возвращаемый массив содержит два свойства:

- `index`, который указывает на начало найденной подстроки, соответствующей шаблону.
- `input` (ввод) – строку, которая сравнивается с регулярным выражением.

В большинстве случаев возвращаемый из *exec( )* массив содержит один элемент – подстроку, соответствующую регулярному выражению. Од-

нако если определенные элементы шаблона сгруппированы посредством круглых скобок, каждая подстрока, соответствующая такой группе, будет отдельным элементом возвращаемого результата:

```
var example:RegExp = /(\d)abc(\d*)/;

var result:Array = example.exec( "7abc" );

// На экран выводится: 3
trace( result.length );
// На экран выводится: 7abc
trace( result[0] );
// На экран выводится: 7
trace( result[1] );
// На экран выводится:
trace( result[2] );
```

Фигурирующий в данном примере шаблон содержит две группы: одну – для выбора «цифр», а другую – для выбора «цифр, повторяющихся любое количество раз, включая 0». В результате выполнения метода *exec( )* получаем массив, содержащий не один, а три элемента.

- Первый элемент, занимающий позицию 0, и есть та самая подстрока, которая совпала с регулярным выражением; в данном случае это «7abc».
- Второй элемент, занимающий позицию 1, – это строка, соответствующая первой группе в скобках `\d`; в данном случае это «7», потому что группа запрашивает цифры.
- Третий элемент, занимающий позицию 2, – это пустая строка, потому что соответствий второй группе `\d*`, что означает «цифра, повторяющаяся любое количество раз, включая 0», не выявлено.

Как отмечалось ранее, *exec( )* всегда начинает поиск соответствий с начала целевой строки, с позиции 0. Однако если в регулярном выражении установлен флаг `global`, то начало поиска определяется свойством `lastIndex`. При каждом вызове метода *exec( )* регулярного выражения с флагом `global` свойству `lastIndex` автоматически присваивается значение следующего индекса относительно индекса последнего символа соответствующей шаблону подстроки. Чтобы выбрать в строке все соответствия шаблону, *exec( )* регулярного выражения с флагом `global` необходимо вызвать несколько раз, по разу для каждого соответствия. Кроме того, можно задать значение свойства `lastIndex`, тогда поиск соответствий будет осуществляться, начиная с заданного элемента строки.

Если в регулярном выражении установлен флаг `global`, то при каждом вызове метода *exec( )* просмотр строки продолжается с индекса, следующего за последним выявленным соответствием. Если метод возвращает `null`, следующий поиск начинается с начала строки.

```
// Создаем регулярное выражение с флагом global,
// чтобы вести поиск нескольких соответствий в строке
```

```
var example:RegExp = /abc/g;

// Ищем первое соответствие
var result:Array = example.exec( "abc abc" );

// На экран выводится 0
trace( result.index );
// На экран выводится 3
trace( example.lastIndex );

// Ищем второе соответствие
result = example.exec( "abc abc" );

// На экран выводится 4
trace( result.index );
// На экран выводится 7
trace( example.lastIndex );

// Снова ищем соответствие, но ничего не находим, поэтому возвращается null
result = example.exec( "abc abc" );

// На экран выводится: null
trace( result );
// На экран выводится: 0
trace( example.lastIndex );
```

В предыдущем примере при первом вызове *exec( )* соответствие обнаруживается под индексом 0, и свойству `lastIndex` автоматически присваивается значение 3. Когда *exec( )* вызывается во второй раз, регулярное выражение начинает поиск с позиции 3 и находит второе соответствие под индексом 4 (автоматически переводя `lastIndex` в позицию 7). При последнем вызове *exec( )* соответствия не выявлены, возвращается `null`. В этом случае `lastIndex` присваивается значение 0, таким образом, при следующем вызове *exec( )* поиск начинается с начала строки.

Для выявления всех соответствий метод *exec( )* можно вызывать в выражении *while*:

```
// Создаем регулярное выражение для выбора трехбуквенных слов
var example:RegExp = /\b[a-z]{3}\b/g;
var target:String = "В этой строке есть два слова по три буквы";

/* Выполняем цикл до тех пор, пока метод exec() не возвратит null.
   В результате выполнения этого цикла while на экран выводится:
   два
   три
*/
var result:Array;
while ( ( result = example.exec( target ) ) != null ) {
  trace( result );
}
```

См. также

Рецепты 13.1 и 13.3.

13.3. Поиск соответствий шаблону

Задача

Требуется найти в строке соответствия шаблону.

Решение

Задача решается посредством методов *String.search( )* и *String.match( )*.

Обсуждение

Поведение метода *String.search( )* аналогично поведению метода *RegExp.test( )*, который обсуждается в рецепте 13.2. Метод *search( )* обрабатывает строковый объект и принимает регулярное выражение в качестве параметра. Если в строке выявлено соответствие регулярному выражению, метод возвращает начальный индекс соответствующей подстроки. Если соответствия не выявлены, возвращается –1:

```
var example:String = "ActionScript 3 Cookbook";

// На экран выводится: 6
trace( example.search( /script/i ) );

// На экран выводится: -1
trace( example.search( /script/ ) );
```

В предыдущем блоке кода в первый раз метод *search( )* проводит нечувствительный к регистру поиск подстроки `script`. Соответствующая подстрока, `Script`, обнаруживается в позиции 6. Во второй раз в *search( )* используется то же регулярное выражение, но на этот раз без флага `ignoreCase`. Поскольку в строке *example* нет подстроки `script` (теперь регистр учитывается), метод возвращает –1.

Метод *search( )* не учитывает флаг `global` или значение свойства `lastIndex` передаваемого в него регулярного выражения. Он всегда начинает поиск с начала строки.

```
// Создаем регулярное выражение с флагом global для выбора трехбуквенных слов
var regex:RegExp = /\b[a-z]{3}\b/g;
var sentence:String = "В этой строке есть два слова по три буквы.";

/* На экран выводится:
  19
  0
  19
*/
trace( sentence.search( regex ) );
trace( regex.lastIndex );
trace( sentence.search( regex ) );
```

Метод *search( )* позволяет выявить только первое соответствие регулярному выражению, даже если установлен флаг `global`. Также не про-

исходит автоматического обновления значения свойства `lastIndex`, как в *RegExp.exec( )* (см. рецепт 13.2). Кроме того, даже если свойство `lastIndex` задается вручную, метод *search( )* его не учитывает:

```
// Задаем индекс, следующий за индексом
// выявленного соответствия, как последний индекс
regex.lastIndex = 20;

/* На экран выводится:
  19
  20
*/
trace( sentence.search( regex ) );
trace( regex.lastIndex );
```

Поскольку метод *search( )* не учитывает флаг `global` регулярных выражений, он может применяться только для проверки наличия в строке соответствий заданному регулярному выражению, но не определения количества этих соответствий (как и метод *RegExp.test( )*). Для извлечения всех подстрок, соответствующих шаблону регулярного выражения, предназначен метод *String.match( )*, поведение которого аналогично методу *RegExp.exec( )* (который также обсуждается в рецепте 13.2).

Метод *match( )* обрабатывает строковый объект и принимает регулярное выражение в качестве параметра. Он возвращает массив, который содержит подстроки, соответствующие шаблону. Если в регулярном выражении не установлен флаг `global`, то массив содержит всего один элемент – соответствующую подстроку, которая была выявлена первой. Если регулярное выражение задано с флагом `global`, метод начинает поиск соответствий с начала строки и возвращает массив, содержащий все выявленные соответствия. Если соответствия не обнаружены, *match( )* возвращает `null`.

```
// Создаем регулярное выражение с флагом global для выбора трехбуквенных слов
var regex:RegExp = /\b[a-z]{3}\b/g;
var sentence:String = "В этой строке есть два слова по три буквы.";

var matches:Array = sentence.match( regex );

/* На экран выводится:
  два,три
  2
*/
trace( matches );
trace( matches.length );
```

См. также

Рецепты 13.1 и 13.2.

13.4. Удаление и замена символов и слов с помощью шаблонов

Задача

Требуется удалить из строки символы, соответствующие регулярному выражению, или заменить их другой подстрокой.

Решение

Это делается при помощи метода *String.replace( )*.

Обсуждение

Метод *String.replace( )* (см. рецепт 12.7) предоставляет возможность использовать строку как шаблон для замены символов и слов. Но регулярные выражения в данном случае более эффективны.

Вспомним из рецепта 12.7, что метод *replace( )* принимает два параметра:

pattern

Подстрока или регулярное выражение, которое требуется найти и заменить.

replace

Значение, которое подставляется на место выявленных соответствий *шаблону* (*pattern*). Это значение обычно является строкой, но также может быть функцией, возвращающей строку; это особенно полезно, когда *pattern* представляет собой регулярное выражение.

Работаем с методом *replace( )* так же, как и раньше, только в качестве шаблона вместо строки выступает регулярное выражение. Помним, что *replace( )* не меняет исходной строки, поэтому необходимо убедиться, что работа ведется со строкой, возвращаемой *replace( )*:

```
var example:String = "<p>Строка с <b>HTML</b> в ней</p>";

// Заменяем HTML с помощью нежадного регулярного
// выражения с флагом global, используя пустую строку
// в качестве подстановки вместо соответствий шаблону.
example = example.replace( /<.*?>/g, "" );

// На экран выводится: Строка с HTML в ней
trace( example );
```

Этот код демонстрирует, как удалить из строки теги HTML. В данном случае следует применить нежадное регулярное выражение (см. рецепт 13.5). Если пустая строка выступает в качестве строки подстановки, то из исходной строки удаляются все соответствия шаблону.

Флаг `global` регулярного выражения определяет поведение метода *replace( )*. Если флаг установлен, заменяются все соответствия шаблону. Если флаг не установлен, заменяется только первое соответствие.

Применение регулярного выражения в качестве параметра `pattern` расширяет и возможности параметра `replace`. Если значение параметра `replace` представляет собой строку, то могут быть задействованы специальные *коды подстановки*, обеспечивающие возможность использовать в качестве подстановки части шаблона. В табл. 13.4 перечислены коды, которые могут присутствовать в строке подстановки, а также то, чем заменяется данный код.

Таблица 13.4. Коды подстановки

Код	Заменяется
$$	Символом $.
$&	Найденной подстрокой.
$`	Текстом в строке, предшествующим найденной подстроке. Обратите внимание, символ после $ – это гравис, или тупое ударение, на клавиатуре обычно располагается сразу слева от клавиши 1 и над клавишей табуляции.*
$'	Текстом в строке, который следует за сопоставимой подстрокой.
$n	*n*-ная найденная группа, где *n* – число от 1 до 9.
$nn	*nn-ная* найденная группа, где *nn* – двузначное число от 10 до 99.

* На кириллической клавиатуре этой кнопке соответствует символ «ё». – *Примеч. науч. ред.*

Коды подстановки – это мощное средство замены текста. Скажем, имеется простая строка текста и требуется заменить в ней все ссылки, начинающиеся с *http://*, их HTML-эквивалентом. Для этого их надо окружить «якорями» (тегами `<a>`) и корректно задать атрибут *href*. Эту задачу практически невозможно решить, если строка выступает в качестве значения `pattern`. Однако данный результат может быть получен с помощью регулярного выражения в `pattern` и кодов подстановки, которые принимает `replace`:

```
var example:String = "Посетите меня на http://www.darronschall.com, "
                   + "или Adobe на http://www.adobe.com.";

// Заменяем ссылки, начинающиеся с http:// и заканчивающиеся .com, .net
// или .org, соответствующей подстрокой, заключенной в тег <a>, используя
// соответствие и как атрибут href, и как текст ссылки
example = example.replace( /http:\/\/.*?\.(com|net|org)/g, "<a href=\"$&\"
>$&</a>" );

/* На экран выводится:
Посетите меня на <a href="http://www.darronschall.com">http://
www.darronschall.com</a>, или Adobe на <a href="http://www.adobe.com">http:/
/www.adobe.com</a>.
*/
trace( example );
```

Коды подстановки – ценная штука, позволяющая решать многие творческие задачи, но больше возможностей для замены текста открывает использование функции подстановки в качестве параметра `replace`. Функция подстановки вызывается при каждом обнаружении соответствия шаблону. Строковое значение, возвращаемое ею, выступает в качестве значения подстановки. В функцию подстановки передаются различные параметры, которые могут служить для формирования соответствующей строки подстановки. Передаваемые в функцию параметры (по порядку):

- Подстрока, соответствующая шаблону.
- Найденные группы, по одному параметру для каждой такой группы в шаблоне; для определения количества передаваемых в функцию подстрок найденных групп используем `arguments.length – 3`.
- Начальный индекс соответствующей подстроки в строке.
- Строка, чей метод *replace( )* был вызван.

Поскольку число групп в регулярном выражении может меняться, строго типизированной сигнатуры этого метода нет. Функция подстановки должна быть определена как безаргументная и использовать массив `arguments` для получения необходимой информации:

```
var example:String = "Посетите меня на http://www.darronschall.com, "
                   + "или Adobe на http://www.adobe.com.";

// Заменяем все ссылки соответственно функции
// replaceLinks, которая определяет содержимое тега <a>
example = example.replace( /http:\/\/.*?\.(com|net|org)/g, replaceLinks );

/* На экран выводится:
Посетите меня на <a href="http://www.darronschall.com">мой веб-сайт</a>,
или Adobe на <a href="http://www.adobe.com">их веб-сайт</a>.
*/
trace( example );
```

Теперь *replaceLinks( )* можно определить так:

```
public function replaceLinks():String {
  var linkText:String;
  // При любой замене ссылки darronschall.com используем "мой веб-сайт"
  // как текстовое значение тега <a>,
  // в противном случае, используется "их веб-сайт"
  if ( arguments[0] == "http://www.darronschall.com" ) {
    linkText = "мой веб-сайт";
  } else {
    linkText = "их веб-сайт";
  }
  // Создаем ссылку HTML из сопоставимого текста,
  // используя текст соответствующей ссылки
  return "<a href=\"" + arguments[0] + "\">" + linkText + "</a>";
}
```

За счет использования функции подстановки предыдущий код меняет текст тега <a> в зависимости от значения найденной подстроки. Такая функциональность выходит далеко за рамки того, что могут коды подстановки.

См. также

Рецепты 12.7 и 13.5.

13.5. Создание нежадного шаблона

Задача

Использование регулярного выражения не дает желаемых результатов. Шаблон ведет себя жадно, выбирая больше символов, чем требуется.

Решение

Замените регулярное выражение его нежадной версией, чтобы выбиралось минимально возможное количество символов.

Обсуждение

Любой шаблон, включающий метасимволы * и + или метапоследовательность {*n*, *m*}, описывающие повторения символов, является *жадным* (*greedy*). Жадный шаблон – это шаблон, который пытается охватить столько текста, сколько возможно, выбирая наибольшую из возможных подстроку. Такое поведение шаблона объясняется базовым кодом механизма регулярных выражений. Понимание принципов работы этого механизма поможет создавать более точные регулярные выражения.

Предположим, требуется удалить из строки теги HTML, как описывается в рецепте 13.4. Любой HTML-тег начинается с открывающей угловой скобки < и заканчивается закрывающей угловой скобкой >. Между ними могут располагаться различные символы, такие как числа, буквы, кавычки, знаки равенства и т. д. Для простоты, чтобы не создавать отдельные классы для всех символов, которые могут находиться в угловых скобках, используется последовательность .*, соответствующая «любому символу, повторяющемуся любое число раз». Таким образом, получаем регулярное выражение /<.*>/g и применяем его к строке, содержащей теги HTML:

```
var example:String = "<b>здравствуй</b>, мир!";

// На экран выводится: <b>здравствуй</b>
trace( example.match( /<.*>/g ) );
```

Требовалось, чтобы регулярное выражение обеспечило выбор двух соответствий, и . Из этого фрагмента кода видно, что в результате

получено только одно соответствие, а именно вся строка `<b>здравствуй</b>`. Это пример жадного шаблона в действии.

Первый символ выражения, открывающая угловая скобка, принимается буквально. Он обнаруживается в самом начале строки, и механизм сравнения перемещается по шаблону далее. Следующий символ шаблона – точка, которая соответствует всему, кроме символа перевода строки. За ним следует звездочка (`*`), обеспечивающая повторение точки ноль или более раз. Вот в этом и кроется причина жадности шаблона.

Точка (`.`) соответствует букве `b`, потому что она подходит под определение «любой символ». После `b` в строке идет закрывающая угловая скобка, которая тоже является «любым символом» и, как вы уже догадались, выбирается, т. к. дает совпадение с точкой. Совпадение с точкой продолжается, и вся строка, `hello</b>, world!`, выбирается до конца. Когда строка заканчивается, заканчиваются и совпадения с точкой. Однако шаблон должен найти закрывающую угловую скобку. Механизм «знает», что повторение `*` осуществлялось из-за наличия точки, поэтому он *меняет курс* и начинает поиск совпадения с закрывающей угловой скобкой, чтобы завершить шаблон. Двигаемся по строке в обратном направлении, от конца. Восклицательный знак (`!`) не соответствует закрывающей угловой скобке, поэтому механизм, помня о том, что до этого был реализован шаблон `.*`, продолжает сравнивать символы в обратной последовательности. Так он перебирает символы `р`, `и`, `м` и т. д., пока не встречает `>` прямо перед запятой. Эта угловая скобка вполне соответствует закрывающей угловой скобке в шаблоне. Механизм считает, что шаблон успешно сопоставлен, и возвращает `<b>здравствуй</b>`.

Механизм *RegExp* стремится максимально быстро завершить сопоставление с шаблоном. Поскольку соответствие найдено, нет необходимости продолжать поиск и пытаться найти более точное соответствие. Поэтому жадные шаблоны всегда возвращают самый длинный вариант соответствия из возможных.

Из этой ситуации есть два выхода. Первый – сделать шаблон нежадным, такие шаблоны еще иногда называют *ленивыми шаблонами* (*lazy patterns*). Второй – творчески подойти к работе с классом символов.

Чтобы сделать шаблон ленивым, надо «умерить аппетиты» повторения. Ленивые шаблоны выбирают минимально возможную подстроку. Создать ленивый шаблон несложно – надо добавить метасимвол `?` в конец выражения повторения, как описано в табл. 13.5.

Таблица 13.5. Ленивые шаблоны

Выражение	«Лениво» сопоставляет предыдущий символ или группу
`??`	Ноль или один раз
`*?`	Ноль или более раз
`+?`	Один или более раз

Выражение	«Лениво» сопоставляет предыдущий символ или группу
{n,}?	По крайней мере *n* раз
{n,m}?	По крайней мере *n*, но не более чем *m* раз

Жадный шаблон /<.*>/g, служивший для сопоставления тегов HTML, можно заменить ленивым шаблоном /<.*?>/g, что обеспечит желаемый эффект, т. е. выбор всех тегов HTML:

```
var example:String = "<b>здравствуй</b>, мир!";

// На экран выводится: <b>,</b>
trace( example.match( /<.*?>/g ) );
```

Базовый механизм *RegExp* обрабатывает ленивый шаблон иначе, чем жадный. Открывающая угловая скобка выбирается точно так же, как и раньше, но на этот раз за точкой следует «ленивый» символ * (на что указывает ?), и механизм знает, что должен выбрать как можно меньше символов. В данной точке требование «любой символ ноль или более раз» уже выполнено, поскольку ноль символов уже выявлен, и механизм начинает поиск закрывающей угловой скобки. В строке попадается символ b, который не соответствует >. Механизм, как и в первом случае, возвращается назад, но эта операция заставляет ленивый * выбирать еще один символ согласно .*. В результате выбирается b, и механизм опять возвращается к поиску >. Следующий символ в строке – закрывающая угловая скобка, таким образом, соответствие найдено и шаблон успешно завершается. Механизм возвращает <b>, именно этот результат нам и требовался.

Можно не создавать ленивый шаблон, а заменить точку (.) классом символов, который включает все символы, кроме закрывающей угловой скобки. В результате получается шаблон /<[^>]*>/g:

```
var example:String = "<b>здравствуй</b>, мир!";

// На экран выводится: <b>,</b>
trace( example.match( /<[^>]*>/g ) );
```

В данном случае, вероятно, класс символов – это лучший вариант, потому что таким образом устраняется обратный перебор строки. Механизм обрабатывает открывающую угловую скобку как обычно, и затем сопоставляет символы до закрывающей угловой скобки, поскольку она не входит в заданный класс символов. Тогда механизм переходит к символу > в шаблоне и сообщает об успешном завершении сопоставления, поскольку нашел в строке символ >.

См. также

Рецепты 13.1, 13.3 и 13.4.

13.6. Проверка данных, вводимых пользователем

Задача

Требуется гарантировать правильность ввода информации пользователем, например адреса электронной почты, номера социального страхования, номера телефона или почтового индекса.

Решение

Используйте общепринятые шаблоны, приведенные в данном рецепте.

Обсуждение

Регулярные выражения исключительно полезны для проверки правильности разнообразных данных, вводимых пользователями. Например, имеется форма, в которую пользователь для подписки на сетевую игру должен вводить адрес электронной почты, и необходимо обеспечить, чтобы был введен действительный адрес. Или требуется гарантировать правильность ввода даты рождения, или проверить действительность номера кредитной карты. Во всех этих ситуациях помогут приведенные ниже регулярные выражения:

- Соответствует дате в формате ##/##/####, где день и месяц могут быть заданы одной или двумя цифрами, а год – двумя или четырьмя и начинаться с 19 или 20:

```
^\d{1,2}\/\d{1,2}\/(\d{2}|(19|20)\d{2})$
```

- Соответствует номеру социальной страховки в формате ###-##-####, где тире могут отсутствовать, а три группы чисел могут быть разделены пробелами:

```
^\d{3}\s*-?\s*\d{2}\s*-?\s*\d{4}$
```

- Соответствует пятизначному почтовому индексу США с необязательным тире и четырехзначным расширением:

```
^\d{5}(-\d{4})?$
```

- Соответствует канадскому почтовому индексу в формате L#L #L# (где L – буква). На первую букву индекса накладывается ограничение, гарантирующее действительность заданной провинции, территории или области:

```
^[ABCEGHJKLMNPRSTVXY]\d[A-Z] \d[A-Z]\d$
```

- Соответствует формату телефонных номеров США (###) ###-####, где код является необязательным, скобки вокруг кода необязательные и могут быть заменены тире и группы цифр могут быть (не обязательно) разделены пробелами:

```
^(\(\s*\d{3}\s*\)|(\d{3}\s*-?))?\s*\d{3}\s*-?\s*\d{4}$
```

- Соответствует формату телефонных номеров США, от предыдущего выражения отличается только тем, что допускает необязательное расширение от одного до пяти знаков, заданное как «x», «ext» или «ext.» и необязательный пробел:

```
^(\(\s*\d{3}\s*\)|(\d{3}\s*-?))?\s*\d{3}\s*-
?\s*\d{4}\s*((x|ext|ext\.)\s*\d{1,5})?$
```

- Соответствует номеру кредитной карты, состоящему из четырех групп цифр, группы могут быть разделены тире или пробелами:

```
^(\d{4}\s*\-?\s*){3}\d{4}$
```

- Соответствует записи денежной суммы в долларах США, начинается со знака $, за которым следует любое число с, максимум, двумя необязательными десятичными знаками:

```
^\$\d(\.\d{1,2})?$
```

- Соответствует адресу электронной почты, где домен не является IP-адресом и может содержать любое число необязательных субдоменов. При создании данного регулярного выражения нелишним будет установить флаг `ignoreCare`:

```
^[a-z0-9][-._a-z0-9]*@([a-z0-9][-_a-z0-9]*\.)+[a-z]{2,6}$
```

- Соответствует IP-адресу, когда интересует только правильность форматирования адреса (что он должен включать четыре группы от одного до трех разрядов, разделенные точками):

```
^(\d{1,3}\.){3}\d{1,3}$
```

Предыдущее регулярное выражение соответствует 999.999.999.999, что формально не является действительным IP-адресом, но имеет соответствующее форматирование.

- Соответствует IP-адресу, когда важна правильность форматирования адреса и то, что каждая группа цифр представляет число в диапазоне от 0 до 255:

```
^((25[0-5]|2[0-4]\d|[01]?\d\d?)\.){3}(25[0-5]|2[0-4]\d|[01]?\d\d?)$
```

- Соответствует адресу электронной почты, где домен является или именем домена, состоящим из любого числа необязательных субдоменов, или IP-адресом:

```
^[a-z0-9][-._a-z0-9]*@(([a-z0-9][-_a-z0-9]*\.)+[a-z]{2,6}|((25[0-5]|2[0-
4]\d|[01]?\d\d?)\.){3}(25[0-5]|2[0-4]\ d|[01]?\d\d?))$
```

Чтобы использовать одно из приведенных выше регулярных выражений в контексте проверки допустимости вводимых пользователем данных, обратимся к методу *RegExp.test( )*, описанному в рецепте 13.2:

```
// Создаем регулярное выражение для проверки допустимости почтового индекса
var zipCode:RegExp = /^\d{5}(-\d{4})?$/;
```

```
// Проверяем, является ли строка, введенная пользователем,
// действительным почтовым индексом
if ( !zipCode.test( "12384-1231" ) ) {
  // Почтовый индекс недействительный,
  // сообщаем пользователю об ошибке
} else {
  // Почтовый индекс действительный, вероятно,
  // здесь больше ничего не надо делать
}
```

При создании регулярных выражений обычно приходится находить компромисс между точностью и сложностью. Рассмотрим два регулярных выражения для выбора IP-адреса из приведенного выше списка. Первое регулярное выражение довольно просто для понимания: оно три раза выбирает группу из одной-трех цифр, за которыми следует точка, и затем выбирает еще одну группу из одной-трех цифр. Это регулярное выражение простое, но оно не совсем точное. Каждая группа чисел в IP-адресе ограничена диапазоном от 0 до 255, тогда как данное регулярное выражение допускает любое число от 0 до 999. Поэтому первое выражение годится только для определения правильности форматирования IP-адресов, но не подходит для проверки их действительности.

Второе регулярное выражение для IP-адреса намного сложнее, но оно точнее. Оно ограничивает группы чисел значениями от 0 до 255 и также гарантирует наличие четырех групп чисел, разделенных точками. Какое регулярное выражение будет использоваться, зависит от ситуации.

Сложные регулярные выражения нелегко понимать и создавать, но они обеспечивают наибольшую точность. Простые регулярные выражения могут давать больше совпадений, чем требуется (ошибочные совпадения), или находить не все, что требуется (ошибочные несовпадения).

Другой пример компромисса между сложностью и точностью – регулярное выражение для выбора даты. Регулярное выражение в формате `##/##/####` допускает ошибочное разрешение. Например, 01/99/2006 принимается шаблоном, но в январе не 99 дней.

К сожалению, создать регулярное выражение для даты в определенном формате, которое обеспечивало бы выбор только допустимых сочетаний месяца, дня и года, практически невозможно из-за большого количества условий. В таких случаях группировка с помощью регулярных выражений используется в сочетании с кодом ActionScript. Это позволяет находить группы, соответствующие регулярному выражению, и применять некоторую логику, гарантирующую, что значения группы имеют смысл:

```
// Проверяем дату исходя из строки, введенной пользователем
var inputDate:Date = extractDate( theInputString );

// Проверяем действительность даты
```

```
if ( inputDate == null ) {
  // Не можем разобрать дату правильно, недействительна
} else {
  // Действительная дата
}
```

Ниже приведено описание функции *extractDate( )*:

```
// Пытаемся извлечь из строки значение даты в формате
// ##/##/#### или ##/##/##. В случае удачной операции
// возвращаем объект Date, описанный строковым значением,
// или null, в противном случае.
public function extractDate( possibleDate:String ):Date {
  var datePattern:RegExp = /^(\d{1,2})\/(\d{1,2})\/(\d{2}|(19|20)\d{2})$/;

  // Используем регулярное выражения для отсеивания дат,
  // заданных в несоответствующем формате
  var result:Array = datePattern.exec( possibleDate );

  // Результат null означает, что формат недействительный
  if ( result == null ) {
    return null;
  }

  // Здесь выбрана дата в правильном формате и результирующий
  // массив содержит соответствующую подстроку, а также все
  // найденные группы. Если possibleDate - "02/08/2006",
  // результирующий массив содержит: 02/08/2006,02,08,2006,20

  // Преобразовываем строковые значения в значения типа int
  var month:int = parseInt( result[1] );
  var day:int = parseInt( result[2] );
  var year:int = parseInt( result[3] );

  // Применяем дополнительную логику, чтобы убедиться,
  // что месяц, день и год имеют смысл
  if ( month > 12 || day > 31 || month == 0 || day == 0 ) {
    // Значение месяца или дня слишком большое
    // или слишком маленькое - недействительное
    return null;
  } else if ( day == 31 && ( month == 9 || month == 4 || month == 6
                          || month == 11 || month == 2 ) ) {
    // 31 день для сентября, апреля, июня, ноября
    // или февраля является недействительным значением
    return null;
  } else if ( day == 30 && month == 2 ) {
    // 30 дней для февраля недействительное значение
    return null;
  } else if ( day == 29 && month == 2
            && !( year % 4 == 0 && ( year % 100 != 0 || year % 400 == 0 ) )
) {
    // 29 дней в феврале, но год невисокосный - неправильно
    return null;
  } else {
```

```
        // Обработать года, когда они заданы двумя последними
        // числами, сложно. Год 99 должен трактоваться как 1999,
        // а 06 - как 2006. В результате использования 06
        // в конструкторе Date получим 1906 год, поэтому произвольно
        // выбираем год, допустим, 15, и все, что меньше этого,
        // будет преобразовываться в 20xx.
        // Все, что больше этого, по умолчанию будет 19xx.
        if ( year <= 15 ) {
          year += 2000;
        }
        // Согласно логике месяц, день и год имеют смысл,
        // поэтому возвращаем соответствующий объект Date.
        // Вычитаем из месяца 1, потому что в ActionScript
        // отсчет месяцев ведется от 0.
        return new Date( year, month - 1, day );
      }
    }
```

См. также

Рецепты 13.1 и 13.2.

14

Даты и время

14.0. Введение

Даты и время играют важную роль во многих приложениях на ActionScript, особенно в тех, которые предлагают различные услуги пользователям. Например, значения даты и времени важны в нормированных по времени операциях, для определения действительности пробного членства или для хранения дат осуществления транзакций.

ActionScript сохраняет даты и время в *миллисекундах эпохи* (*epoch milliseconds*) – количество миллисекунд, прошедшее от начала отсчета времени, т. е. с 1 января 1970 года по всемирному координированному времени (Coordinated Universal Time, UTC). Для наших целей UTC можно считать эквивалентным более привычному времени по Гринвичу (Greenwich Mean Time, GMT). Чем они отличаются, можно найти на сайте Обсерватории военно-морских сил США (*http://aa.usno.navy.mil/faq/docs/UT.html*). Во многих языках программирования даты отсчитываются от начала эпохи (часто измеряются в секундах, а не миллисекундах), поэтому в ActionScript можно без труда работать со значениями дат и времени, импортированными из других источников (и наоборот).

Кроме того, класс *Date* посредством таких свойств, как `fullYear`, `month` и др., позволяет задавать и получать значения даты и времени в привычном для нас формате, отображающем год, месяц, день и время. Эти свойства существуют только для удобства пользователя и разработчика, все значения все равно хранятся в миллисекундах эпохи.

14.1. Как получить текущие дату и время

Задача

Требуется получить текущие дату и время.

Решение

Создайте новую дату с помощью конструктора *Date( )* без параметров или сценария CGI[1] или любой другой серверной программы для получения серверного времени и создайте новый объект *Date* на основании этого значения.

Обсуждение

Дату и время ActionScript вычисляет самостоятельно на основании настроек даты и времени клиентского компьютера. Следовательно, если на компьютере пользователя время установлено неправильно, объект *Date* покажет неправильное время. Текущее время и дату пользовательского компьютера можно получить, создавая новый объект посредством конструктора без параметров:

```
// Создаем новый объект Date.
var current:Date = new Date();

// Выводим на экран дату и время клиента.
trace(current);
```

Если установлено соединение с Интернетом, ролик Flash пытается получить дату и время с сервера. Так можно получить более точные значения даты и времени. Конечно, время на сервере тоже может быть неточным, но во всяком случае оно будет одинаковым для всех клиентов.

Основные составляющие процесса получения показаний времени с сервера следующие:

1. На веб-сервере создается сценарий CGI, который возвращает количество секунд, прошедшее с полуночи 1 января 1970 года (от начала эпохи).
2. Показания времени загружаются с помощью объекта *flash.net.URLLoader* ActionScript.
3. Загруженное значение в секундах преобразуется из строки в число, умножается на 1000 и результат передается в конструктор для создания нового объекта *Date*.

PHP – язык сценариев, который используется на многих веб-серверах. Создать PHP-страницу для получения текущих даты и времени как количество секунд, прошедших с начала эпохи, довольно просто. Для этого достаточно создать PHP-документ со следующим содержимым и загрузить его на сервер:

```
<?php echo time();?>
```

Если на сервере не установлен PHP или если вам удобнее использовать Perl (еще один язык, который доступен практически на всех веб-серве-

[1] CGI (Common Gateway Interface) – общий шлюзовой интерфейс, стандартный интерфейс обмена данными.

рах), то вот сценарий на Perl, который делает то же самое – возвращает число секунд с начала эпохи:

```
#!/usr/local/bin/perl
print "Content-type:text/plain\n\n";
print time;
```

Вот несколько подсказок, которые необходимо учесть при установке этого сценария на сервере:

- Первая строка указывает на то, где можно найти интерпретатор Perl. Приведенное в примере значение достаточно универсальное. Однако если возникают какие-то проблемы, необходимо связаться с администратором веб-сервера.
- На многих серверах отключена возможность удаленного выполнения сценариев без определенных расширений файлов, но расширение *.cgi* обычно разрешено. Попытайтесь изменить имя сценария на *getDate.cgi*.
- Большинство веб-серверов ограничивают доступ CGI определенными каталогами. Обычно эти каталоги находятся в корневом каталоге учетной записи (или в корневом каталоге веб-узла учетной записи) и называются *cgi* или *cgi-bin*. Убедитесь, что сценарий располагается в соответствующем каталоге.
- На серверах UNIX сценарий CGI должен иметь права доступа 755. Большинство программ FTP позволяют менять права доступа. При работе из сценария оболочки (shell script) применяется следующая команда:

```
chmod 755 имяфайла
```

Независимо от того, какой язык используется на сервере, загружается значение времени с помощью ActionScript. Это выполняется посредством объекта *flash.net.URLLoader*.

Если в конструктор *Date* передается одно значение, ActionScript интерпретирует его как количество миллисекунд, прошедшее от начала эпохи, и создает новый объект *Date*, соответствующий этому значению. Поэтому возвращенное сценарием значение (в секундах) надо умножить на 1000, например:

```
package {

  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import flash.events.Event;

  public class ServerDateTimeExample {

    public function ServerDateTimeExample() {
      // Следующий код создает объект URLLoader, добавляет слушателя
      // событий, чтобы при загрузке данных вызывался метод
      // onDateTimeLoad(), и делает запрос к сценарию сервера.
```

```
            var loader:URLLoader = new URLLoader();
            loader.addEventListener(Event.COMPLETE, onDateTimeLoad);
            loader.load(new URLRequest("script.cgi"));
        }

        private function onDateTimeLoad(event:Event):void {
            var loader:URLLoader = URLLoader(event.target);
            var data:int = parseInt(loader.data);
            var current:Date = new Date(data * 1000);
            trace(current);
        }

    }
```

ActionScript всегда сохраняет даты в миллисекундах, прошедших с начала эпохи, и всегда отображает их соответственно локальным установкам часового пояса (если только пользователь специально не обращается к методам UTC). Поэтому если на клиентской машине часовой пояс задан неправильно, время может отображаться неправильно, но фактическая дата (поскольку она хранится в миллисекундах эпохи) все равно будет правильной.

См. также

Рецепт 14.9; если значения времени, предоставляемые вашим сервером, не заслуживают доверия или их точность не отвечает вашим требованиям, в Интернете есть множество серверов даты и времени, с которых можно получить точную информацию о дате и времени, например *http://tycho.usno.navy.mil*. Синхронизация времени через сетевой протокол синхронизации времени (Network Time Protocol, NTP) подробно описана по адресу *http://www.ntp. org*.

14.2. Как получить значения даты

Задача

Требуется извлечь из объекта *Date* значения года, месяца, дня месяца, дня недели, часов, минут, секунд или миллисекунд.

Решение

Для этого нужны свойства *fullYear*, *date*, *month*, *day*, *hours*, *minutes*, *seconds* или *milliseconds*.

Обсуждение

Свойства *fullYear*, *date*, *month*, *day*, *hours*, *minutes*, *seconds* и *milliseconds* возвращают значения объекта *Date*:

- Свойство *fullYear* (полный год) определяет год как четырехзначное значение, например 2010.

- Свойство *date* (дата) определяет день месяца как значение в диапазоне от 1 до 31.
- Свойство *month* (месяц) определяет месяц как целое от 0 (январь) до 11 (декабрь).
- Свойство *day* (день) определяет день недели как целое от 0 (воскресение) до 6 (суббота).
- Свойство *hours* (часы) возвращает количество часов как целое от 0 (полночь) до 23 (11 часов вечера).
- Свойства *minutes* (минуты) и *seconds* (секунды) возвращают значения от 0 до 59.
- Свойство *milliseconds* (миллисекунды) возвращает значение от 0 до 999.

Для каждого из этих свойств существуют UTC-соответствия (по всемирному координированному времени или времени по Гринвичу). Например, хотя свойство *hours* возвращает текущие часы на основании показаний часов клиентского компьютера, свойство *hoursUTC* возвращает текущее значение часов по UTC-времени.

14.3. Как получить название дня или месяца

Задача

Требуется получить название дня или месяца.

Решение

Создайте массивы, содержащие строковые значения названий дней недели и месяцев года. Для извлечения строковых значений из массивов используйте числовое значение дня и месяца.

Обсуждение

Класс *Date* предоставляет свойства *day* и *month*, которые возвращают целые значения дня недели (от 0 до 6) и месяца (от 0 до 11). Однако нам привычнее представлять день или месяц их названием, а не порядковым номером, отсчет которого ведется от нуля. Для выполнения этой задачи создаются массивы, элементами которых являются названия дней и месяцев или константы специального класса *ascb.util.DateFormat*. Все они – `DAYS`, `DAYS_ABBREVIATED`, `MONTHS` и `MONTHS_ABBREVIATED` – представляют собой массивы строк, соответствующих числовым значениям, которые возвращают свойства *day* и *month*. Эти константы имеют следующие описания:

```
public static const DAYS:Array = ["Sunday", "Monday", "Tuesday", "Wednesday"
, "Thursday", "Friday", "Saturday"];
public static const DAYSABBREVIATED:Array = ["Sun", "Mon", "Tues", "Wed", "
Thurs", "Fri", "Sat"];
```

```
public static const MONTHS:Array = ["January", "February", "March", "April",
"May", "June", "July", "August", "September", "October", "November", "
December"];
public static const MONTHSABBREVIATED:Array = ["Jan", "Feb", "Mar", "Apr", "
May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];
```

Как используются эти константы, показано в следующем примере:

```
var example:Date = new Date(2010, 3, 10);

trace(DateFormat.DAYS[example.day]);     // На экран выводится: Saturday
trace(DateFormat.DAYSABBREVIATED[example.day]); // На экран выводится: Sat
trace(DateFormat.MONTHS[example.month]); // На экран выводится: April
trace(DateFormat.MONTHSABBREVIATED[example.month]); // На экран выводится: Apr
```

См. также

Рецепт 14.4.

14.4. Форматирование даты и времени

Задача

Требуется отображать значение даты и/или времени в определенном формате.

Решение

Обратитесь к методу *Date.toString( )* или специальному методу *DateFormat.format( )*, который возвращает дату и время в виде строки в определенном формате.

Обсуждение

Метод *Date.toString( )* возвращает удобную для пользователя строковую версию заданного объекта *Date*; например:

```
// На экран выводится (примерно такое): Tue Jan 5 14:25:20 GMT-0800 2010
trace((new Date()).toString());
```

Поскольку ActionScript автоматически вызывает метод *toString( )* для любого объекта, используемого в контексте строки, тот же результат можно получить и без *toString( )*, как показано в следующем примере:

```
// Тоже выводится: Tue Jan 5 14:25:20 GMT-0800 2010
trace(new Date());
```

Класс *Date* не обладает никакими другими встроенными функциональными возможностями для какого-либо специального форматирования даты или времени. Конечно, можно комбинировать строковые значения, как в следующем примере:

```
var example:Date = new Date(2010, 0, 5, 10, 25);
var formatted:String = (example.month + 1) + "/" + example.fullYear;
```

```
trace(formatted);  // На экран выводится: 1/2010
```

Однако в этом случае каждый раз, когда потребуется отображать дату и/или время, придется писать много кода. Вместо этого можно задействовать метод *format( )* экземпляра *ascb.util.DateFormat*. Класс *DateFormat* специально разработан для форматирования дат и времени по заданной маске. Маска может быть составлена из любых символов, но некоторые из них действуют как переменные. В табл. 14.1 показаны символы, которые могут выступать в качестве переменных при создании маски.

Таблица 14.1. Символы даты и времени

Символ	Значение	Пример
a	a.m. или p.m. в нижнем регистре	a.m.
A	A.M. или P.M. в верхнем регистре	P.M.
d	День месяца (с 0 впереди)	01
D	Сокращенное название дня недели	Sun
F	Месяц	January
g	12-часовой	1
G	24-часовой	1
h	12-часовой (с 0 впереди)	01
H	24-часовой (с 0 впереди)	01
i	Минуты (с 0 впереди)	01
j	День месяца	1
l	День недели	Sunday
m	Порядковый номер месяца (с 0 впереди)	01
M	Сокращенное название месяца	Jan
n	Порядковый номер месяца	1
s	Секунды (с 0 впереди)	01
t	Дней в месяце	31
w	Порядковый номер дня недели	0
y	Двухзначный год	06
Y	Четырехзначный год	2006

Маска передается при создании объекта *DateFormat* как параметр конструктора. Большинство символов, которые не используются как переменные в контексте маски, интерпретируются буквально. Например, следующий фрагмент кода создает объект *DateFormat*, который возвращает дату в стандартном для США формате, где месяц, день и год разделены слэшами:

```
var formatter:DateFormat = new DateFormat("m/d/Y");
```

Когда объект *DateFormat* создан, метод *format( )* может вызываться для представления любого экземпляра *Date* в строковом виде в соответствии с заданной маской:

```
var example:Date = new Date(2010, 0, 5, 10, 25);
var formatter:DateFormat = new DateFormat("m/d/Y");
trace(formatter.format(example));  // На экран выводится: 01/05/2010
```

Свойство `mask` позволяет задавать и возвращать маску. Это означает, что можно менять маску для существующего объекта *DateFormat*:

```
var example:Date = new Date(2010, 0, 5, 10, 25);
var formatter:DateFormat = new DateFormat("m/d/Y");
trace(formatter.format(example)); // На экран выводится: 01/05/2010
formatter.mask = "m/d/Y h:i a";
trace(formatter.format(example)); // На экран выводится: 01/05/2010 10:25 am
```

Если требуется трактовать какую-либо часть маски буквально, эта часть заключается в одинарные кавычки.

Это важно, если требуется отображать символы, которые иначе были бы интерпретированы в маске как переменные:

```
var example:Date = new Date(2010, 0, 5, 10, 25);
var formatter:DateFormat = new DateFormat("m/d/Y at h:i a");
trace(formatter.format(example));  // На экран выводится:
                                   // 01/05/2010 am31 10:25 am
formatter.mask = "m/d/Y 'at' h:i a";
trace(formatter.format(example));  // На экран выводится:
                                   // 01/05/2010 at 10:25 am
```

См. также

Рецепт 14.5 – в масках задействованы такие же символы-переменные, что и в PHP. Полезную справочную информацию можно найти по адресу *http://www.php.net/manual/en/function.date.php*.

14.5. Представление значений секунд или миллисекунд в виде минут и секунд

Задача

Требуется представить секунды или миллисекунды в виде минут и секунд (*mm*:*ss*).

Решение

Здесь нужны специальные методы *ascb.util.DateFormat.formatSeconds( )* или *ascb.util.DateFormat.formatMilliseconds( )*.

Обсуждение

Многие значения в ActionScript представляются в миллисекундах или секундах. Например, продолжительность звука дается в миллисекун-

дах. Однако в большинстве случаев пользователю удобны представления в виде минут и секунд. Задача решается с помощью небольшого по объему кода. Что упрощает задачу, этот код уже реализован как статические методы класса *DateFormat*.

Класс *ascb.utils.DateFormat* имеет два метода, преобразующие число в формат *mm*:*ss*. Для преобразования секунд в этот формат предназначен метод *formatSeconds( )*, а для преобразования миллисекунд – метод *formatMilliseconds( )*.

См. также

Рецепт 14.4.

14.6. Преобразование из формата DMYHMSM в миллисекунды эпохи и наоборот

Задача

Требуется реализовать преобразования из формата DMYHMSM (дни, месяцы, годы, часы, минуты, секунды, миллисекунды) в миллисекунды эпохи и наоборот.

Решение

Задача решается с помощью свойства *time*.

Обсуждение

Большинству из нас удобнее представлять даты и время в их компонентах, часах, днях и года, а не работать с миллисекундами или секундами эпохи. Например, намного проще представить время и дату как 10:25 утра, вторник, 5 января 2010, чем обсуждать соответствующее этой дате значение 1262715900000 в миллисекундах эпохи. Однако многие языки программирования, к которым относится и ActionScript, хранят время в миллисекундах эпохи (или секундах эпохи). Поэтому важно иметь возможность преобразовывать эти форматы друг в друга при отображении дат и времени пользователям или передаче данных приложениям, работающим с другими форматами.

ActionScript поддерживает формат представления даты DMYHMSM:

```
// Формируем дату 10:25 AM, вторник, 5 января 2010
var example:Date = new Date(2010, 0, 5, 10, 25);
```

ActionScript автоматически преобразует и сохраняет дату как соответствующее значение в миллисекундах эпохи. Чтобы извлечь это значение, необходимо просто обратиться к свойству *time* объекта *Date*:

```
// Для тихоокеанского региона на экран выводится: 1262715900000.
// Результат может меняться в зависимости от часового пояса пользователя.
```

```
trace(example.time);
```

Значение в секундах эпохи, возвращенное *time*, можно передавать в другое приложение (например, сценарий CGI) или использовать для осуществления математических операций над датами (см. рецепт 14.1).

Но может потребоваться задать дату и в миллисекундах эпохи. Например, в рецепте 14.1 сценарий CGI возвращает во Flash текущее время на сервере в секундах эпохи (которые надо преобразовать в миллисекунды, умножив на 1000). Также для вычислений удобнее задавать даты в миллисекундах эпохи. Есть два способа сделать это. Один – передать значение в миллисекундах в конструктор *Date* как единственный параметр, другой – присвоить значение в миллисекундах свойству *time* существующего экземпляра *Date*. В обоих случаях результат одинаковый.

```
// Создаем новый объект Date для 10:25 AM, вторник, 5 января 2010. Здесь
// используется значение, отображаемое в окне Output, из предыдущего примера.
var example:Date = new Date(1262715900000);

// На экран выводится: Tue Jan 5 10:25:00 GMT-0800 2010
// (смещения часового пояса могут меняться)
trace(example);
```

См. также

Рецепт 14.1.

14.7. Работа с таймерами

Задача

Требуется вызывать метод через определенный интервал времени или ввести задержку вызова.

Решение

Это делается при помощи класса *flash.utils.Timer*.

Обсуждение

Класс *flash.utils.Timer* обеспечивает возможность задавать таймируемые события или откладывать вызовы методов. Экземпляр *Timer* можно создать с помощью конструктора. В качестве параметра в конструктор передается количество миллисекунд, определяющее интервал между событиями таймера. В следующем примере создается экземпляр объекта *Timer*, который рассылает события каждые 1000 миллисекунд (1 секунду):

```
var timer:Timer = new Timer(1000);
```

После создания таймера необходимо добавить слушателя, который будет обрабатывать рассылаемые таймером события. Объекты *Timer*

формируют события *flash.event.TimerEvent* с интервалом, заданным при создании объекта *Timer* (или свойством задержки). В следующем примере кода определяется слушатель событий таймера (с использованием константы *TimerEvent.TIMER*), вызывающий метод *onTimer( )*:

```
timer.addEventListener(TimerEvent.TIMER, onTimer);
```

В метод-обработчик события передается объект *TimerEvent*:

```
function onTimer(event:TimerEvent):void {
  trace("по таймеру");
}
```

Объекты-таймеры не запускаются автоматически; для этого необходимо вызвать метод *start( )*:

```
timer.start();
```

По умолчанию таймер выполняется до тех пор, пока не будет остановлен методом *stop( )*. Однако, передав второй параметр в конструктор, можно задать число повторений таймера. Значение по умолчанию 0 обусловливает бесконечное повторение таймера, тогда как отличные от нуля значения определяют конечное число повторений. Следующий конструктор создает таймер, который сработает всего пять раз:

```
var timer:Timer = new Timer(1000, 5);
```

Определив таймер всего с одним повторением, можно создать задержку вызова метода. Следующий таймер откладывает вызов метода *deferredMethod( )* (отложенный метод) на пять секунд:

```
var timer:Timer = new Timer(5000, 1);
timer.addEventListener(TimerEvent.TIMER, deferredMethod);
timer.start();
```

14.8. Вычисление прошедшего времени или интервалов между датами

Задача

Требуется вычислить прошедшее время, дату окончания периода времени или относительное время.

Решение

Чтобы найти промежуток времени или дату его окончания, можно вычитать и добавлять значения времени в миллисекундах эпохи или использовать значение, возвращенное методом *getTimer( )*. Более сложные преобразования выполняются посредством методов специального класса *DateUtilities*.

Обсуждение

Простые операции, такие как добавление или вычитание часов, дней или недель, реализуются путем добавления или вычитания соответствующих значений в миллисекундах из даты, выраженной в миллисекундах эпохи. При этом секунда – это 1000 миллисекунд, минута – 60 000 миллисекунд, час – 3 600 000 миллисекунд, неделя – 604 800 000 миллисекунд и т. д. Эти переводные значения удобно сохранить как константы. Такие константы уже определены в классе *ascb.util.DateUtilities*:

```
public static const MILLISECOND:Number = 1;
public static const SECOND:Number = MILLISECOND * 1000;
public static const MINUTE:Number = SECOND * 60;
public static const HOUR:Number = MINUTE * 60;
public static const DAY:Number = HOUR * 24;
public static const WEEK:Number = DAY * 7;
```

Свойство *Date.time* позволяет получить текущее значение даты в миллисекундах эпохи и затем присвоить новое значение свойству *time* относительно текущего значения. В следующем примере к заданному объекту *Date* добавляется один день:

```
var example:Date = new Date(2010, 0, 5, 10, 25);

// На экран выводится: Tue Jan 5 10:25:00 GMT-0800 2010
trace(example);

// Задаем новое значение даты/времени, добавляя к исходному значению
// даты/времени DateUtilities.DAY (количество миллисекунд в одном дне).
example.time += DateUtilities.DAY;

// На экран выводится: Wed Jan 6 10:25:00 GMT-0800 2010
trace(example);
```

Часто вычисления прошедшего времени требуются при создании таймера для игры или другой деятельности. Подсчет прошедшего времени заключается в записи времени в момент запуска и последующем сравнении этого значения с текущим временем во время выполнения. Эти вычисления можно осуществлять с помощью объекта *Date*, но намного эффективнее использовать функцию *flash.utils.getTimer( )*. Она возвращает количество миллисекунд, прошедшее с момента запуска проигрывателя. В следующем примере функция *getTimer( )* служит для многократного обновления отображаемого текста:

```
package {

    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.utils.Timer;
    import flash.events.TimerEvent;
    import flash.utils.getTimer;
```

```
    public class Example extends Sprite {

        private var _text:TextField;
        private var _start:uint;

        public function Example() {
            _start = getTimer();
            _text = new TextField();
            addChild(_text);
            var timer:Timer = new Timer(1000);
            timer.addEventListener(TimerEvent.TIMER, onTimer);
            timer.start();
        }

        private function onTimer(event:TimerEvent):void {
            _text.text = (getTimer() - _start) + " миллисекунд";
        }

    }
}
```

Здесь предыдущий пример немного изменен для создания таймера обратного отсчета:

```
package {

  import flash.display.Sprite;
  import flash.text.TextField;
  import flash.utils.Timer;
  import flash.events.TimerEvent;
  import flash.utils.getTimer;

  public class Example extends Sprite {

    private var _text:TextField;
    private var _timer:Timer;
    private var _start:uint;
    private var _count:uint = 20;

    public function Example() {
      _start = getTimer();
      _text = new TextField();
      addChild(_text);
      _timer = new Timer(500);
      _timer.addEventListener(TimerEvent.TIMER, onTimer);
      _timer.start();
    }

    private function onTimer(event:TimerEvent):void {
      var elapsed:int = Math.round(_count - (getTimer() - _start) / 1000);
      _text.text = elapsed + " секунд";
      if(elapsed < 0) {
         _text.text = "---------";
         _timer.stop();
      }
    }
```

```
    }
}
```

В предыдущем примере истекшее время вычислялось с помощью объектов *Date*. Когда необходимо добавлять или вычитать годы и месяцы, полагаться на константы нельзя, потому что число миллисекунд в месяце меняется в зависимости от количества дней в нем, а в високосном году больше миллисекунд, чем в обычном. Однако класс *Date* прозрачно обрабатывает подобные вычисления, привлекая для этого методы *геттеры* и *сеттеры*. Самый эффективный способ вычисления дат – посредством метода *DateUtilities.addTo( )*, принимающего до восьми параметров, семь из которых описаны в табл. 14.2. Первый параметр обязателен. Это объект *Date*, относительно которого производятся вычисления. Все параметры, перечисленные в табл. 14.2, могут отсутствовать и могут принимать как положительные, так и отрицательные значения.

Таблица 14.2. Параметры метода addTo( )

Параметр	Описание
years	Число лет, добавляемых к дате.
months	Число месяцев, добавляемых к дате.
days	Число дней, добавляемых к дате.
hours	Число часов, добавляемых к дате.
minutes	Число минут, добавляемых к дате.
seconds	Число секунд, добавляемых к дате.
milliseconds	Число миллисекунд, добавляемых к дате.

Ниже показано, как посредством метода *addTo( )*добавлять (и вычитать) года, месяцы и дни:

```
var example:Date = new Date(2010, 0, 5, 10, 25);

trace(DateUtilities.addTo(example, 10));
trace(DateUtilities.addTo(example, -4));
trace(DateUtilities.addTo(example, 0, 1, 1));
trace(DateUtilities.addTo(example, 0, -1, -1));

/* На экран выводится:
Sun Jan 5 10:25:00 GMT-0800 2020
Thu Jan 5 10:25:00 GMT-0800 2006
Sat Feb 6 10:25:00 GMT-0800 2010
Fri Dec 4 10:25:00 GMT-0800 2009
*/
```

Данный пример демонстрирует создание нового объекта *Date* из существующего объекта *Date* на основании прошедшего времени. Однако может потребоваться вычислить истекшее время между двумя сущест-

вующими объектами *Date*, что не так просто, как может показаться. Вычитание значения свойства *time* одного объекта *Date* из другого не обеспечивает универсального решения в данном случае. Хотя эта операция и дает количество миллисекунд между двумя датами, результат сложно обработать, если идет речь о периоде, большем чем один день. Преобразовывать миллисекунды в количество лет, месяцев и дней затруднено тем, что количество дней в месяцах и годах разное. Еще сложнее обрабатывать отрицательные значения истекшего времени.

В данном случае удобно создать объект *Date*, представляющий прошедший период времени. Это позволяет вычислять количество годов, месяцев и дней, разделяющих две даты, посредством встроенных методов класса *Date*. Большинство смещений можно получить с помощью *геттеров* UTC, как показано в следующем примере:

```
var one:Date = new Date();

var two:Date = DateUtilities.addTo(one, 4, 1, 3, 10);

var elapsed:Date = new Date(two.time - one.time);

trace(elapsed);  // На экран выводится: Sun Feb 3 16:00:00 GMT-0800 1974
trace(elapsed.hoursUTC);  // На экран выводится: 10
trace(elapsed.monthUTC);  // На экран выводится: 1
```

Однако здесь есть несколько предостережений. ActionScript сохраняет даты в относительном исчислении от начала эпохи, но большинство методов класса *Date* возвращают абсолютные значения дат. Так, свойство *year* для 1971 года возвращает значение 1971, а не 1. Чтобы использовать объект времени *elapsed*, необходимо вычесть константы из значений *year* и *day*. (Значения *month*, *hour*, *minute*, *second* и *millisecond* для времени эпохи равны 0, поэтому специальные методы не возвращают относительные значения для этих значений.) Вычитаем 1970 из значения года и затем вычитаем 1 из дня:

```
trace(elapsed.dateUTC);        // На экран выводится: 4 (incorrect)
trace(elapsed.dateUTC - 1);    // На экран выводится: 3 (correct)
trace(elapsed.dateUTC);        // На экран выводится: 1974 (incorrect);
trace(elapsed.dateUTC - 1970); // На экран выводится: 4 (correct);
```

Для более эффективной работы с истекшими периодами времени можно задействовать некоторые методы класса *DateUtilities*. У него есть статический метод *elapsed( )* (истекший), который возвращает количество истекшего времени между двумя датами, как показано в следующем примере:

```
var one:Date = new Date();

var two:Date = DateUtilities.addTo(one, 4, 1, 3);

var elapsed:Object = DateUtilities.elapsed(two);
for(var item:String in elapsed) {
  trace(item + ": " + elapsed[item]);
```

```
}

/* На экран выводится:
milliseconds: 0
seconds: 0
minutes: 0
hours: 0
days: 3
months: 1
years: 4
*/
```

Если передан только один параметр, то метод *elapsed( )* предполагает, что требуется вычислить количество времени, прошедшее между заданной и текущей датами.

В классе *DateUtilities* также есть несколько дополнительных статических методов, которые возвращают истекшее время между двумя датами в заданных интервалах (*годах*, *месяцах*, *днях* и т. д.). По умолчанию каждый из этих методов вычисляет общее число заданных интервалов между двумя датами, как показано в данном фрагменте кода:

```
var one:Date = new Date();

var two:Date = DateUtilities.addTo(one, 4, 1, 3);

trace(DateUtilities.elapsedYears(two, one));   // На экран выводится: 4
trace(DateUtilities.elapsedMonths(two, one));  // На экран выводится: 49
trace(DateUtilities.elapsedDays(two, one));    // На экран выводится: 1495
trace(DateUtilities.elapsedHours(two, one));   // На экран выводится: 35880
trace(DateUtilities.elapsedMinutes(two, one));      // На экран выводится:
                                                    // 2152800
trace(DateUtilities.elapsedSeconds(two, one));      // На экран выводится:
                                                    // 129168000
trace(DateUtilities.elapsedMilliseconds(two, one)); // На экран выводится:
                                                    // 129168000000
```

В любой из этих методов может передаваться необязательный третий параметр – Булево значение. Если задано `true`, метод возвращает относительные значения, а не абсолютные. Исключением является метод *elapsedYears( )*, который в любом случае возвращает одно и то же значение, как показано в следующем примере:

```
var one:Date = new Date();

var two:Date = DateUtilities.addTo(one, 4, 1, 3);

trace(DateUtilities.elapsedMonths(two, one, true));   // На экран выводится: 1
trace(DateUtilities.elapsedDays(two, one, true));     // На экран выводится: 3
trace(DateUtilities.elapsedHours(two, one, true));    // На экран выводится: 0
trace(DateUtilities.elapsedMinutes(two, one, true));  // На экран выводится: 0
trace(DateUtilities.elapsedSeconds(two, one, true));  // На экран выводится: 0
trace(DateUtilities.elapsedMilliseconds(two, one, true));  // На экран
                                                           // выводится: 0
```

14.9. Получение даты из строки

Задача

Требуется создать объект *Date* из строки.

Решение

Это делается при помощи метода *parse( )* объекта *DateFormat*.

Обсуждение

ActionScript не предоставляет собственных методов для синтаксического разбора строк и использования полученных данных для создания объекта *Date*. Во многих случаях в этом нет ничего трудного. Например, технология Flash Remoting позволяет возвращать собственные объекты *Date* из других приложений. Даже если обходиться без Flash Remoting, Flash может обмениваться значениями с другими приложениями, используя секунды/миллисекунды эпохи. Однако для преобразования строки в дату надо применять метод *parse( )* (разобрать) класса *ascb.util.DateFormat*. Этот метод принимает строковое значение как параметр, проводит его синтаксический разбор на составляющие даты (год, час и т. д.) и возвращает новый объект *Date*.

Чтобы использовать класс *DateFormat*, с помощью конструктора необходимо создать его экземпляр. При создании в конструктор в качестве параметра передается строка-маска. Подробнее о создании масок рассказывается в рецепте 14.4:

```
var formatter:DateFormat = new DateFormat("m/d/Y");
```

Создав экземпляр *DateFormat*, можно вызывать метод *parse( )*. Передаем в него строку в формате, заданном маской, он проводит ее синтаксический анализ и возвращает новый экземпляр *Date*:

```
// На экран выводится: Sat May 1 00:00:00 GMT-0700 2010
// (смещение часового пояса может меняться)
trace(formatter.parse("05/01/2010"));

formatter.mask = "m/d/Y 'at' h:i a";

// На экран выводится: Sat May 1 22:25:00 GMT-0700 2010
// (смещение часового пояса может меняться)
trace(formatter.parse("05/01/2010 at 10:25 PM"));
```

См. также

Рецепт 14.4.

15

Программирование звука

15.0. Введение

Удачно подобранный звук способен значительно улучшить впечатление пользователя от приложения, игры или веб-сайта. Flash IDE позволяет импортировать звук в библиотеку, размещать в кадрах на временной диаграмме, подключать к клипам и т. д. Данная глава посвящена вопросам программирования звука в ActionScript 3.0 с помощью класса *Sound* и связанных с ним классов.

Класс *Sound* предназначен для загрузки и потокового воспроизведения внешних звуковых файлов в формате MP3. Поскольку эти файлы не встроены в *.swf* и загружаются с внешнего URL, здесь действуют ограничения для обеспечения безопасности доменов, оговоренные в главе 3.

В данной главе рассматриваются классы:

- *Sound*
- *SoundChannel*
- *SoundLoaderContext*
- *SoundMixer*
- *SoundTransform*

Все они входят в пакет *flash.media*, поэтому при работе с примерами надо в первую очередь убедиться, что `flash.media.Sound` и все остальные необходимые классы импортированы.

15.1. Создание объекта Sound и загрузка звука

Задача

Требуется загрузить звук в *.swf* и использовать его в приложении.

Решение

Создайте объект *Sound* и загрузите в него внешний звуковой файл.

Обсуждение

Создать объект *Sound* так же просто, как создать экземпляр любого класса. Однако прежде всего необходимо убедиться, что класс *Sound* импортирован:

```
import flash.media.Sound;
```

и затем создать его экземпляр:

```
_sound = new Sound();
```

Теперь, конечно, необходимо задать звуковой файл, который будет воспроизводиться. Как говорилось ранее, это будет внешний MP3-файл, например песня. Во всех примерах данной главы фигурирует MP3-файл *song.mp3*, хранящийся на сервере или на жестком диске в том же каталоге, что и *.swf*.

Чтобы загрузить звуковой файл в объект *Sound*, сначала создается объект *URLRequest* (необходимо импортировать *flash.net.URLRequest*), в который передается строка, содержащая путь к MP3-файлу:

```
soundFile = new URLRequest("song.mp3");
```

Затем этот объект можно передать в метод загрузки объекта *Sound*:

```
_sound.load(soundFile);
```

Можно даже объединить два последних шага, например так:

```
package {
    import flash.display.Sprite;
    import flash.media.Sound;
    import flash.net.URLRequest;

    public class LoadSoundExample extends Sprite {
        private var _sound:Sound;

        public function LoadSoundExample () {
            _sound = new Sound();
            _sound.load(new URLRequest("song.mp3"));
        }
    }
}
```

Теперь у данного класса есть свойство `_sound`, пригодное для воспроизведения этого звука. Обратите внимание, что это не обеспечивает начало воспроизведения звука. Мы просто настроили и предоставили информацию о звуке, который будет использоваться. Управление воспроизведением рассмотрено в рецепте 15.2.

Еще проще передать *URLRequest* прямо в конструктор при создании звукового объекта:

```
public function LoadSoundExample() {
    _sound = new Sound(new URLRequest("song.mp3"));
}
```

Когда запрос передается в конструктор, как здесь, `sound` сразу же автоматически вызывает собственный метод *load( )* и начинает загрузку звуковых данных. Это особенно полезно, если в объект *Sound* загружается всего один звуковой файл.

В других случаях лучше создавать объект *Sound* заранее и вызывать метод *load( )*, когда уже известно, какие звуковые файлы требуется загрузить. В качестве примера можно привести проигрыватель, в котором пользователь может выбирать песню из списка воспроизведения. Когда песня выбрана, приложение передает ее путь в метод *load( )* звукового объекта, после чего она готова к воспроизведению.

15.2. Начало и прекращение воспроизведения звука

Задача

Вы создали объект *Sound* и начали потоковую передачу звукового файла в него. Теперь требуется воспроизвести его и через некоторое время остановить.

Решение

Воспроизведение звука запускается посредством метода *play( )*. Остановить потоковую передачу звука позволяет метод *close( )*.

Обсуждение

Звук, загруженный в объект *Sound*, воспроизводится при помощи метода *play( )* этого объекта.

```
_sound = new Sound(new URLRequest("song.mp3"));
_sound.play();
```

Вот и все. В методе *play( )* есть некоторые дополнительные необязательные параметры, которые рассматриваются в рецептах 15.1 и 15.10.

Метод *close( )* объекта *Sound* прекращает не только воспроизведение звука, но и потоковую передачу звукового файла. Чтобы опять воспроизвести этот звук, необходимо вызвать метод *load( )* и повторно запустить загрузку звуковых данных. Метод *close( )* должен вызываться, только когда работа с данным конкретным звуком полностью завершена. В рецепте 15.10 обсуждается класс *SoundChannel*, который обеспечивает возможность остановить воспроизведение звука, не прерывая потоковую передачу.

См. также

Рецепт 15.1, в котором рассказывается, как загружать внешние звуковые файлы, и рецепт 15.10.

15.3. Как задать аудиобуфер

Задача

Требуется обеспечить равномерное воспроизведение звука.

Решение

Задайте размер буфера в единицах времени с помощью класса *SoundLoaderContext*.

Обсуждение

Класс *Sound* осуществляет потоковую загрузку аудиоданных во время воспроизведения звука, т. е воспроизведение начинается еще до того, как все данные полностью загружены. Это особенно важно для больших звуковых файлов, таких как песни, размер которых может достигать нескольких мегабайт.

В зависимости от кодировки звуковых данных и доступной пропускной способности интернет-соединения звук может воспроизводиться быстрее, чем загружаться. В этом случае воспроизведение приостанавливается до загрузки следующей порции данных. Во избежание этого задается буфер. Тогда объект *Sound* будет загружать определенный объем звуковых данных до начала воспроизведения. При этом если возникает кратковременное снижение пропускной способности, без всяких пауз продолжается воспроизведение данных, накопленных в буфере.

По умолчанию объект *Sound* при загрузке звукового файла создает односекундный буфер. Иначе говоря, даже если звук должен воспроизводиться немедленно, объект *Sound* ожидает, пока не будут загружены аудиоданные, воспроизведение которых займет одну секунду. Кроме того, если все данные буфера выбраны и воспроизведение приостанавливается в ожидании дозагрузки данных, возобновлено оно будет только после того, как объект заполнит этот односекундный буфер.

Если параметры сети неустойчивы или звуковые файлы характеризуются высоким битрейтом (при котором требуется большее количество бит информации на секунду воспроизведения), то для обеспечения равномерного воспроизведения может потребоваться больший буфер. Его можно реализовать, создав объект *SoundLoaderContext* (контекст загрузчика звука). При создании объекта в его конструктор передается размер буфера в миллисекундах. Например, пятисекундный буфер создается так:

```
buffer = new SoundLoaderContext(5000);
```

Этот объект может применяться при любом методе загрузке звука – в конструкторе объекта *Sound* или в методе *load( )* (см. рецепт 15.1) – достаточно передать его как второй параметр после параметра *URLRequest*:

```
var request:URLRequest = new URLRequest("song.mp3");
var buffer:SoundLoaderContext = new SoundLoaderContext(5000);
_sound = new Sound(request, buffer);
_sound.play();
```

или:

```
var request:URLRequest = new URLRequest("song.mp3");
var buffer:SoundLoaderContext = new SoundLoaderContext(5000);
_sound = new Sound();
_sound.load(request, buffer);
_sound.play();
```

См. также

Рецепт 15.1 посвящен загрузке внешнего звукового файла.

15.4. Смещение начала воспроизведения

Задача

Требуется воспроизводить звук не сначала файла, а с некоторым смещением.

Решение

Передайте момент начала воспроизведения в метод *play( )*.

Обсуждение

Иногда звуковой файл требуется воспроизводить не с начала, а сместив начало воспроизведения. Объект *Sound* предоставляет средства для очень точного управления звуковыми данными, поэтому момент начала воспроизведения можно задать с точностью до миллисекунд.

Если метод *play( )* вызывается без параметров, он начинает воспроизведение с самого начала файла. Но в него может быть передан необязательный параметр, определяющий, с какой миллисекунды звукового файла начинается воспроизведение. Например, так мы определяем начало воспроизведения, отстоящее точно на 5,5 секунды от начальной точки:

```
_sound.play(5500);
```

Это особенно полезно, когда надо пропустить начало звукового файла. Если нельзя отредактировать звуковой файл и вырезать эту часть, необходимо просто задать момент начала и перескочить через нее.

В качестве другого примера возьмем создание системы сигнальных точек в звуковом объекте. Скажем, есть запись речи и требуется предоставить пользователям возможность прослушивать любую из ее частей. Выбрав точки начала каждой из них, можно определить массив значений сигнальных точек. Тогда, задавая сигнальную точку начала воспроизведения звука, можно получать момент начала из массива и передавать его в метод *play( )*. Следующий класс демонстрирует простой пример:

```
package {
    import flash.display.Sprite;
    import flash.media.Sound;
    import flash.net.URLRequest;

    public class CuePoints extends Sprite {
        private var _sound:Sound;
        private var _cuePoints:Array;

        public function CuePoints() {
            _cuePoints = [0, 10000, 30000, 68000, 120000];
            _sound = new Sound(new URLRequest("song.mp3"));

            // Воспроизводим с третьей сигнальной точки (30 секунд от начала)
            playCuePoint(2);
        }

        public function playCuePoint(index:int):void {
            _sound.play(_cuePoints[index]);
        }

    }
}
```

Конечно, в полном приложении для выбора сигнальной точки были бы реализованы кнопки или какой-то другой элемент пользовательского интерфейса, который передавал бы индекс точки в метод `playCuePoint()`.

В рецепте 15.10 описывается создание в проигрывателе функции прерывания/возобновления воспроизведения.

См. также

Рецепты 15.10 и 15.2.

15.5. Многократное (циклическое) воспроизведение звука

Задача

Требуется воспроизводить звуковой файл несколько раз или повторять его в непрерывном цикле.

Решение

Задать параметр цикла в методе *play( )*.

Обсуждение

По умолчанию звуковой файл воспроизводится один раз – от начала до конца. Иногда требуется проиграть звук более одного раза или даже воспроизводить его непрерывно, например короткий музыкальный фрагмент, играющий роль фона в игре или на веб-сайте. Можно организовать непрерывное воспроизведение этого фрагмента, тогда он будет звучать как одна длинная песня.

Количество циклов (или повторений) воспроизведения задается вторым параметром метода *play( )*. Конечно, в этом случае надо определить и значение первого необязательного параметра – смещение точки начала (см. рецепт 15.1). Для того чтобы задать циклическое воспроизведение без смещения, в качестве первого параметра надо передать ноль:

```
_sound.play(0, 3);
```

В этом случае звуковой файл воспроизводится с самого начала (нулевое смещение) и повторяется три раза.

Минимальное допустимое значение параметра цикла – единица, при этом звук воспроизводится один раз. Если передается ноль (или даже отрицательное число), звук все равно будет воспроизведен только один раз.

Нет возможности автоматически организовать бесконечное воспроизведение звука. Однако можно передать в метод очень большое значение. Проще всего передать `int.MAX_VALUE` – максимально возможное число в переменной типа `integer`, равное 2 147 483 647. Звук продолжительностью в одну секунду будет воспроизводиться в течение почти 70 лет, что для данной задачи вполне можно считать «бесконечностью».

См. также

Рецепты 15.1 и 15.2.

15.6. Как получить размер звукового файла

Задача

Необходимо узнать общий размер загружаемого аудиофайла MP3, а также объем загруженных на данный момент данных (чтобы можно было создать визуальный индикатор потоковой передачи).

Решение

Вам потребуются свойства *bytesTotal* и *bytesLoaded* объекта *Sound*.

Обсуждение

При потоковой передаче аудиофайла нелишним будет предоставить пользователю информацию о том, сколько данных уже загружено. Flash Player загружает звук с максимально возможной скоростью и воспроизводит его с той скоростью передачи данных, с которой он был закодирован. В идеале звуковые данные загружаются быстрее, чем воспроизводятся, поэтому звук не прерывается в ожидании загрузки следующей порции данных. Однако при плохом соединении или низкой пропускной способности интернет-соединения, или при кодировании звука с высоким битрейтом, проигрыватель Flash может не справиться с непрерывным воспроизведением.

Поэтому лучше обеспечить пользователю некоторое визуальное представление процесса загрузки аудиоданных в буфер и воспроизведения звукового файла. Такое представление можно найти во многих медиапроигрывателях, например в Windows Media Player или QuickTime Player. Обычно оно имеет вид индикатора хода воспроизведения. Фон индикатора может быть белым. По мере воспроизведения фон закрывается черной полосой, показывающей, какая часть песни прослушана. Кроме того, можно ввести и серую полосу, которая будет показывать объем буферизованных данных. Пока серая полоса (буфер) опережает черную (точка воспроизведения), воспроизведение будет непрерывным. Когда они совпадают, проигрыватель останавливает воспроизведение (черная полоса замирает), а серая полоса выдвигается вперед, показывая, что некоторый объем данных поступает в буфер. Когда буфер заполняется, воспроизведение возобновляется, черная полоса вновь удлиняется.

В этом рецепте показано, как создать серую полосу с помощью свойств объекта *Sound*: *bytesTotal* (общее количество байт) и *bytesLoaded* (количество загруженных байт). (В рецепте 15.9 показано, как создавать черную полосу.) Имена этих свойств говорят сами за себя: *bytesTotal* содержит общий размер загружаемого MP3-файла, а *bytesLoaded* показывает, какой объем этих данных загружен. Разделив два значения друг на друга, получаем загруженную часть данных в процентном выражении.

В следующем примере описывается обработчик *enterFrame*. В каждом кадре он отрисовывает белый прямоугольник и вычисляет, какая часть звукового файла загружена в процентном соотношении. Эти данные он использует для отрисовки другого прямоугольника (серого, конечно), который представляет эту долю.

```
package {
    import flash.display.Sprite;
    import flash.media.Sound;
    import flash.net.URLRequest;
    import flash.events.Event;
```

```
    public class ProgressBar extends Sprite {
        private var _sound:Sound;

        public function ProgressBar() {
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            _sound = new Sound(new URLRequest("song.mp3"));
            _sound.play();
        }

        public function onEnterFrame(event:Event):void
        {
            var barWidth:int = 200;
            var barHeight:int = 5;
            var loaded:int = _sound.bytesLoaded;
            var total:int = _sound.bytesTotal;
            if(total > 0) {
                // Отрисовываем фоновую полосу
                graphics.clear();
                graphics.beginFill(0xFFFFFF);
                graphics.drawRect(10, 10, barWidth, barHeight);
                graphics.endFill();

                // Доля загруженных аудиоданных
                // в процентном выражении
                var percent:Number = loaded / total;

                // Отрисовываем полосу, представляющую
                // долю загруженных данных
                graphics.beginFill(0xCCCCCC);
                graphics.drawRect(10, 10,
                                  barWidth * percent, barHeight);
                graphics.endFill();
            }
        }
    }
}
```

Если воспроизведение звука происходит с жесткого диска или локального сервера, движение индикатора выполнения невозможно заметить; звук загружается практически моментально, полоса просто быстро переходит к отметке 100%. Если это возможно, попытайтесь разместить MP3-файл где-то на внешнем веб-сервере и организуйте к нему доступ через Интернет. Это обеспечит более точную картину того, что будет видеть конечный пользователь. Также броузеры обычно кэшируют аудиофайлы при воспроизведении, поэтому чтобы обеспечить более реалистичное отображение буферизации, перед тестированием необходимо очистить кэш броузера.

См. также

Рецепт 15.1, в котором рассмотрена загрузка внешних звуковых файлов, и рецепт 15.7.

15.7. Чтение тега ID3 звукового файла

Задача

Требуется получить доступ к информации о воспроизводимом MP3-файле, такой как название песни, исполнитель, альбом, жанр и т. д.

Решение

Обратитесь к свойству *id3* объекта *Sound*.

Обсуждение

Аудиофайлы в формате MP3 могут содержать множество метаданных. Чаще всего метаданные в музыкальных файлах содержат названия песни и альбома, имена исполнителя и композитора, жанр, год выхода и т. д. Количество информации, включенной в эти теги, зависит от того, кто кодировал файл или расставлял теги. Однако в большинстве случаев, как минимум, присутствуют теги *songname* (название песни) и *artist* (исполнитель).

Эти данные можно получить посредством свойства *id3* объекта *Sound*.

Это свойство представляет собой экземпляр класса *flash.media.ID3Info* и содержит следующие свойства:

- `album`
- `artist`
- `comment`
- `genre`
- `songName`
- `track`
- `year`

Таким образом, имя песни можно получить следующим образом:

```
_sound.id3.songName
```

Однако здесь есть одна тонкость: доступ к тегам *id3* невозможен до тех пор, пока они не будут загружены в *.swf*. Если попытаться прочитать тег сразу после создания объекта *Sound* или вызова *play( )*, он будет не определен. Фактически, в этот момент запрос к аудиофайлу еще даже не достиг сервера, поэтому, конечно, данные тега недоступны.

Как же узнать, что данные *id3* доступны? К счастью, объект *Sound* рассылает событие *ID3*, которое можно слушать. Событие сформировано, значит, можно обращаться к тегам *id3*. Класс *Sound* расширяет класс *EventDispatcher*, поэтому слушать это событие, которое определено как *flash.events.Event.ID3*, можно посредством метода addEventListener, и назначать метод-обработчик, который будет читать теги *id3*.

В следующем примере создается текстовое поле, в которое выводятся все доступные теги *id3*:

```
package {
    import flash.display.Sprite;
    import flash.media.Sound;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.text.TextField;

    public class ID3Reader extends Sprite {
        private var _sound:Sound;

        public function ID3Reader () {
            _sound = new Sound(new URLRequest("song.mp3"));
            _sound.addEventListener(Event.ID3, onID3);
            _sound.play();
        }

        public function onID3(event:Event):void {
            // Создаем текстовое поле и отображаем его
            var id3Display:TextField = new TextField();
            addChild(id3Display);
            id3Display.x = 10;
            id3Display.y = 20;
            id3Display.width = 200;
            id3Display.height = 200;
            id3Display.background = true;
            id3Display.multiline = true;
            id3Display.wordWrap = true;

            // Выводим информацию о песне в текстовое поле
            id3Display.text += _sound.id3.songName + "\n";
            id3Display.text += _sound.id3.artist + "\n";
            id3Display.text += _sound.id3.album + "\n";
            id3Display.text += _sound.id3.year + "\n";            }
        }
    }
}
```

См. также

Рецепт 15.1, чтобы узнать, как загружать внешние звуковые файлы, и рецепт 15.6.

15.8. Как узнать, когда закончилось воспроизведение звукового файла

Задача

Требуется определить момент окончания воспроизведения звукового файла.

Решение

Слушайте событие *soundComplete*.

Обсуждение

Существует масса причин, по которым может потребоваться отслеживать момент окончания воспроизведения звука. Например, если имеется аудиовступление, которое должно воспроизводиться перед тем, как пользователь перейдет к следующему разделу сайта. Или в музыкальном проигрывателе, воспроизводящем список песен, прежде чем начать новую песню, необходимо знать, что предыдущая завершена.

В этом и нескольких последующих рецептах мы будем работать с еще одним классом пакета *flash.media*, *SoundChannel* (звуковой канал). Метод *play( )* объекта *Sound* возвращает объект *SoundChannel*. Таким образом, каждый воспроизводимый в *.swf* аудиофайл представлен отдельным объектом *SoundChannel*. Конечный аудиовыход формируется объединением этих каналов.

По завершении воспроизведения звукового файла соответствующий ему объект *SoundChannel* рассылает событие *soundComplete* (звук завершен). Оно определено как *flash.events.Event.SOUND_COMPLETE*. В объект *SoundChannel* можно ввести слушателя этого события и предоставить метод-обработчик, который будет вызываться по завершении воспроизведения.

В следующем примере создается простой список воспроизведения:

```
package {
    import flash.display.Sprite;
    import flash.media.Sound;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.media.SoundChannel;

    public class PlayList extends Sprite {
        private var _sound:Sound;
        private var _channel:SoundChannel;
        private var _playList:Array;      // список песен
        private var _index:int = 0;       // текущая песня

        public function PlayList() {
            // Создаем список воспроизведения
            // и начинаем воспроизведение
            _playList = ["song1.mp3",
                         "song2.mp3",
                         "song3.mp3"];
            playNextSong();
        }

        private function playNextSong():void
        {
```

```
            // Если в списке воспроизведения еще есть песни
            if(_index < _playList.length) {
                // Создаем новый объект Sound,
                // загружаем и воспроизводим его.
                // _playList[_index] содержит имя
                // и путь следующей песни
                _sound = new Sound();
                _sound.load(new URLRequest(_playList[_index]));
                _channel = _sound.play();

                // Добавляем слушателя событий channel

                _channel.addEventListener(Event.SOUND_COMPLETE,
                                          onComplete);

                // Даем приращение счетчику
                _index++;
            }
        }

        public function onComplete(event:Event):void
        {
            playNextSong();
        }
    }
}
```

Здесь индексная переменная отсчитывается от нуля. Таким образом, `playList[index]` интерпретируется как `"song.mp3"`, которая загружается первой. Затем индекс получает приращение. Когда формируется событие *soundComplete*, метод *playNextSong( )* загружает следующую песню из `playList`. Это происходит до тех пор, пока значение индекса не превысит количество песен в списке воспроизведения.

См. также

Рецепт 15.1, в котором рассказывается, как загружать внешние звуковые файлы.

15.9. Отслеживание процесса воспроизведения звука

Задача

Требуется знать местоположение головки воспроизведения в текущей песне, чтобы видеть, какая часть песни прослушана.

Решение

Задача решается с помощью свойства *Sound.length*, позволяющего определить длительность песни, и свойства *SoundChannel.position*, показывающего, какая часть ее воспроизведена.

Обсуждение

В рецепте 15.6 обсуждалось, как добавить индикатор выполнения, который показывает не только текущую позицию воспроизведения звукового файла, но и объем файла, загруженный в проигрыватель. В том рецепте была создана часть индикатора, отвечающая за отображение процесса загрузки данных.

Данный рецепт рассматривает другую часть: отображение процесса воспроизведения звукового файла. Для этого необходимо знать длину аудиофайла и текущее положение головки воспроизведения. Может показаться, что это нелогично но два данные свойства находятся в двух разных классах. Длительность звукового файла – это свойство объекта *Sound*, а текущее положение головки воспроизведения – это часть класса *SoundChannel*. Аналогично процессу создания индикатора процесса буферизации, эти два значения сравниваются для получения воспроизведенной части звукового файла в процентном отношении к общей длительности файла.

К сожалению, на этот раз все немного сложнее, чем в случае с индикатором буферизации. Проблема в том, что точное значение свойства *length* неизвестно, пока не загружен весь звуковой файл. До тех пор свойство отражает размер загруженных данных. Таким образом, если, к примеру, на данный момент загружено 10% десятиминутного аудиофайла, свойство *length* сообщает, что длительность файла 1 минута. (На самом деле оба свойства, *length* и *position*, возвращают время в миллисекундах, но если требуется выводить эти значения на экран, их можно преобразовать в минуты и секунды.)

К счастью, фактическую длительность звукового файла можно оценить, произведя некоторые нехитрые вычисления. Свойство *length* возвращает длительность некоторой части звукового файла. Если это значение разделить на долю, которую оно составляет от общей длительности файла, результат будет очень близок к реальной длительности. В нашем примере *length* показывает время в одну минуту. Однако это $^1/_{10}$ длительности загружаемого файла. Если разделить 1 на $^1/_{10}$ (то же самое, что умножить на 10), получаем длительность файла, равную 10 минутам.

Часть в данном случае определяется соотношением *bytesLoaded*/*bytesTotal*, которое уже вычислено, хранится в переменной и используется для отрисовки индикатора буферизации. Таким образом, чтобы скорректировать длительность, необходима всего одна строка кода:

```
length /= percentBuffered;
```

Как и в рецепте 15.2, более реалистичная картина работы этого кода будет получена, если организовать доступ к MP3-файлу через веб и очищать кэш веб-броузера перед каждой загрузкой.

Следующий пример показывает оба индикатора вместе:

```
package {
    import flash.display.Sprite;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;
    import flash.events.Event;

    public class ProgressBar2 extends Sprite {
        private var _sound:Sound;
        private var _channel:SoundChannel;

        public function ProgressBar2() {
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            _sound = new Sound(new URLRequest("song.mp3"));
            _channel = _sound.play();
        }

        public function onEnterFrame(event:Event):void
        {
            var barWidth:int = 200;
            var barHeight:int = 5;

            var loaded:int = _sound.bytesLoaded;
            var total:int = _sound.bytesTotal;

            var length:int = _sound.length;
            var position:int = _channel.position;

            // Отрисовываем фоновую полосу
            graphics.clear();
            graphics.beginFill(0xFFFFFF);
            graphics.drawRect(10, 10, barWidth, barHeight);
            graphics.endFill();

            if(total > 0) {
                // Загруженная часть звукового файла
                // в процентном выражении
                var percentBuffered:Number = loaded / total;

                // Отрисовываем полосу, представляющую,
                // какая часть звукового файла загружена
                graphics.beginFill(0xCCCCCC);
                graphics.drawRect(10, 10,
                                  barWidth * percentBuffered,
                                  barHeight);
                graphics.endFill();

                // Корректируем длительность аудиофайла
                length /= percentBuffered;

                  // Воспроизведенная часть аудиофайла
                  // в процентном выражении
                  var percentPlayed:Number = position / length;
```

```
                    // Отрисовываем полосу, представляющую
                    // воспроизведенную часть звукового файла.
                    graphics.beginFill(0x666666);
                    graphics.drawRect(10, 10,
                                      barWidth * percentPlayed,
                                      barHeight);
                    graphics.endFill();
                }
            }
        }
    }
```

См. также

Рецепт 15.1, в котором рассказывается, как загружать внешние звуковые файлы, и рецепт 15.2.

15.10. Приостановка и возобновление воспроизведения звука

Задача

Требуется приостановить воспроизведение звука и возобновить его позже с этого же момента.

Решение

Обратите внимание на значение свойства *position* соответствующего *SoundChannel*. Возобновляя воспроизведение, используйте его как смещение.

Обсуждение

Как отмечалось в рецепте 15.2, можно вызвать метод *close( )* объекта *Sound* и остановить воспроизведение аудиофайла. Однако это прекращает и потоковую передачу данных, поэтому, чтобы возобновить воспроизведение, понадобится повторно вызывать метод *load( )*.

К счастью, класс *SoundChannel* предлагает метод *stop( )*, который останавливает воспроизведение, не оказывая влияния на загрузку. Воспроизведение возобновляется повторным вызовом метода *play( )*.

Однако следует заметить, что метод *play( )* воспроизводит звуковой файл с начала. Такое поведение предполагается для кнопки «stop» (стоп) многих медиапроигрывателей. А вот над реализацией кнопки «pause» (пауза) придется немного потрудиться. Идея в том, чтобы создать метод-обработчик, который будет вызываться по нажатию кнопки «pause». Данная функция будет читать и сохранять текущее значение свойства *position* объекта *SoundChannel*, что обеспечит время в миллисекундах от начала файла. Сохраняем это значение в переменной клас-

са, чтобы иметь к ней доступ позже. Теперь кнопка паузы становится кнопкой воспроизведения. При повторном ее нажатии вызывается метод *play( )* объекта *Sound*, в который передается сохраненное положение головки воспроизведения. Таким образом, воспроизведение аудиофайла начинается с этого момента, т. е. именно с той точки, в которой было прервано. Следующий пример реализует данную стратегию:

```
package {
    import flash.display.Sprite;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.display.Sprite;
    import flash.events.MouseEvent;

    public class PlayPause extends Sprite {
        private var _sound:Sound;
        private var _channel:SoundChannel;
        private var _playPauseButton:Sprite;
        private var _playing:Boolean = false;
        private var _position:int;

        public function PlayPause() {
            // Создаем звуковой объект
            // и начинаем его воспроизведение
            _sound = new Sound(new URLRequest("song.mp3"));
            _channel = _sound.play();
            _playing = true;

            // Спрайт, используемый как кнопка Play/Pause
            _playPauseButton = new Sprite();
            addChild(_playPauseButton);
            _playPauseButton.x = 10;
            _playPauseButton.y = 20;
            _playPauseButton.graphics.beginFill(0xcccccc);
            _playPauseButton.graphics.drawRect(0, 0, 20, 20);
            _playPauseButton.addEventListener(MouseEvent.MOUSE_UP,
                                              onPlayPause);
        }

        public function onPlayPause(event:MouseEvent):void {
            // Если идет воспроизведение, останавливаем.
            // Замечаем положение
            if(_playing) {
                   _position = _channel.position;
                   _channel.stop();
            }
            else {
                // Если не воспроизводится, возобновляем
                // воспроизведение с последней известной точки
                _channel = _sound.play(_position);
            }
```

```
                _playing = !_playing;
            }
        }
    }
```

В данном коде создается спрайт, выполняющий роль кнопки. В обработчике события отпускания кнопки мыши, если идет воспроизведение, звук прерывается и сохраняется точка, в которой воспроизведение приостановлено. При следующем нажатии кнопки воспроизведение начинается с момента, в котором было приостановлено.

См. также

Рецепт 15.2.

15.11. Чтение уровня звука

Задача

Требуется знать громкость воспроизведения звука.

Решение

Задействуйте свойства *SoundChannel.leftPeak* и *SoundChannel.rightPeak*.

Обсуждение

Звуки характеризуются, в частности, силой, или *амплитудой*. ActionScript 3.0 обеспечивает возможность доступа к амплитуде отдельно для левого и правого каналов стереозвука посредством свойств *leftPeak* (левый пик) и *rightPeak* (правый пик) объекта *SoundChannel*, который создается в момент начала воспроизведения звукового файла.

Значения этих свойств лежат в диапазоне от 0,0 до 1,0, где 1,0 – это максимальное значение амплитуды. Не путайте с громкостью, которая является общей настройкой для звука и управляется через объект *SoundTransorm* (см. рецепт 15.14). Это уровень громкости звука конкретного экземпляра, который может быть изменен во время воспроизведения.

В следующем примере значения этих свойств считываются и на их основании создаются две полосы, длина которых соответствует текущей амплитуде каждого канала в каждом кадре:

```
package {
    import flash.display.Sprite;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;
    import flash.events.Event;
```

```
        public class SoundLevels extends Sprite {
            private var _sound:Sound;
            private var _channel:SoundChannel;

            public function SoundLevels() {
                addEventListener(Event.ENTER_FRAME, onEnterFrame);
                _sound = new Sound(new URLRequest("song.mp3"));
                _channel = _sound.play();
            }

            public function onEnterFrame(event:Event):void
            {
                var leftLevel:Number = _channel.leftPeak * 100;
                var rightLevel:Number = _channel.rightPeak * 100;
                graphics.clear();
                graphics.beginFill(0xcccccc);
                graphics.drawRect(10, 10, leftLevel, 10);
                graphics.endFill();
                graphics.beginFill(0xcccccc);
                graphics.drawRect(10, 25, rightLevel, 10);
                graphics.endFill();

            }
        }
    }
```

См. также

Рецепт 15.1, в котором рассказывается, как загружать внешние звуковые файлы, и рецепт 15.14.

15.12. Прекращение воспроизведения всех звуков

Задача

Требуется прекратить воспроизведение всех воспроизводимых в настоящий момент звуковых файлов.

Решение

Это делается с помощью метода *stopAll( )* объекта *SoundMixer*.

Обсуждение

Как уже говорилось, когда начинается воспроизведение звукового файла, создается объект *SoundChannel*. В *.swf* может воспроизводиться множество звуков, и каждому из них будет назначен собственный звуковой канал. Некоторые аспекты звука контролируются посредством самого объекта *Sound*, некоторые – через звуковой канал. Звук, исходящий из динамиков, представляет собой объединение всех воспроизводящихся в настоящий момент каналов. Отвечает за это объект

SoundMixer (звуковой микшер). Его свойства и методы применяются ко всем звукам ролика, поскольку они смешаны вместе.

Одним из этих методов является *stopAll( )* (остановить все). Как можно ожидать из его имени, данный метод прерывает все воспроизводящиеся в настоящее время в *.swf* звуки.

Хотя звук может значительно улучшить впечатление от веб-сайта, игры и т. д., надо предоставить возможность пользователю выключить его. Метод *stopAll( )* – самый простой способ сделать это. Это статический метод класса *SoundMixer*, поэтому вызывается прямо из него. Обычно вызов этого метода осуществляется в обработчике события кнопки или спрайта:

```
public function stopSounds(event:Event):void {
    SoundMixer.stopAll();
}
```

См. также

Рецепт 15.2.

15.13. Чтение спектрального состава звука

Задача

Требуется визуально представить звуки, воспроизводимые в *.swf*.

Решение

Это делается с помощью метода *SoundMixer.computeSpectrum( )*, позволяющего заполнить байтовый массив данными спектрального состава звука, воспроизводимого в *.swf*. Значения массива используются для визуализации этих данных.

Обсуждение

Одно из наиболее захватывающих дополнений к возможностям работы со звуком в ActionScript 3.0 – доступ к данным спектрального состава звука. Многие годы Flash-разработчики жаждали добраться до этих данных. Некоторые средства сторонних производителей позволяли читать их и сохранять в отдельный текстовый файл. Этот файл можно было читать и синхронизировать со звуком. Однако эти решения были слишком сложны и не очень эффективны. Теперь, когда данная возможность встроена в класс *SoundMixer*, и введен новый класс *ByteArray* (байтовый массив), можно без труда визуализировать спектр звуковых частот, написав всего пару строк кода.

Сначала рассмотрим класс *ByteArray*. Это новый специальный класс ActionScript 3.0, который помогает быстро, эффективно и оптимально обрабатывать двоичные данные. Он входит в пакет *flash.utils*. Его имя

абсолютно соответствует его сущности: это массив байтов. Но его методы обеспечивают возможность манипулировать и организовывать доступ к его данным намного быстрее, чем обычный массив ActionScript.

Прежде всего необходимо создать пустой *ByteArray*, как показано в следующем примере:

```
var spectrum:ByteArray = new ByteArray();
```

Затем этот *ByteArray* передается в метод *SoundMixer.computeSpectrum( )*, который принимает мгновенную копию звука во время воспроизведения и обсчитывает текущую звуковую волну левого и правого каналов. Полученные данные он разбивает на 256 значений для каждого канала, каждое в диапазоне от –1,0 до 1,0. Эти значения сохраняются в переданный пустой *ByteArray*.

После этого данные можно использовать как угодно. Просто перебираем 512 элементов массива *ByteArray*, вызывая метод *getFloat( )*. Первые 256 вызовов дают значения для левого канала, следующие 256 – для правого.

В следующем примере эти значения, а также объект *BitmapData* и метод *setPixel32( )* служат для создания визуального представления звуковых волн обоих каналов:

```
package {
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.media.SoundMixer;
    import flash.net.URLRequest;
    import flash.utils.ByteArray;

    public class Spectrum extends Sprite {
        private var _sound:Sound;
        private var _channel:SoundChannel;
        private var _spectrumGraph:BitmapData;

        public function Spectrum() {
            // Создаем растровое изображение для отображения
            // спектрального состава звука
            _spectrumGraph = new BitmapData(256, 60,
                                            true,
                                            0x00000000);
            var bitmap:Bitmap = new Bitmap(_spectrumGraph);
            addChild(bitmap);
            bitmap.x = 10;
            bitmap.y = 10;

            addEventListener(Event.ENTER_FRAME, onEnterFrame);
```

```
            _sound = new Sound(new URLRequest("song.mp3"));
            _channel = _sound.play();
        }

        public function onEnterFrame(event:Event):void
        {
            // Создаем байтовый массив и заполняем его данными
            var spectrum:ByteArray = new ByteArray();
            SoundMixer.computeSpectrum(spectrum);

            // Очищаем растровое изображение
            _spectrumGraph.fillRect(_spectrumGraph.rect,
                                    0x00000000);

            // Создаем визуальное представление левого канала
            for(var i:int=0;i<256;i++) {
                _spectrumGraph.setPixel32(i,
                               20 + spectrum.readFloat() * 20,
                               0xffffffff);
            }

            // Создаем визуальное представление правого канала
            for(var i:int=0;i<256;i++) {
                _spectrumGraph.setPixel32(i,
                               40 + spectrum.readFloat() * 20,
                               0xffffffff);
            }
        }
    }
}
```

См. также

Рецепт 15.14.

15.14. Изменение громкости звука или баланса стереодорожек

Задача

Требуется изменить громкость звука, делая его громче или тише, или настроить стереобаланс (баланс между левым и правым каналами) звука.

Решение

Создайте новый объект *SoundTransform*, определяя требуемую громкость воспроизведения звука или размер и направление панорамирования. Передайте его как значение свойства *soundTransform* объекта *SoundChannel*, ассоциированного с соответствующим звуковым файлом.

Обсуждение

В предыдущих версиях ActionScript можно было задавать громкость и баланс звука непосредственно в самом объекте *Sound*. Теперь эти аспекты воспроизведения звука вынесены в класс *SoundTransform*.

Объект *SoundChannel* имеет свойство *soundTransform*, которое является экземпляром класса *SoundTransform*. Чтобы изменить громкость или баланс звука, создаем новый объект *SoundTransform* и задаем необходимые значения. Далее определяем этот новый объект как значение свойства *soundTransform* объекта *SoundChannel*. Например, в следующем фрагменте кода устанавливается громкость звука 50%:

```
var _sound:Sound = new Sound(new URLRequest("song.mp3"));
var channel:SoundChannel = _sound.play();
var transform:SoundTransform = new SoundTransform();
transform.volume = .5;
channel.soundTransform = transform;
```

Нетрудно понять, что звук меняется в диапазоне от 0,0 (тишина) до 1,0 (максимальная громкость). Аналогично задается стереобаланс звука:

```
var channel:SoundChannel = _sound.play();
var transform:SoundTransform = new SoundTransform();
transform.pan = -1.0;
channel.soundTransform = transform;
```

Таким образом, звук будет звучать только в левом динамике, поскольку в данном случае диапазон допустимых значений от –1,0 (только левый) до 1,0 (только правый).

Кроме того, громкость и значения баланса можно передавать прямо в конструктор *SoundTransform*:

```
var channel:SoundChannel = _sound.play();
var transform:SoundTransform = new SoundTransform(.5, -1.0);
channel.soundTransform = transform;
```

Первый параметр – громкость, второй – стереобаланс.

См. также

Рецепт 15.13.

15.15. Создание приложения работы со звуком

Задача

Требуется создать полнофункциональное приложение работы со звуком, такое как проигрыватель, поддерживающий потоковое воспроизведение MP3-файлов.

Решение

Вооружившись остальными рецептами данной главы, реализуйте воспроизведение потокового звука и добавьте необходимые возможности.

Обсуждение

Данный рецепт собирает многие техники, рассмотренные в этой главе, в одном приложении. Это приложение, конечно, не показывает все возможности, но дает представление о том, как их использовать и сочетать.

```
package {
    import flash.display.Sprite;
    import flash.display.Stage;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.text.TextField;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.events.TimerEvent;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import flash.net.URLRequest;
    import flash.text.TextFormat;
    import flash.utils.Timer;

    public class CookBookPlayer extends Sprite {
        private var _channel:SoundChannel;
        private var _displayText:TextField;
        private var _sound:Sound;
        private var _panControl:PanControl;
        private var _playing:Boolean = false;
        private var _playPauseButton:Sprite;
        private var _position:int = 0;
        private var _spectrumGraph:SpectrumGraph;
        private var _volumeControl:VolumeControl;

        public function CookBookPlayer() {
            // Выравнивание сцены
            stage.scaleMode =
                      flash.display.StageScaleMode.NO_SCALE;
            stage.align = flash.display.StageAlign.TOP_LEFT;

            // Добавляем обработчик события таймера
            var timer:Timer = new Timer(20);
            timer.addEventListener(TimerEvent.TIMER, onTimer);
            timer.start();
            _playing = true;

            // Отображаем текстовое поле
            _displayText = new TextField();
```

```
        addChild(_displayText);
        _displayText.x = 10;
        _displayText.y = 17;
        _displayText.width = 256;
        _displayText.height = 14;

        // Создаем звуковой объект
        _sound = new Sound(new URLRequest("http://www.rightactionscript.com
        samplefiles/sample.mp3"));
        _sound.addEventListener(Event.ID3, onID3);
        _channel = _sound.play();

        // Создаем растровое изображение
        // для отображения спектра звуковых частот
        _spectrumGraph = new SpectrumGraph();
        _spectrumGraph.x = 10;
        _spectrumGraph.y = 33;
        addChild(_spectrumGraph);

        // Создаем кнопки Play и Pause
        _playPauseButton = new PlayButton();
        _playPauseButton.x = 10;
        _playPauseButton.y = 68;
        addChild(_playPauseButton);
        _playPauseButton.addEventListener(MouseEvent.MOUSE_UP,
                                          onPlayPause);

        // Создаем элементы управления звука и стереобаланса
        _volumeControl = new VolumeControl();
        _volumeControl.x = 45;
        _volumeControl.y = 68;
        addChild(_volumeControl);
        _volumeControl.addEventListener(Event.CHANGE,
                                        onTransform);

        _panControl = new PanControl();
        _panControl.x = 164;
        _panControl.y = 68;
        addChild(_panControl);
        _panControl.addEventListener(Event.CHANGE,
                                     onTransform);
    }

    public function onTransform(event:Event):void
    {
        // Получаем данные звука и стереобаланса от элементов
        // управления и применяем к новому объекту SoundTransform
        _channel.soundTransform = new SoundTransform(
                                      _volumeControl.volume,
                                      _panControl.pan);
    }

    public function onPlayPause(event:MouseEvent):void
    {
```

```
        // Если воспроизводится, останавливаем
        // и записываем положение головки воспроизведения
        if(_playing) {
            _position = _channel.position;
            _channel.stop();
        }
        else {
            // Или возобновить воспроизведение с заданной точки
            _channel = _sound.play(_position);
        }
        _playing = !_playing;
    }

    public function onID3(event:Event):void {
        // Выводим в текстовое поле выбранные теги id3
        _displayText.text = _sound.id3.artist + " : " +
                            _sound.id3.songName;
        _displayText.setTextFormat(
                    new TextFormat("_typewriter", 8, 0));
    }

    public function onTimer(event:TimerEvent):void {
        var barWidth:int = 256;
        var barHeight:int = 5;

        var loaded:int = _sound.bytesLoaded;
        var total:int = _sound.bytesTotal;

        var length:int = _sound.length;
        var position:int = _channel.position;

        // Отрисовываем фоновую полосу
        graphics.clear();
        graphics.beginFill(0xFFFFFF);
        graphics.drawRect(10, 10, barWidth, barHeight);
        graphics.endFill();

        if(total > 0) {
            // Загруженная часть звукового файла
            // в процентном выражении
            var percentBuffered:Number = loaded / total;

            // Отрисовываем полосу, показывающую,
            // какая часть звукового файла загружена
            graphics.beginFill(0xCCCCCC);
            graphics.drawRect(10, 10,
                              barWidth * percentBuffered,
                              barHeight);
            graphics.endFill();

            // Корректируем длительность звукового файла
            length /= percentBuffered;

            // Воспроизведенная часть звукового файла
            // в процентном выражении
```

```
                var percentPlayed:Number = position / length;
                // Отрисовываем полосу, представляющую
                // какая часть звукового файла воспроизведена
                graphics.beginFill(0x666666);
                graphics.drawRect(10, 10,
                                  barWidth * percentPlayed,
                                  barHeight);
                graphics.endFill();

                _spectrumGraph.update();
            }

        }
    }
}

// "вспомогательные классы"
// (Эти объявления находятся вне пакета, но они доступны
// классам в том же файле)
import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.display.Sprite;
import flash.events.Event;
import flash.events.MouseEvent;
import flash.filters.DropShadowFilter;
import flash.geom.Rectangle;
import flash.media.SoundMixer;
import flash.utils.ByteArray;

class PlayButton extends Sprite {
    public function PlayButton() {
        // Отрисовываем кнопки Play/Pause
        graphics.beginFill(0xcccccc);
        graphics.drawRoundRect(0, 0, 20, 16, 4, 4);
        graphics.endFill();
        graphics.beginFill(0x333333);
        graphics.moveTo(4, 4);
        graphics.lineTo(8, 8);
        graphics.lineTo(4, 12);
        graphics.lineTo(4, 4);
        graphics.drawRect(10, 4, 2, 8);
        graphics.drawRect(14, 4, 2, 8);
        graphics.endFill();

    }
}
class SpectrumGraph extends Sprite {
    private var _spectrumBMP:BitmapData;

    public function SpectrumGraph()
    {
        // Растровое изображение для представления
        // спектра звуковых частот
```

```
            _spectrumBMP = new BitmapData(256, 30,
                                          true, 0x00000000);
            var bitmap:Bitmap = new Bitmap(_spectrumBMP);
            bitmap.filters = [new DropShadowFilter(3, 45, 0, 1,
                                                   3, 2, .3, 3)];
            addChild(bitmap);
        }

        public function update():void
        {
            // Получаем данные спектрального состава
            var spectrum:ByteArray = new ByteArray();
            SoundMixer.computeSpectrum(spectrum);

            // Отрисовываем их в растровом изображении
            _spectrumBMP.fillRect(_spectrumBMP.rect, 0xff666666);
            _spectrumBMP.fillRect(new Rectangle(1, 1, 254, 28),
                                  0x00000000);
            for(var i:int=0;i<256;i++) {
                _spectrumBMP.setPixel32(i,
                                        10 + spectrum.readFloat() * 10,
                                        0xff000000);
            }
            for(var i:int=0;i<256;i++) {
                _spectrumBMP.setPixel32(i,
                                        20 + spectrum.readFloat() * 10,
                                        0xff000000);
            }
        }
    }

    class VolumeControl extends Sprite {
        public var volume:Number = 1.0;

        public function VolumeControl()
        {
            addEventListener(MouseEvent.CLICK, onClick);
            draw();
        }

        public function onClick(event:MouseEvent):void
        {
            // По щелчку полосы пользователем задаем громкость
            volume = event.localX / 100;
            draw();
            dispatchEvent(new Event(Event.CHANGE));
        }

        private function draw():void {
            // Отрисовываем полосу и текущий уровень громкости
            graphics.beginFill(0xcccccc);
            graphics.drawRect(0, 0, 102, 16);
            graphics.endFill();
```

```
            graphics.beginFill(0x000000);
            graphics.drawRect(volume * 100, 0, 2, 16);
        }
    }

    class PanControl extends Sprite {
        public var pan:Number = 0;

        public function PanControl()
        {
            addEventListener(MouseEvent.CLICK, onClick);
            draw();
        }

        public function onClick(event:MouseEvent):void
        {
            // По щелчку полосы пользователем задаем стереобаланс
            pan = event.localX / 50 - 1;
            draw();
            dispatchEvent(new Event(Event.CHANGE));
        }

        private function draw():void {
            // Отрисовываем полосу и текущий стереобаланс
            graphics.beginFill(0xcccccc);
            graphics.drawRect(0, 0, 102, 16);
            graphics.endFill();

            graphics.beginFill(0x000000);
            graphics.drawRect(50 + pan * 50, 0, 2, 16);
        }
    }
```

16

Работа с видеоданными

16.0. Введение

Flash Player может воспроизводить видеоданные. И хотя их нетрудно встроить в файл *.swf*, как правило, видеоданные Flash сохраняют в видеофайлах с расширением *.flv* и загружают во Flash Player во время выполнения с помощью ActionScript. Благодаря этому мы получаем файлы *.swf* меньшего размера, более гибкое управление содержимым и больший контроль над загрузкой и воспроизведением видеосодержимого.

Видеоданные Flash, загружаемые из файлов *.flv*, бывают двух типов: с прогрессивной загрузкой и потоковые. Потоковая передача *.flv* требует наличия потокового сервера, такого как Flash Media Server. Прогрессивная загрузка, напротив, не требует никакого дополнительного программного обеспечения. Однако в целом код ActionScript для потоковой передачи и прогрессивной загрузки видеоданных идентичен. В рецептах данной главы обсуждается работа с видеоданными с прогрессивной загрузки и особое внимание уделяется работе с *.flv*-файлами.

16.1. Загрузка и воспроизведение видеоданных

Задача

Требуется загрузить и воспроизвести видеоданные Flash с прогрессивной загрузкой.

Решение

Задача решается с помощью объекта *NetStream*, позволяющего загрузить и воспроизвести видеоданные, и объекта *Video*, отвечающего за отображение их на экране.

Обсуждение

В ActionScript 3.0 загрузка и воспроизведение видеоданных Flash обеспечивается совместной работой нескольких классов. Для загрузки видеоданных и управления воспроизведением предназначен объект *NetStream* (сетевой поток). Однако класс *NetStream* занимается только перемещением данных, а их рендеринг ему недоступен. Для этого нужен объект *Video*. В него передается объект *NetStream*, данные которого и используются для рендеринга видео на экране.

В конструктор *NetStream* надо передать объект *NetConnection* (сетевое соединение). Объект *NetConnection* определяет источник данных, которые обрабатываются объектом *NetStream*. Если потоковые видеоданные поступают с Flash Communication Server или Flash Media Server, то объект *NetConnection* указывает на сервер. Однако для видеосодержимого с прогрессивной загрузкой объект *NetConnection* использует *null* в качестве строки соединения. В следующем фрагменте кода создается объект *NetConnection*. Он инициализируется для работы с видеоданными с прогрессивной загрузкой. Обратите внимание, в примере предполагается, что класс *flash.net.NetConnection* импортирован:

```
var videoConnection:NetConnection = new NetConnection();
videoConnection.connect(null);
```

После создания объекта *NetConnection* и передачи в его метод *connect( )* значения `null` мы создаем объект *NetStream*, передавая в его конструктор ссылку на объект *NetConnection*. В следующем фрагменте кода создается объект *NetStream* (предполагается, что *flash.net.NetStream* импортирован):

```
var videoStream:NetStream = new NetStream(videoConnection);
```

Создав объект *NetStream*, добавляем объект *Video* и ассоциируем *NetStream* с *Video*.

Класс *flash.media.Video* – это отображаемый объект, т. е. будучи созданным, он должен быть добавлен в список отображения. В следующем фрагменте кода создается новый объект *Video* и добавляется в список отображения:

```
var video:Video = new Video();
addChild(video);
```

Объект *NetStream* можно ассоциировать с объектом *Video* посредством метода *attachNetStream( )* объекта *Video*, в который в качестве параметра передается ссылка на объект *NetStream*:

```
video.attachNetStream(videoStream);
```

Если объект *NetStream* подключен к объекту *Video*, то объект *Video* визуализирует все видеоданные, управляемые объектом *NetStream*.

Все, что обсуждалось до сих пор, требуется для инициализации всех необходимых объектов и связей. Однако до сих пор ничто не указывает Flash Player, что надо загрузить данные и начать воспроизведение. Оба этих действия осуществляются вызовом одного простого метода – метода *play( )* класса *NetStream*. Он загружает и начинает воспроизведение видеоданных Flash соответственно передаваемому в него параметру. В качестве параметра выступает абсолютный или относительный URL. Следующий фрагмент кода указывает Flash Player загрузить и начать воспроизведение видеофайла Flash *example.flv*, который находится в одном каталоге с вызывающим *.swf*:

```
videoStream.play("example.flv");
```

Если файл *.flv* находится в одном домене с вызывающим его *.swf*, вызов *play( )* осуществляется без участия системы безопасности Flash Player. Однако если FLV располагается в другом домене, необходим файл политики безопасности, который разрешит SWF загружать и воспроизводить FLV.

Воспроизведение видеоданных начинается автоматически, как только заполняется буфер. Буферизацией и загрузкой можно управлять, что обсуждается в рецепте 16.7.

Если файл FLV содержит встроенные метаданные (а большинство кодеров встраивают метаданные), придется также обрабатывать событие метаданных, как описывается в рецепте 16.4. Кроме того, если в файле FLV есть сигнальные точки, придется обрабатывать события сигнальных точек, как описывается в рецепте 16.8. Если FLV содержит метаданные и/или сигнальные точки и соответствующие события не обрабатываются, то Flash Player формирует ошибки.

См. также

Рецепты 16.4, 16.7 и 16.8.

16.2. Управление звуком видеофайла

Задача

Требуется регулировать громкость и стереобаланс звуковой части видеофайла.

Решение

Обратитесь к свойству `soundTransform` объекта *NetStream*.

Обсуждение

Если в видеофайле Flash есть звуковая дорожка, звук воспроизводится автоматически вместе с видео. Если требуется регулировать громкость или менять стереобаланс звука, через свойство `soundTransform` объекта

NetStream можно получить ссылку на объект *SoundTransform*, ассоциированный с аудиоданными. Более подробно о работе с объектами *SoundTransform* рассказывается в рецепте 15.14.

См. также

Рецепт 15.14.

16.3. Как узнать время воспроизведения

Задача

Требуется получить текущее время воспроизведения видеозаписи.

Решение

Это делается при помощи свойства `time` объекта *NetStream*.

Обсуждение

Свойство `time` класса *NetStream* доступно только для чтения и возвращает время воспроизведения видеозаписи (в секундах от начала). Следующий код отображает время воспроизведения видеозаписи в текстовом поле:

```
textField.text = videoStream.time + " seconds";
```

Обратите внимание, что значение не округляется до ближайшего целого числа, т. е. можно получить и значение 1, и значение 5,235. Для представления длительности в целых числах можно применять методы *Math.round( )*, *Math.floor( )* или *Math.ceil( )*.

Свойство `time` доступно только для чтения, поэтому оно не может применяться для управления воспроизведением. При необходимости задействуется метод *seek( )*, как описывается в рецепте 16.5.

См. также

Рецепты 4.3 и 16.5

16.4. Как получить длительность видеозаписи

Задача

Требуется получить общую продолжительность (длительность) видеозаписи.

Решение

Это делается при помощи метода обратного вызова *onMetaData( )* и значения метаданных `duration`.

Обсуждение

В классе *NetStream нет* свойства, которое возвращало бы общую продолжительность видеофайла Flash. Однако в большинстве случаев это значение можно получить из самого файла FLV. В FLV-файлах могут содержаться метаданные, и практически все видеокодеры включают значение `duration` (длительность), в котором хранится длительность видеозаписи в секундах (и долях секунд) и которое можно прочитать средствами ActionScript.

При загрузке файла FLV объект *NetStream* автоматически вызывает метод обратного вызова *onMetaData( )* сразу после считывания метаданных. Модель обратного вызова отличается от стандартной модели событий, используемой большинством API ActionScript 3.0. При работе с событиями посредством метода *addEventListener( )* добавляются слушатели этих событий. Однако чтобы обрабатывать события метаданных, для объекта необходимо определить метод *onMetaData( )* и затем присвоить этот объект как значение свойства `client` (клиент) объекта *NetStream*. В метод *onMetaData( )* в качестве параметра автоматически передается ассоциативный массив типа *Object*, содержащий свойства и значения, соответствующие свойствам метаданных, которые были прочитаны из FLV. В следующем примере метод *trace( )* служит для отображения значений метаданных `duration` видеоролика:

```
var client:Object = new Object();
client.onMetaData = function(metadata:Object):void {
    trace(metadata.duration);
};
videoStream.client = client;
```

В практическом смысле предыдущий пример не представляет особой ценности. Обычно ссылка на функцию присваивается свойству `onMetaData` объекта *NetStream*. В этом случае вызов функции осуществляется в соответствующем контексте. В следующем примере класс загружает, воспроизводит видеозапись и отображает время воспроизведения и длительность видеозаписи:

```
package {

    import flash.text.TextField;
    import flash.media.Video;
    import flash.net.NetConnection;
    import flash.net.NetStream;
    import flash.events.NetStatusEvent;
    import flash.text.TextFieldAutoSize;
    import flash.display.Sprite;
    import flash.events.Event;

    public class Example extends Sprite {

        private var _stream:NetStream;
        private var _video:Video;
```

```
        private var _playbackTime:TextField;
        private var _duration:uint;

        public function Example() {
            _video = new Video(160, 120);
            _playbackTime = new TextField();
            _playbackTime.autoSize = TextFieldAutoSize.LEFT;
            _playbackTime.y = 120;
            _playbackTime.text = "test";
            _duration = 0;
            var connection:NetConnection = new NetConnection();
            connection.connect(null);
            _stream = new NetStream(connection);
            _stream.play("video.flv");
            var client:Object = new Object();
            client.onMetaData = onMetaData;
            _stream.client = client;
            _video.attachNetStream(_stream);
            addChild(_video);
            addChild(_playbackTime);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onMetaData(data:Object):void {
            _duration = data.duration;
        }

        private function onEnterFrame(event:Event):void {
            if(_duration > 0 && _stream.time > 0) {
                _playbackTime.text = Math.round(_stream.time) + " / " +
                Math.round(_duration);
            }
        }

    }
}
```

См. также

Рецепт 16.8.

16.5. Управление временем воспроизведения

Задача

Требуется управлять воспроизведением видеозаписи.

Решение

Задача решается с помощью метода *seek( )* объекта *NetStream*.

Обсуждение

Интуитивно кажется, что время воспроизведения можно задать, присваивая значение свойству `time` объекта *NetStream*, который управляет видеозаписью, но это ничего не даст. Из рецепта 16.3 мы знаем, что свойство `time` доступно только для чтения. Поэтому задействуется метод *seek( )* (установить). Он принимает один обязательный параметр, определяющий количество секунд от начала видеозаписи, на которое необходимо перенести воспроизведение. Следующий пример перед началом воспроизведения смещает головку воспроизведения на пять секунд от начала видеозаписи:

```
videoStream.seek(5);
```

При использовании метода *seek( )* для видеозаписей с прогрессивной загрузкой переносить головку можно только в пределах загруженной части видеозаписи. Например, если загружена половина 60-секундной видеозаписи, можно перенести головку на 10 секунд, но нельзя установить ее на 50 секунду видеозаписи.

В случае применения метода *seek( )* Flash Player переходит к ключевому кадру, ближайшему к указанному значению. Кодировка видеозаписей осуществляется с разными частотами ключевых кадров. Чем выше частота ключевых кадров, тем выше требования к пропускной способности. Однако высокие частоты ключевых кадров обеспечивают большую точность при установке головки воспроизведения.

См. также

Рецепт 16.12.

16.6. Масштабирование видеоизображения

Задача

Требуется изменить размеры видеоизображения.

Решение

Обратитесь к свойствам `width` и `height` объекта *Video*. Если требуется воспроизводить видеозапись с теми размерами, с которыми она была кодирована, задайте значения `videoWidth` и `videoHeight`.

Обсуждение

Размеры видеоизображения при воспроизведении определяются свойствами `width` и `height` объекта *Video*. При создании *Video* можно задать исходные ширину и высоту изображения. Так, следующий фрагмент кода создает объект *Video* размером 160×120 пикселов:

```
var video:Video = new Video(160, 120);
```

Однако посредством свойств `width` и `height` размеры можно менять и после создания объекта. Следующий код удваивает размеры объекта *Video*, созданного в предыдущем примере:

```
video.width = 320;
video.height = 240;
```

Класс *Video* также определяет два свойства, доступных только для чтения, `videoWidth` и `videoHeight`, возвращающих высоту и ширину, с которыми видеозапись была кодирована. Значения свойств `videoWidth` и `videoHeight` могут служить для задания свойств `width` и `height` объекта *Video*:

```
video.width = video.videoWidth;
video.height = video.videoHeight;
```

Однако заметим, что свойства `videoWidth` и `videoHeight` определены корректно только после начала загрузки FLV. Следовательно, если требуется задать ширину и высоту объекта *Video* на основании значений, используемых при кодировании, необходимо подождать до тех пор, пока свойства `videoWidth` и `videoHeight` не будут определены корректно. В этом случае можно задействовать слушатель события `netStatus` (статус сети):

```
videoStream.addEventListener(NetStatusEvent.NET_STATUS, onStatus);
```

Теперь определяем слушателя с помощью выражения `if`, проверяющего, что значения `videoWidth` и `videoHeight` больше 0, а также что высота и ширина не равны `videoWidth` и `videoHeight`. Поскольку `videoWidth` и `videoHeight` задаются одновременно, достаточно проверить только одно из этих свойств:

```
private function onStatus(event:NetStatusEvent):void {
    if(_video.videoWidth > 0 && _video.width != _video.videoWidth) {
        _video.width = _video.videoWidth;
        _video.height = _video.videoHeight;
    }
}
```

16.7. Контроль и управление буферизацией и загрузкой

Задача

Требуется управлять процессом буферизации видеоданных и отслеживать процесс загрузки.

Решение

С помощью свойства `bufferTime` задайте размер буфера (в секундах), который должен быть загружен, прежде чем начнется воспроизведение, и затем отслеживайте процесс загрузки посредством свойств `bytesLoaded` и `bytesTotal`.

Обсуждение

По умолчанию воспроизведение начинается сразу после загрузки в буфер 0,1 (одной десятой) секунды видеозаписи. Такое значение при медленном или неустойчивом соединении чревато паузами при воспроизведении или частыми сбросами буфера, ведущими к необходимости его повторного заполнения. Чем больше размер буфера, тем выше гарантия непрерывного воспроизведения. Размер буфера можно задать посредством свойства `bufferTime` (время буфера). Следующий код устанавливает буфер размером 10 секунд:

```
videoStream.bufferTime = 10;
```

Организация одного буфера – не всегда оптимальное решение. В большинстве случаев такой подход работоспособен, но не обеспечивает наивысшего качества воспроизведения. Чтобы гарантировать всем пользователям отличное качество, надо немного потрудиться. Прежде всего надо определить пропускную способность (см. рецепт 16.14). После этого возможны два основных варианта развития событий:

- Если допускается, чтобы пользователи с более медленными сетевыми соединениями просматривали копии видеозаписи более низкого качества, можно кодировать несколько версий видеозаписи с разным битрейтом и выбирать соответствующую версию исходя из пропускной способности соединения.
- Если же требуется, чтобы все пользователи просматривали один и тот же видеофайл (кодированный с одинаковым битрейтом), то можно задавать размер буфера исходя из установленной пропускной способности. Это означает, что для пользователей с более медленными соединениями размер буфера будет увеличен, и им придется дольше ожидать начала воспроизведение видеозаписи.

Отображение процесса буферизации видеофайла можно организовать, обратившись к свойствам `bufferTime` и `bufferLength` (длительность буфера). Свойство `bufferLength` доступно только для чтения и возвращает число, показывающее, сколько секунд видеозаписи загружено в буфер. Отношение `bufferLength/bufferTime` отражает степень заполнения буфера.

Свойство `bufferLength` сообщает, сколько секунд видеозаписи загружено в буфер, но не дает информации о том, какая часть всего видеофайла загружена. Определить это помогут свойства `bytesLoaded` (сколько байт загружено) и `bytesTotal` (общее количество байт).

См. также

Рецепты 16.1 и 16.14.

16.8. Обнаружение сигнальных точек

Задача

Требуется найти сигнальные точки, кодированные в FLV.

Решение

Это делается при помощи метода обратного вызова *onCuePoint( )*.

Обсуждение

Формат FLV позволяет встраивать в файл сигнальные точки, которые помогают, в частности:

- Организовывать заголовки
- Синхронизировать анимацию и видеоизображение
- Вести статистику воспроизведения видеозаписей

Сигнальные точки встраиваются в файл FLV при помощи одного из кодировщиков FLV, например Flash Video Exporter (который поставляется с Flash Professional), On2 Flix (*http:// www.on2.com*) или Sorenson Squeeze (*http://www.sorensonmedia.com*). Когда при воспроизведении видеозаписи встречается сигнальная точка, Flash Player получает уведомление. Как и при работе с метаданными, в этом случае задействуется не стандартная модель событий ActionScript 3.0, а модель обратного вызова. По достижении сигнальной точки Flash Player вызывает метод обратного вызова *onCuePoint( )*. Как и *onMetaData( )*, метод *onCuePoint( )* должен быть определен для объекта, который задается как свойство `client` объекта *NetStream*. Метод *onCuePoint( )* принимает один параметр – объект, имеющий следующие свойства:

`name`

Имя, присвоенное сигнальной точке при кодировании.

`time`

Местоположение сигнальной точки на временной шкале видеозаписи (в секундах).

`type`

«event» или «navigation» в зависимости от того, что выбрано при кодировании.

`parameters`

Ассоциативный массив, содержащий пары ключ/значение, заданные при кодировании FLV.

Следующее иллюстрирует один из способов создания клиентского объекта, задание ссылки на *onCuePoint* и назначение клиентского объекта объекту *NetStream*:

```
var client:Object = new Object();
client.onCuePoint = onCuePoint;
videoStream.client = client;
```

Затем соответствующим образом определяем метод:

```
private function onCuePoint(cuePoint:Object):void {
    trace(cuePoint.name + " " + cuePoint.time);
}
```

См. также

Рецепт 16.4.

16.9. Применение фильтров к видеоизображению

Задача

Требуется применить к видеоизображению фильтры (размытие, цвет, текстура и т. д.).

Решение

Присвойте массив всех фильтров, которые требуется применить, свойству `filters` объекта *Video*.

Обсуждение

Класс *Video* наследует свойство `filters` от класса *DisplayObject*. Это означает, что к объектам *Video* можно применять фильтры так же, как и к отображаемым объектам (см. рецепт 10.5).

См. также

Рецепт 10.5.

16.10. Приостановка и возобновление воспроизведения видеозаписи

Задача

Требуется приостановить или возобновить воспроизведение видеозаписи.

Решение

Задача решается посредством методов *pause( )*, *resume( )* и *togglePause( )* класса *NetStream*.

Обсуждение

Метод *pause( )* класса *NetStream* позволяет приостанавливать воспроизведение видеозаписи. Если видеозапись в данный момент не проигрывается, ничего не произойдет.

```
videoStream.pause();
```

Метод *resume( )* возобновляет воспроизведение, если оно было до этого прервано, и ничего не делает, если проигрывание видеозаписи уже идет:

```
videoStream.resume();
```

Метод *togglePause( )* переключает состояние видеоизаписи (приостанавливает, если воспроизводится, и возобновляет, если приостановлена):

```
videoStream.togglePause();
```

Очень часто посредством метода *play( )* ошибочно пытаются возобновлять воспроизведение приостановленной видеозаписи. Метод *play( )* предназначен только для загрузки и воспроизведения видеозаписи с начала.

16.11. Остановка воспроизведения

Задача

Требуется остановить воспроизведение и загрузку видеоданных.

Решение

Обратитесь к методу *close( )* класса *NetStream*.

Обсуждение

Если требуется остановить воспроизведение видеозаписи, применяют метод *close( )* объекта *NetStream*, который управляет видеоданными. Метод *pause( )* приостанавливает воспроизведение, которое можно возобновить (обычно по нажатию кнопки пользовательского интерфейса), но данные FLV продолжают загружаться, даже когда воспроизведение приостановлено. Для того чтобы прекратить загрузку видеоданных, используется метод *close( )*:

```
videoStream.close();
```

При вызове метода *close( )* Flash Player удаляет данные FLV из проигрывателя. Если запрос к этим видеоданным поступает снова, приходится начинать загрузку заново. Однако броузер может кэшировать FLV, и на последующие запросы к одному и тому же FLV будет возвращаться кэшированная версия. В этом случае воспроизведение при последующих запросах будет начинаться быстрее. Но если FLV кэширован, то пользователь увидит кэшированную версию, а не обновленную.

Чтобы пользователь гарантированно просматривал самую последнюю версию FLV, при запросе к URL FLV добавляется уникальная строка, как показано в следующем примере:

```
videoStream.play("video.flv?uniqueIndex=" + (new Date()).getTime());
```

Если по соображениям защиты авторских прав кэширование видеоданных нежелательно, то видеофайлы с прогрессивной загрузкой не годятся. В таких случаях осуществляется потоковая передача видеозаписи с применением такой технологии, как Flash Media Server, которая выходит за рамки рассмотрения данной книги.

16.12. Перемотка видеозаписи

Задача

Необходимо «перемотать» видеозапись (перемещать головку воспроизведения вперед или назад во время воспроизведения видеозаписи).

Решение

Используйте элемент управления «ползунок» в сочетании с методом *seek( )*.

Обсуждение

Обычно воспроизведением видеозаписи управляют путем «перемотки» с помощью управляющего ползунка. Перемотку можно реализовать, создав ползунок, управляющий объектом *NetStream* посредством метода *seek( )*.

Следующий пример кода иллюстрирует один из способов создания такого элемента управления:

```
package com.oreilly.as3cb.components {

    import flash.display.Sprite;
    import flash.net.NetStream;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.geom.Rectangle;

    public class VideoScrubber extends Sprite {

        private var _thumb:Sprite;
        private var _track:Sprite;
        private var _stream:NetStream;
        private var _duration:Number;
        private var _scrubbing:Boolean;

        public function VideoScrubber(stream:NetStream, duration:Number) {
            _stream = stream;
            _duration = duration;
            _track = new Sprite();
```

```
            _track.graphics.lineStyle();
            _track.graphics.drawRect(0, -2.5, 100, 5);
            addChild(_track);
            _thumb = new Sprite();
            _thumb.graphics.lineStyle();
            _thumb.graphics.beginFill(0xFFFFFF);
            _thumb.graphics.drawRect(-5, -5, 10, 10);
            _thumb.graphics.endFill();
            addChild(_thumb);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            _thumb.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            _thumb.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
        }

        private function onMouseDown(event:MouseEvent):void {
            _scrubbing = true;
            var rectangle:Rectangle = new Rectangle(0, 0, _track.width, 0);
            _thumb.startDrag(false, rectangle);
        }

        private function onMouseUp(event:MouseEvent):void {
            _scrubbing = false;
            _thumb.stopDrag();
        }

        private function onEnterFrame(event:Event):void {
            if(_duration > 0) {
                if(_scrubbing) {
                    _stream.seek(_duration * _thumb.x / _track.width);
                }
                else {
                    _thumb.x = _stream.time / _duration * _track.width;
                }
            }
        }

    }
}
```

В этом примере конструктор принимает параметр *NetStream* и параметр, определяющий продолжительность видеозаписи, и создает элементы ползунка. Когда пользователь щелкает по нему, вызывается метод *startDrag( )* и переменной `_scrubbing` присваивается значение `true`. Пока значение `_scrubbing` не изменится, код вызывает метод *seek( )* потока, передавая в него значение времени воспроизведения, соответствующее положению ползунка.

Обратите внимание, что предыдущий пример не отслеживает события *mouseUp*, потому что мышь может находиться вне элемента управления «ползунок», когда пользователь отпускает ее кнопку. Решение этой задачи довольно сложное (и поэтому не обсуждается в данном рецепте). Подробно об этом рассказано в рецепте 6.9.

См. также

Рецепт 6.9.

16.13. Очистка области видеоизображения

Задача

Требуется очистить область видеоизображения.

Решение

Вызывайте метод *clear( )* объекта *Video*.

Обсуждение

При закрытии объекта *NetStream* не происходит автоматической очистки области видеоизображения. Последний кадр остается видимым в объекте *Video* до тех пор, пока или не будет удален объект *Video*, или не будет вызван метод *clear( )*. Следующий пример очищает объект *Video* путем вызова метода *clear( )*:

```
video.clear();
```

Удалить объект *Video* из списка отображения можно методом *removeChild( )*.

```
removeChild(video);
```

16.14. Определение пропускной способности соединения пользователя

Задача

Требуется оптимизировать воспроизведение видеоданных. Для этого необходимо определить пропускную способность сети пользователя.

Решение

Загрузите файл изображения, измерьте время загрузки и вычислите скорость сетевого соединения пользователя.

Обсуждение

К сожалению, во Flash Player нет встроенной системы определения пропускной способности. И поскольку фактическая пропускная способность зависит от многих факторов (таких как загруженность сети, интерференция в беспроводных сетях, использование сети другими приложениями, выполняющимися в той же системе, и т. д.), невозможно точно предвидеть, какой будет пропускная способность пользователя в следующие 10 минут, час или любой другой промежуток вре-

мени. Однако можно измерить фактическую пропускную способность соединения пользователя за некоторый период времени и на основании этого значения определить *возможную* пропускную способность в ближайшем будущем.

Чтобы измерить пропускную способность, необходимо загрузить во Flash Player файл (несжатый), например файл JPEG. ActionScript позволяет определить общее количество загруженных байтов и время, которое заняла их загрузка. Имея эти два значения, вычисляем среднюю скорость загрузки данных в единицу времени. При работе с видеоданными пропускная способность обычно определяется как скорость передачи битов и выражается в килобитах в секунду. В байте 8 бит, а в килобайте 1000 байт, т. е. преобразовать байты в килобиты можно по следующей формуле:

```
kilobits = bytes / 1000 * 8;
```

Соотношение байт/килобайт и бит/килобит зависит от контекста. Если речь идет о хранении данных на диске, отношение составляет 1/1024, тогда как для передачи данных оно – 1/1000.

Чем больше загружаемый файл, тем более точным будет измерение. Например, если пользователь загружает файл размером 10 килобайт, запрос может попасть в пик загрузки сети и полученный результат будет намного меньшим, чем реальная средняя пропускная способность. С другой стороны, не надо заставлять пользователя загружать слишком большие файлы, поскольку тогда измерение пропускной способности может занять слишком много времени.

Одно из решений состоит в том, чтобы взять файл среднего размера (примерно 50–100 килобайт) и провести тестирование несколько раз. В этом случае, если первые два тестирования дают близкие результаты, можно предположить, что такие же значения будут получены и при последующих проверках, и завершить тестирование. Если результаты первых двух тестирований сильно отличаются, можно проводить дополнительные тесты. При многократной загрузке одного и того же файла необходимо гарантировать, что каждый раз используется уникальный URL, и Flash Player не извлекает файл из кэша броузера. Следующий код иллюстрирует такое тестирование пропускной способности:

```
package com.oreilly.as3cb.util {

    import flash.events.EventDispatcher;
    import flash.net.URLLoader;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.utils.getTimer;

    public class BandwidthTest extends EventDispatcher {

        private var _downloadCount:uint;
        private var _bandwidthTests:Array;
```

```
private var _detectedBandwidth:Number;
private var _startTime:uint;

public function get detectedBandwidth():Number {
    return _detectedBandwidth;
}

public function BandwidthTest() {
    _downloadCount = 0;
    _bandwidthTests = new Array();
}

// Начинаем тестирование пропускной способности.
public function test():void {

    // Используем URLLoader для загрузки данных.
    var loader:URLLoader = new URLLoader();

    // Используем URL с уникальной строкой запроса, чтобы
    // гарантировать, что данные загружаются не из кэша броузера.
    var request:URLRequest =
                new URLRequest("bandwidthtestimage.jpg?unique="
    + (new Date()).getTime());
    loader.load(request);
    loader.addEventListener(Event.OPEN, onStart);
    loader.addEventListener(Event.COMPLETE, onLoad);
}

// Когда начинается загрузка файла,
// получаем текущее значение времени.
private function onStart(event:Event):void {
    _startTime = getTimer();
}

private function onLoad(event:Event):void {

    // Время загрузки – это разность между временем окончания
    // загрузки файла и временем начала загрузки. Затем делим
    // результат на 1000, чтобы преобразовать
    // миллисекунды в секунды.
    var downloadTime:Number = (getTimer() - _startTime) / 1000;
    _downloadCount++;

    // Преобразовываем байты в килобиты.
    var kilobits:Number = event.target.bytesTotal / 1000 * 8;

    // Делим килобиты на время загрузки.
    var kbps:Number = kilobits / downloadTime;

    // Добавляем полученное значение в массив.
    _bandwidthTests.push(kbps);

    if(_downloadCount == 1) {
        // Если тестирование выполнено всего один
        // раз, выполняем его второй раз.
        test();
    }
```

```
            else if(_downloadCount == 2) {
                // Если выполнено два тестирования, определяем разницу
                // между полученными значениями. Если она невелика
                // (в данном примере, менее 50 kbps), формируем
                // событие complete. Если нет, продолжаем тестирование.
                if(Math.abs(_bandwidthTests[0] - _bandwidthTests[1]) < 50) {
                    dispatchCompleteEvent();
                }
                else {
                    test();
                }
            }
            else {
                // После третьей проверки формируем событие complete.
                dispatchCompleteEvent();
            }
        }
        private function dispatchCompleteEvent():void {
            // Определяем среднюю пропускную способность.
            _detectedBandwidth = 0;
            var i:uint;
            for(i = 0; i < _bandwidthTests.length; i++) {
                _detectedBandwidth += _bandwidthTests[i];
            }
            _detectedBandwidth /= _downloadCount;
            // Формируем событие complete.
            dispatchEvent(new Event(Event.COMPLETE
        }
    }
}
```

Экземпляры описанного выше класса могут использоваться для выполнения проверки пропускной способности. Для этого создается новый экземпляр класса, добавляется слушатель и выполняется метод *test( )*.

```
var bandwidthTester:BandwidthTest = new BandwidthTest();
bandwidthTester.addEventListener(Event.COMPLETE, onBandwidthTest);
bandwidthTester.test();
```

Когда событие `complete` произошло, полученную пропускную способность можно извлечь из свойства `detectedBandwidth` (определенная пропускная способность).

```
private function onBandwidthTest(event:Event):void {
  trace(event.target.detectedBandwidth);
}
```

См. также

Рецепты 19.1 (могут использоваться API, загружающие двоичные данные, такие как файлы изображений) и 19.4.

17

Хранение данных

17.0. Введение

Многие данные, с которыми работает ActionScript, во время воспроизведения ролика хранятся в памяти. После закрытия ролика память высвобождается, и данные становятся недоступными. А если требуется сохранять данные между отдельными запусками ролика? Или использовать их в разных роликах, выполняющихся на одном клиенте? Чтобы уметь решать эти задачи, рассмотрим способы хранения данных вне Flash Player.

В ActionScript реализовать хранение данных на клиентской машине позволяет класс *SharedObject* (совместно используемый объект). Существует два типа таких объектов: локальные и удаленные. В данной главе рассмотрены только *локальные совместно используемые объекты* (*Local Shared Objects*, LSO).

Локальные совместно используемые объекты подобны файлам cookies броузера в том, что они хранятся на клиентской машине. LSO служат тем же целям, для которых традиционно применяются куки, например для сохранения данных пользователя на веб-сайте, чтобы ему не приходилось регистрироваться вручную при каждом посещении. Однако LSO – это более мощное средство, чем cookies, потому что по умолчанию имеют бесконечный срок действия, могут хранить больше данных, не передаются между клиентом и сервером и могут хранить собственные типы данных ActionScript. В отличие от *удаленных совместно используемых объектов* (*Remote Shared Objects*, RSO), для применения LSO не требуется никакого дополнительного программного обеспечения ни на клиенте, ни на сервере.

17.1. Создание и открытие локального совместно используемого объекта

Задача

Требуется сохранять информацию между выполнениями файла *.swf*.

Решение

Задача решается при помощи LSO.

Обсуждение

Начиная эту главу, мы уже сказали, что LSO для Flash – это то же самое, что cookies для веб-броузеров, но только намного больше. Некоторые разработчики называют их «супер-cookies», т. к. эти объекты позволяют сохранять большие объемы данных и хранить и извлекать многие встроенные типы данных ActionScript (а также объекты специальных классов). В общем LSO – это *Flash-куки* (*Flash cookies*).

По умолчанию максимальный размер LSO составляет 100 Кбайт. Однако это вовсе не значит, что ролику будет доступна вся эта память, поскольку пользователи полностью контролируют размер LSO посредством Settings Manager (Диспетчер настроек) Flash Player и могут ограничивать его по своему усмотрению. Объекты LSO хранятся на клиентском компьютере в виде двоичных файлов с расширением *.sol*. Ролики Flash в рамках одного домена могут писать и читать файлы *.sol* средствами класса *flash.net.SharedObject* ActionScript.

Созданный файл *.sol* размещается в каталоге данных приложения Flash Player. Для пользователей, работающих в Microsoft Windows, он будет располагаться в примерно таком каталоге: *C:\Documents and Settings\[имяпользователя]\Application Data\Macromedia\Flash Player\#SharedObjects\[случайное имя каталога]*. В Mac OS X каталог будет каким-то таким: */Users/[имя пользователя]/Library/Preferences/Macromedia/Flash Player/#SharedObject/[случайное имя каталога]*. *Случайное имя каталога* необходимо для обеспечения безопасности, ведь файлы *.sol* хранятся в стандартных каталогах. Злонамеренный ролик *.swf* может угадать имя и месторасположение совместно используемого объекта определенного веб-сайта и, таким образом, получит возможность загружать LSO из файловой системы и получать доступ к данным, которые обычно для него недоступны. Внося в путь к файлу *.sol* элемент случайности, мы практически сводим к нулю возможность угадать этот путь.

Статический метод *getLocal( )* является средством как создания LSO, так и открытия его для чтения. Этот метод принимает минимум один обязательный параметр: строку, определяющую имя совместно используемого объекта, который должен быть создан или открыт:

```
var example:SharedObject = SharedObject.getLocal( "example" );
```

Сначала метод *getLocal( )*пытается по заданному имени найти на клиентском компьютере существующий LSO с расширением *.sol*. Если файл не обнаружен, Flash Player создает новый LSO с таким именем. В любом случае открывается существующий или новый LSO. Метод *getLocal( )* возвращает экземпляр *SharedObject*. Он используется вместо оператора *new* для создания новых локальных совместно используемых объектов.

См. также

Рецепт 17.2, содержащий важную информацию о том, как сохранять данные в локальный совместно используемый объект. В рецепте 17.3 подробно рассказывается, как извлекать данные из локального совместно используемого объекта. Рецепт 17.4 рассказывает о методе *SharedObject.flush( )*, применяемом для сохранения данных в совместно используемый объект вручную. Рецепт 17.5 обсуждает совместное использование данных различными роликами. В рецепте 17.8 подробно рассматривается вкладка Local Storage (Локальное хранилище) диалогового окна Flash Player Settings, управляющая распределением дискового пространства.

17.2. Запись данных в LSO

Задача

Требуется добавить данные в LSO.

Решение

Добавляйте значения как свойства объекта `data` совместно используемого объекта.

Обсуждение

Объекты LSO имеют специальное встроенное свойство `data` (данные), представляющее собой объект. В него и добавляется любая информация, которую требуется сохранить в совместно используемом объекте:

```
// Сохраняем значение свойства username
// в совместно используемом объекте example.
example.data.username = "Деррон";
```

В отличие от предыдущих версий ActionScript, теперь нельзя создавать свойства непосредственно в объекте LSO. При попытке сохранить данные в самом экземпляре *SharedObject*, а не в свойстве `data` возникает ошибка компиляции. Это предотвращает потенциально ошибочные попытки сохранения данных:

```
example.variable = "Это приведет к ошибке компиляции.";
```

Кроме того, нельзя задавать значение прямо свойству `data`:

```
example.data = "Это опять приведет к ошибке компиляции.";
```

Правильный подход состоит в том, чтобы задать значение новому свойству свойства `data`:

```
example.data.variable = "Вот это правильный способ сохранения данных.";
```

В свойстве `data` совместно используемого объекта могут сохраняться некоторые собственные типы данных ActionScript:

```
example.data.exampleArray = new Array( "a", "b", "c" );
example.data.exampleDate = new Date();
example.data.exampleObject = { key1: "example", key2: -12, key3: null };
```

Однако в свойство `data` объекта LSO нельзя сохранять визуальные элементы (клипы, спрайты, кнопки или текстовые поля) или совместно используемые объекты. Это ограничение связано с невозможностью корректного хранения данных между сеансами в описанных случаях.

17.3. Сохранение локального совместно используемого объекта

Задача

Требуется сохранить данные в LSO на клиентском компьютере.

Решение

Это делается при помощи метода *SharedObject.flush( )*.

Обсуждение

Flash автоматически пытается сохранить данные LSO на диск, когда ролик выгружается из проигрывателя (например, при закрытии проигрывателя), когда экземпляр *SharedObject* подвергается уничтожению в результате сборки мусора (что происходит, когда больше нет ни одной ссылки на этот экземпляр) или при вызове метода *SharedObject.clear( )*. Однако не следует полагаться на автоматическое сохранение, поскольку есть несколько причин, которые могут помешать успешной записи данных на диск. Поэтому необходимо явно записывать данные на диск. Для этого применяется метод *SharedObject.flush( )*:

```
var flushResult:String = example.flush();
```

Метод *flush( )* (сбросить) принимает необязательный параметр, определяющий минимальный размер дискового пространства в байтах, необходимый для записи совместно используемого объекта. Значение по умолчанию равно 0, т. е. для записи локального совместно используемого объекта на диск задействуется лишь минимально необходимый объем памяти.

При вызове метод *flush( )* пытается записать данные на клиентский компьютер. Возможны три результата вызова *flush( )*:

- Если пользователь запрещает сохранение LSO для домена или если Flash Player по какой-то причине не удается сохранить данные, данные не сохраняются и метод формирует ошибку – *Error*.
- Если для сохранения данных LSO требуется меньше дискового пространства, чем заданно в локальных настройках для домена, данные записываются на диск, и метод возвращает *SharedObjectFlushStatus.FLUSHED*, что является признаком успешного вызова. Если во *flush( )* указано, что должен использоваться минимально необходимый объем дискового пространства, для обеспечения успешного сохранения данных выделенный объем памяти должен быть больше или равен этому объему.
- Если необходимый для сохранения объекта LSO объем дискового пространства не выделен, пользователь получает предупреждение о том, что необходимо предоставить достаточно памяти или отказаться от сохранения данных. Когда это происходит, метод возвращает *SharedObjectFlushStatus.PENDING*. Если пользователь решает предоставить доступ, автоматически выделяется дополнительный объем памяти и данные сохраняются.

В третьем случае, когда метод *flush( )* возвращает константу *SharedObjectFlushStatus.PENDING*, можно предпринять дополнительный шаг, чтобы выяснить, предоставляет или запрещает пользователь доступ для сохранения данных. Когда пользователь выбирает в автоматически появляющемся диалоге один из предлагаемых вариантов, формируется событие `netStatus`, а разработчик уже должен определить обработчик события для соответствующей обработки результатов. При вызове обработчика события в него передается параметр типа *flash. events.NetStatusEvent*. Проверяем строковое свойство `info.code`, чтобы определить, предоставил ли пользователь доступ (значение *SharedObject.Flush.Success*) или запретил его (значение *SharedObject.Flush.Failed*).

Вот пример вызова метода *flush( )* для явного сохранения данных и последующей обработки возможных результатов:

```
var example:SharedObject = SharedObject.getLocal( "example" );
example.data.someData = "значение";
try {
  var flushResult:String = example.flush();
  // Если операция сохранения не завершена, добавляем
  // обработчик события netStatus для определения решения
  // пользователя (разрешает он или запрещает доступ).
  // В противном случае просто проверяем результат.
  if ( flushResult == SharedObjectFlushStatus.PENDING ) {
    // Добавляем обработчик события netStatus, чтобы иметь
    // возможность проверить, предоставил ли пользователь
    // достаточно дискового пространства для сохранения
    // совместно используемого объекта. Вызываем метод
```

```
        // onStatus при возникновении события netStatus.
        example.addEventListener( NetStatusEvent.NET_STATUS, onStatus );

      } else if ( flushResult == SharedObjectFlushStatus.FLUSHED ) {
        // Операция сохранения прошла успешно.
        // Размещаем здесь любой код, который выполняется
        // после успешного сохранения данных.
      }
    } catch ( e:Error ) {
      // Это означает, что в локальных настройках пользователя
      // указано 'Never.' Если важно, чтобы данные были сохранены,
      // здесь можно предупредить пользователя. Также если вы хотите
      // упростить пользователю задачу по изменению его настроек,
      // можно открыть вкладку local storage диалогового окна Settings
      // проигрывателя с помощью следующего кода:
      // Security.showSettings( SecurityPanel.LOCAL_STORAGE );.
    }

    // Определяем функцию onStatus() для обработки события
    // status совместно используемого объекта, формируемого,
    // когда пользователь делает выбор в диалоге, который
    // выводится, когда flush() возвращает "pending."
    function onStatus( event:NetStatusEvent ):void {
      if ( event.info.code == "SharedObject.Flush.Success" ) {
        // Если значение свойства event.info.code -
        // "SharedObject.Flush.Success", это означает,
        // что пользователь предоставил доступ. Здесь можно
        // разместить любой код, который должен быть выполнен,
        // когда пользователь предоставляет доступ.
      } else if ( event.info.code == "SharedObject.Flush.Failed" ) {
        // Если значение свойства event.info.code -
        //  "SharedObject.Flush.Failed", это означает,
        // что пользователь запретил доступ. Здесь размещается
        // любой код, который должен быть выполнен,
        // когда пользователь запрещает доступ.
      }

      // Теперь удаляем слушателя события, потому что оно
      // может быть сформировано только один раз
      example.removeEventListener( NetStatusEvent.NET_STATUS, onStatus );
    };
```

Если заранее известно, что размер совместно используемого объекта будет увеличиваться с каждым сеансом, то при его создании разумно запросить больший объем локальной памяти. В противном случае пользователь будет постоянно получать запрос о предоставлении дополнительного дискового пространства. Задание большего объема памяти поможет избежать этих повторяющихся запросов разрешения пользователя на сохранение постоянно увеличивающего количества данных. В этом случае в метод *flush( )* должен быть передан параметр, определяющий минимальный объем памяти, который должен быть выделен для сохранения совместно используемого объекта:

```
// Запрашиваем для совместно используемого объекта 500 Кбайт памяти.
var flashResult:String = example.flush( 500 * 1024 );
```

См. также

Рецепт 17.8.

17.4. Чтение данных из совместно используемого объекта

Задача

Требуется читать значения, которые были записаны в LSO.

Решение

Читайте значения из свойств, хранящихся в свойстве `data` совместно используемого объекта.

Обсуждение

Читать значения из клиентского совместно используемого объекта несложно. Все постоянные значения хранятся в свойстве `data`, поэтому достаточно читать значения свойств объекта `data`:

```
// Читаем значение exampleProperty совместно используемого
// объекта example и выводим их в окне Output.
trace( example.data.exampleProperty );
```

Сочетая чтение и запись данных, можно определить, просматривал ли пользователь этот файл *.swf* ранее.

```
// Создаем совместно используемый объект
// и сохраняем в него некоторые данные
var example:SharedObject = SharedObject.getLocal( "example" );

if ( example.data.previouslyViewed ) {
  // Пользователь уже просматривал файл .swf, вероятно,
  // пропустим в данном случае вводное окно справки.
} else {
  // Пользователь первый раз просматривает файл .swf,
  // потому что previouslyViewed еще не задано значение true.
  // Задаем previouslyViewed значение true, чтобы при следующем
  // выполнении этого кода было видно, что пользователь уже был здесь.
  example.data.previouslyViewed = true;
  example.flush();
}
```

17.5. Удаление данных из совместно используемого объекта

Задача

Требуется удалить свойства из совместно используемого объекта или полностью удалить этот объект с диска.

Решение

Данные из свойства `data` совместно используемого объекта удаляются при помощи оператора *delete*, а для удаления совместно используемого объекта служит метод *clear( )*.

Обсуждение

Удалить данные из совместно используемого объекта просто, но надо все сделать правильно. В ActionScript для удаления сложных переменных, таких как объекты или массивы, им нередко присваивается значение `null` или `undefined`. Однако с совместно используемыми объектами так поступать нельзя:

```
// Пытаемся удалить someVariable из совместно
// используемого объекта example. Это выражение будет
// скомпилировано, но не обеспечит необходимого эффекта.
example.data.someVariable = null;
```

Значения `null` и `undefined` — это допустимые значения для совместно используемого объекта. Поэтому предыдущий код не удаляет `someVariable` из свойства `data`, а просто присваивает `someVariable` значение `null`. Чтобы полностью удалить что-то из совместно используемого объекта, необходимо обратиться к оператору *delete*:

```
// Удаляем someVariable из совместно
// используемого объекта example.
delete example.data.someVariable;
```

Вызывая метод *clear( )*, можно удалить весь совместно используемый объект. После вызова метода *clear( )* физический файл *.sol*, хранящий совместно используемый объект, удаляется с диска. Пример удаления совместно используемого объекта.

```
// Создаем совместно используемый объект
// и сохраняем в него некоторые данные
var example:SharedObject = SharedObject.getLocal( "example" );
example.data.someData = "значение";

// На экран выводится: значение
trace( example.data.someData );

// Удаляем совместно используемый объект с диска
example.clear();
```

```
// На экран выводится: undefined
trace( example.data.someData );
```

После вызова метода *clear( )* совместно используемого объекта ссылка на него остается действительной. Это означает, что с данным объектом можно работать как с только что созданным. Добавленные в свойство `data` значения автоматически сохраняются, т. е. наблюдается обычное поведение совместно используемого объекта. Фактически метод *clear( )* полностью очищает совместно используемый объект, и в случае необходимости с ним можно опять работать.

17.6. Сериализация специальных классов

Задача

Требуется сохранить в LSO экземпляр специального класса.

Решение

Обратитесь к методу *flash.net.registerClassAlias( )* для сохранения информации о типе и добавления экземпляра класса в свойство `data` совместно используемого объекта.

Обсуждение

Для кодирования информации LSO применяется специальный двоичный формат, *Action Message Format* (AMF). При сохранении экземпляра класса в LSO он кодируется как универсальный объект, содержащий свойства. Таким образом, при попытке восстановить экземпляр из совместно используемого объекта он не может быть восстановлен как экземпляр класса, потому что при кодировании не была сохранена информация о типе.

Метод *registerClassAlias( )* (зарегистрировать псевдоним класса) пакета *flash.net* позволяет обойти это ограничение. Данный метод очень похож на *Object.registerClass( )* из ActionScript 1.0 и 2.0, изъятый из ActionScript 3.0 и замененный *flash.net.registerClassAlias( )*.

Метод *registerClassAlias( )* принимает два параметра. Первый – *строковый псевдоним*, представляющий класс. В качестве псевдонима может выступать любое строковое значение, но лучше всего, чтобы это было полное имя класса. Например, если имеется класс `Person` (человек) в пакете `model` (модель), для обеспечения ясности и во избежание неоднозначности должен использоваться строковый псевдоним `model.Person`. Этот подход также обеспечивает невозможность дублирования строковых псевдонимов, поскольку не может существовать двух классов с одинаковыми полными именами. Второй параметр – *ссылка* на класс, который требуется сопоставить с псевдонимом класса.

```
registerClassAlias( "somePackage.ExampleClass", ExampleClass );
```

Эта уловка заставляет LSO сохранить информацию о классе, создавшем его. Поэтому при извлечении данных из совместно используемого объекта Flash Player будет знать, что это за объект.

Псевдоним специального класса должен быть зарегистрирован во всех роликах, извлекающих данные специального объекта из совместно используемого объекта.

Следующий код представляет собой полную реализацию сохранения экземпляра класса в LSO. Сначала создается специальный класс, подлежащий сериализации.

```
// Создаем класс Person в пакете model
package model {

  public class Person {

    private var _firstName:String;
    private var _age:int;

    public function Person(firstName:String, age:int) {
      _firstName = firstName;
      _age = age;
    }

    public function toString():String {
      return _firstName + _age + " лет";
    }

  }

}
```

Затем создается основной класс для чтения и записи данных.

```
package {
  import flash.net.registerClassAlias;
  import flash.net.SharedObject;
  import model.Person;

  public class Example {

    public function Example() {
      // Сопоставляем "model.Person" с классом Person
      registerClassAlias( "model.Person", Person );

      // Создаем совместно используемый объект
      // и сохраняем в нем экземпляр Person
      var example:SharedObject = SharedObject.getLocal( "example" );

      // Проверяем, не сохранен ли экземпляр этого класса
      if ( example.data.person == undefined ) {
        trace( "экземпляр person сохраняется впервые" );
        var person:Person = new Person("Деррон", 24);

        // Записываем экземпляр класса в локальный
        // совместно используемый объект
```

```
            example.data.person = person;
        } else {
          trace( "экземпляр person уже сохранен, используем
                  сохраненные значения" );
        }
        /* При каждом выполнении этого кода на экран выводится следующее:
        Деррон 24 лет
        */
        trace( example.data.person.toString() );
    }
```

По поводу этого кода надо иметь в виду, что метод *registerClassAlias( )* необходимо вызвать *раньше*, чем метод *SharedObject.getLocal( )*. Чтобы обеспечить корректное чтение совместно используемого объекта, надо создать псевдоним класса *до* восстановления совместно используемого объекта. Создание совместно используемого объекта без предварительного создания псевдонима приводит к невозможности восстановления экземпляра `Person`: он восстанавливается как универсальный объект, а не как экземпляр класса.

17.7. Совместное использование данных Flash-приложениями

Задача

Требуется обеспечить двум роликам одного домена доступ к одному и тому же LSO.

Решение

При создании и открытии LSO задайте параметр локального пути.

Обсуждение

По умолчанию LSO сохраняются на клиентском компьютере в каталоге, который уникально определяется доменом, путем и именем файла *.swf*. Его можно получить, вызывая метод *getLocal( )*. Это предупреждает конфликты имен между LSO из разных доменов или даже разных роликов одного домена. Например, в системе, работающей под Windows XP, если ролик *movie.swf*, размещенный в *http://www.person13.com/ascb*, создает LSO с именем `example`, он сохраняется в таком каталоге:

```
C:\Documents and Settings\[имя пользователя]\Application
Data\Macromedia\Flash Player\#SharedObjects\ [случайное имя
каталога]\person13.com\ascb\movie.swf\example.sol
```

Имя файла *.swf* включено в путь к файлу LSO. Это обеспечивает уникальность имен LSO, создаваемых разными роликами одного домена. Однако в некоторых случаях требуется, чтобы несколько роликов име-

ли доступ к одному и тому же LSO. Тогда при создании и открытии LSO методом *getLocal( )* необходимо задавать необязательный *параметр локального пути*.

Параметр локального пути (второй параметр, передаваемый в *getLocal( )*) должен быть строкой, которая определяет полный или частичный путь к *.swf*-файлу ролика, создавшего LSO. Например:

```
var example:SharedObject = SharedObject.getLocal( "example", "/" );
```

Если предыдущий код помещен в *movie.swf*, находящийся в *http://www.person13.com/ascb*, то LSO хранится в каталоге:

```
C:\Documents and Settings\[имя пользователя]\Application
Data\Macromedia\Flash Player\#SharedObjects\ [случайное имя
каталога]\person13.com\example.sol
```

Отличие между данным каталогом и представленным ранее в том, что в нем нет информации о ролике и пути. LSO, созданный таким образом, может быть открыт любым другим роликом Flash из того же домена следующей строкой кода на ActionScript:

```
var example:SharedObject = SharedObject.getLocal( "example", "/" );
```

Важно понимать, что ролик может создавать и/или открывать только те LSO, которые находятся в каталогах, входящих в его полный или частичный путь. Чтобы разобраться, рассмотрим пример с двумя роликами Flash: *movieOne.swf* и *movieTwo.swf*. Оба ролика находятся в одном домене (*http://www.person13.com*), но в разных каталогах: *movieOne.swf* – в *http://www.person13.com/ascb/firstGroup*, и *movieTwo.swf* – в *http:// www.person13.com/ascb/secondGroup*. В данном сценарии *movieOne.swf* может создавать и читать LSO с любым из следующих значений локального пути:

```
/
/ascb
/ascb/firstGroup
```

и *movieTwo.swf* может создавать и читать LSO с любым из следующих значений локального пути:

```
/
/ascb
/ascb/secondGroup
```

Следовательно, если требуется, чтобы оба ролика имели доступ к общему LSO, при вызове метода *getLocal( )* необходимо определить один из двух совместно используемых роликами локальных каталогов (/ или */ascb*).

Покажем, как разделить данные между двумя (или более) роликами Flash одного домена, на примере. Если ролики находятся в разных каталогах, то сначала необходимо определить для них общий локальный каталог в иерархиях каталогов. Для начала посмотрим, что произойдет, если не удается задать общий локальный каталог:

1. Создаем новый ролик *.swf*, содержащий следующий код, и называем его *movieA.swf*:

```
var count:SharedObject = SharedObject.getLocal( "count" );

// При первом чтении совместно используемого объекта
// задаем значение по умолчанию 0
if ( count.data.value == undefined ) {
  count.data.value = 0;
} else {
  // При каждом чтении совместно используемого объекта
  // увеличиваем значение счетчика
  count.data.value++;
}

// Создаем текстовое поле для отображения значения счетчика
var output:TextField = new TextField();
output.text = "значение счетчика: " + count.data.value;
addChild( output );
```

2. Создаем новый каталог на вашем компьютере. Назовем его *LSOTest*.
3. Создаем в *LSOTest* два подкаталога. Называем их *movieAPath* и *movieBPath*.
4. Копируем файл *movieA.swf* в оба подкаталога, которые были только что созданы.
5. Меняем имя ролика *movieA.swf* на *movieB.swf* в каталоге *LSOTest/movieBPath*.
6. Несколько раз открываем и закрываем файл *movieA.swf*. При каждой загрузке файла значение счетчика должно увеличиваться на единицу.
7. Несколько раз открываем и закрываем файл *movieB.swf*. При каждой загрузке файла значение счетчика должно увеличиваться на единицу. Однако обратите внимание, что отсчет начинается с 0. Дело в том, что в данных условиях *movieA.swf* и *movieB.swf* работают с разными объектами LSO. И хотя имена у этих объектов одинаковые, сами они находятся в разных каталогах, почему и создаются разные *.sol*-файлы совместно используемых объектов.
8. Чтобы оба ролика работали с одним и тем же совместно используемым объектом, необходимо, чтобы они искали его в одном каталоге. Меняем строку кода, открывающую совместно используемый объект:

```
var count:SharedObject = SharedObject.getLocal( "count", "/" );
```

   В этом случае оба ролика работают с совместно используемым объектом из одного каталога, т.е. с одним и тем же файлом.
9. Переименовываем и копируем обновленный файл *.swf* в соответствующие каталоги согласно предыдущим шагам, помещая *movieA.swf* в каталог *movieAPath* и *movieB.swf* в каталог *movieBPath*.

10. Повторяем шаги 5 и 6. На это раз заметьте, что каждое обновление любого из роликов обусловливает приращение значения счетчика, потому что теперь используется один и тот же объект LSO.

17.8. Управление размерами локально используемых объектов

Задача

Требуется управлять размером дискового пространства, отведенного для совместно используемого объекта.

Решение

Обратитесь к методу *Security.showSettings( )* или посетите страницу Flash Player Settings Manager.

Обсуждение

В рецепте 17.1 говорилось, что размер LSO по умолчанию составляет 100 Кбайт, а в рецепте 17.3 – что с помощью метода *flush( )* можно запрашивать для LSO определенный размер дискового пространства. Если запрашиваемый объем памяти превышает максимально допустимое значение, пользователь получает уведомление с предложением удовлетворить или отклонить запрос. Метод *flash.system.Security.showSettings( )* позволяет открывать диалоговое окно, в котором пользователь может задать максимально допустимый для хранения в LSO объем данных. Следующий фрагмент кода открывает раздел диалогового окна Settings для настроек локального хранилища.

```
// Показываем настройки локального хранилища
Security.showSettings( SecurityPanel.LOCAL_STORAGE );
```

Подобно Settings UI для отдельного домена есть более развернутый Settings Manager, в котором пользователь может менять глобальные настройки. Хотя Settings и Settings Manager предоставляют аналогичную функциональность, это абсолютно разные элементы. Settings UI – это специальная форма во Flash Player, которая открывается методом *showSettings( )* и обеспечивает возможность пользователю управлять настройками конкретного домена. Settings UI можно открыть посредством ActionScript, что иллюстрирует предыдущий фрагмент кода. Это окно также можно открыть, одновременно нажав правую кнопку мыши и клавишу Control во Flash Player и выбрав пункт Settings контекстного меню. Доступ к Settings Manager, напротив, осуществляется через веб-сайт *macromedia.com*. Чтобы открыть Settings Manager в виде Website Storage, отправляемся на *http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager07.html*.

Он может выглядеть как простая веб-страница или снимок экрана, но на самом деле это функциональное приложение Flash, меняющее глобальные настройки Flash Player. Изменение значения посредством Settings Manager меняет поведение Flash Player. Например, если требуется изменить размер по умолчанию LSO (для LSO, которые еще не созданы), отправляемся к странице Global Storage Settings (глобальные настройки хранилища) (*http://www.macromedia.com/support/documentation/en/flashplayer/ help/settings_manager03.html*).

Перемещая ползунок, задаем необходимое значение. После этого максимальный размер LSO по умолчанию изменен.

Запомните, что разработчик не может управлять дисковым пространством, выделенным под LSO, на клиентском компьютере. Максимум, что можно сделать – это запросить определенный объем памяти, объяснить, почему может потребоваться столько памяти, и надеяться, что пользователь предоставит ее. Поэтому не рекомендуется полагаться на LSO в критически важных для приложения вопросах, потому что последнее слово относительно размера (и даже существования) LSO за пользователем.

18

Обмен информацией между роликами

18.0. Введение

Взаимодействие роликов Flash необходимо рассматривать с точки зрения двух возможных сценариев. Первый – взаимодействие двух роликов, выполняющихся на одном клиентском компьютере. Второй – обмен информацией между роликами, выполняющимися на разных компьютерах.

До версии 6 единственным средством обмена информацией между роликами Flash Player, выполняющимися на одном компьютере (функциональность серверной стороны не задействовалась), была функция *fscommand( )*. Она обеспечивала возможность выполнять методы JavaScript и использовать их как связующее звено между роликами. К сожалению, эти методы были и довольно громоздкими, и не всегда надежными (из-за отсутствия поддержки некоторыми броузерами). Кроме того, область их применения ограничивалась роликами Flash, воспроизводившимися в броузерах, окна которых содержали ссылки друг на друга. Однако Flash Player 6 ввел *локальные соединения*, с помощью которых любой ролик Flash может передавать и принимать сообщения от любого другого ролика, выполняющегося на том же компьютере. Преимущества локальных соединений:

- С ними относительно просто работать.
- Они полностью реализованы на ActionScript и применимы к роликам, выполняющимся во Flash Player 6 или более поздних версиях.

Обмен информацией между роликами, воспроизводимыми на разных клиентских компьютерах, реализуется при помощи серверной технологии по одному из трех возможных вариантов:

- Отправка и получение данных разными роликами между собой могут обеспечиваться применением удаленных совместно используе-

мых объектов в сочетании с медиа-сервером Flash Media Server (раньше он назывался Flash Communication Server).

- Может быть установлено сетевое соединение с сервером, которое будет обеспечивать передачу сообщений между всеми подключенными клиентами.
- Обмен информацией не в режиме реального времени можно реализовать посредством последовательного опроса в сочетании с серверной технологией, что обеспечит отправку и получение новой информации с сервера через определенные интервалы времени.

Основное внимание в данной главе уделено реализации взаимодействия роликов, выполняющихся на одном клиентском компьютере, с помощью класса *LocalConnection* (локальное соединение).

См. также

Главу 24, где рассказывается, как при помощи сетевых соединений организовать взаимодействие клиентов, выполняющихся на разных компьютерах. В главе 19 представлена информация об отправке и получении данных, которая может быть полезной в ситуациях, когда не требуется взаимодействие в режиме реального времени.

18.1. Создание локальных соединений

Задача

Требуется передать информацию от одного или более роликов Flash другому ролику Flash, выполняющемуся на том же клиентском компьютере.

Решение

Задача решается при помощи класса *LocalConnection* пакета *flash.net* и вызова функции получающего ролика по сигналу от отправляющего ролика. Используйте метод *LocalConnection.connect( )*, который будет слушать сообщения в получающем ролике, и определите вызываемую функцию. Функцию получающего ролика вызывайте при помощи *LocalConnection.send( )* отправляющего(-их) ролика(-ов). И отправляющий, и получающий ролики для обмена информацией должны использовать канал с одинаковым именем.

Обсуждение

Два или более ролика Flash, выполняющиеся на одном клиентском компьютере, могут обмениваться информацией друг с другом через локальное соединение, созданное с помощью класса *flash.net.LocalConnection*. Поскольку ролики выполняются на одном компьютере, соединение может быть установлено независимо от того, в каких доменах располагаются ролики.

По умолчанию ролики могут обмениваться информацией только с роликами, находящимися с ними в одном домене. Однако их можно конфигурировать на получение информации и от роликов из других доменов. См. рецепт 18.4.

Чтобы обеспечить успешный обмен информацией между несколькими роликами на одном компьютере, необходимо сделать следующее:

1. Сконфигурировать получающий ролик так, чтобы он ожидал соединения.
2. Указать получающему ролику, что он должен делать при получении сообщений.
3. Посылать сообщения из отправляющего(-их) ролика(-ов) в получающий ролик.

Получающий ролик должен слушать сообщения, поступающие по специальному именованному локальному соединению. Чтобы установить этот канал связи, в получающем ролике необходимо создать объект локального соединения и с помощью метода *connect( )* указать ему слушать сообщения, передаваемые по именованному соединению:

```
import flash.net.LocalConnection;

// Создаем объект локального соединения
// в получающем ролике.
var receiver:LocalConnection = new LocalConnection();

// Указываем экземпляру локального соединения слушать
// сообщения, посланные по каналу "_exampleChannel".
receiver.connect( "_exampleChannel" );
```

Лучше всего, как в этом примере, начинать имя канала связи (не объекта локального соединения) с символа подчеркивания (_). Единый стиль именования упрощает обмен информацией между доменами. Если имя соединения начинается с символа подчеркивания, Flash Player не дополняет его информацией о домене. В противном случае строка имени автоматически будет преобразована в формат *домен:имяСоединения*, где домен – это или имя домена, в котором располагается файл *.swf*, или localhost, если файл *.swf* выполняется локально.

Весь обмен информацией через локальное соединение отображается в получающем экземпляре локального соединения. По умолчанию получающий объект локального соединения ищет определенные в экземпляре метод, имя которого совпадает с именем метода, присланного в запросе. Так, если отправляющий ролик посылает сообщение с запросом к методу *example( )*, то получающий объект ищет у себя метод *example( )*. В предыдущих версиях ActionScript в получающем объекте просто была бы создана функция с именем `example`.

```
// В более ранних версиях ActionScript это обусловило бы
// создание функции example, которая могла бы вызываться
```

```
// через локальное соединение.
receiver.example = function ( ) {
  output.text = "сообщение получено";
};
```

Но в ActionScript 3.0 это не сработает. По умолчанию класс *LocalConnection* не допускает создание свойств и методов в экземплярах класса во время выполнения. Отсюда следуют три основных решения:

- Создаем динамический подкласс *LocalConnection* и используем его экземпляры как получающие локальные соединения.
- Создаем подкласс *LocalConnection*, определяющий методы, которые предполагается вызывать из отправляющего *.swf*.
- Перенаправляем запросы в клиентский объект.

Далее рассмотрим каждое из этих решений.

Чтобы класс допускал изменение своих экземпляров во время выполнения, он должен быть объявлен с ключевым словом *dynamic* (динамический). В ActionScript 3.0 для этого проще всего *расширить* класс *LocalConnection* классом *dynamic* и использовать вновь созданный класс как получатель для добавления методов в него. Этот класс может выглядеть так:

```
package {

  import flash.net.LocalConnection;

  // Создаем динамический класс LocationConnection,
  // в который сможем добавлять новые функции.
  dynamic public class DynamicLocalConnection extends LocalConnection {
    // Пустой – нам необходимо только ключевое слово dynamic
  }
}
```

Теперь вместо обычного класса *LocalConnection* используем *DynamicLocalConnection* (динамическое локальное соединение) и создаем получающую функцию, как в предыдущих версиях ActionScript:

```
// Создаем объект локального соединения в получающем ролике.
var receiver:DynamicLocalConnection = new DynamicLocalConnection();

// Указываем экземпляру локального соединения слушать
// сообщения, переданные по каналу "_exampleChannel".
receiver.connect( "_exampleChannel" );

// Поскольку класс DynamicLocalConnection является
// dynamic, в его экземпляре можно создать новую функцию,
// реагирующую на сообщения локального соединения.
receiver.example = function():void {
  trace( "сообщение получено" );
};
```

Преимущество первого решения обусловлено его гибкостью. Класс *DynamicLocalConnection* может обеспечивать любые получающие ло-

кальные соединения в любых проектах. Проблема в том, что эта гибкость достигается отсутствием у класса четко определенных интерфейсов, к чему обычно стремятся в объектно-ориентированном проектировании.

Второе решение требует применения четко определенного API. Создается подкласс *LocalConnection*, определяющий конкретный метод или методы, подлежащие обработке. Например, следующий класс определяет метод *example( )*.

```
package {
  import flash.net.LocalConnection;
  public class ExampleLocalConnection extends LocalConnection {
    public function ExampleLocalConnection() {}
    public function example():void {
      trace("сообщение получено");
    }
  }
}
```

В следующем решении для перенаправления запросов в другой объект, не получающий экземпляр локального соединения, задействуется свойство `client` объекта *LocalConnection*. Клиентский объект должен определять запрашиваемые методы как открытые. Пример:

```
package {
  import flash.net.LocalConnection;
  public class Example {
    private var _localConnection:LocalConnection;
    public function Example() {
      _localConnection = new LocalConnection();
      _localConnection.connect( "_exampleChannel" );
      _localConnection.client = this;
    }
    public function example():void {
      trace("сообщение получено");
    }
  }
}
```

Запрашиваемые методы можно определить как закрытые, если использовать универсальный объект как посредник, что показано в следующем примере:

```
package {
  import flash.net.LocalConnection;
  public class Example {
    private var _localConnection:LocalConnection;
    public function Example() {
      _localConnection = new LocalConnection();
      _localConnection.connect( "_exampleChannel" );
      _localConnection.client = {example: example};
```

```
    }
    private function example():void {
      trace("сообщение получено");
    }
  }
}
```

Итак, мы сконфигурировали получающий *.swf* и можем заняться отправляющим. В отправляющем *.swf* отправка сообщения реализуется посредством метода *LocalConnection.send( )*. Первый параметр метода *send( )* – строка, определяющая имя соединения, по которому будут передаваться сообщения, здесь можно задать несколько разных соединений. Второй параметр метода *send( )* – строка, определяющая имя метода, который должен быть вызван в получающем ролике. При получении сообщения ролик вызывает в получающем объекте локального соединения метод с аналогичным именем.

Следующий пример вызывает удаленный метод `example`:

```
// Посылаем по каналу "_exampleChannel" сообщение, которое
// вызывает метод example() получающего ролика.
var sender:LocalConnection = new LocalConnection();
sender.send( "_exampleChannel", "example" );
```

Класс *LocalConnection* обеспечивает соединение многие-к-одному. То есть для данного канала может существовать несколько отправителей, но только один получатель. Если получатель попытается соединиться с каналом, открытым другим получателем, метод *connect( )* сформирует ошибку и не допустит установления соединения. Чтобы проверить возможность создания принимающего соединения для данного канала, в получателе можно использовать блок *try…catch*, как в следующем примере:

```
var receiver:LocalConnection = new LocalConnection();
try {
  receiver.connect( "_exampleChannel" );
} catch( e:Error ) {
  // Ошибка - не может использовать _exampleChannel,
  // потому что другой получатель уже заявил свои права на него.
}
```

Для обеспечения правильной работы *LocalConnection* и отправляющий, и получающий ролики должны выполняться одновременно на одном компьютере. Для обмена информацией между роликами, выполняющимися на одном компьютере в разное время, используется LSO. Подробно об LSO рассказано в рецептах 17.1 и 17.7.

См. также

Рецепты 17.1, 17.7 и 18.2.

18.2. Отправка данных

Задача

Требуется отправить данные одному или более роликам, выполняющимся на том же компьютере.

Решение

Передайте данные как дополнительные параметры метода *send( )* отправляющего объекта локального соединения.

Обсуждение

Через локальное соединение можно отправлять данные другому ролику, передавая их как дополнительные параметры метода *send( )* отправляющего объекта локального соединения. Метод *send( )* принимает как минимум два обязательных параметра: имя соединения и имя метода, который должен быть вызван в принимающем ролике. Все дополнительные параметры метода *send( )* автоматически передаются как параметры вызова метода получающего ролика. Обратите внимание, что имя вызываемого метода получающего ролика не может совпадать с именами встроенных методов или свойств класса *LocalConnection*, или вызов *send( )* даст сбой. Имена встроенных методов и свойств, которые не могут выступать в качестве параметра имени метода: *send*, *connect*, *close*, *allowDomain*, *allowInsecureDomain*, *client* и *domain*.

Чтобы получающий метод принимал переданные в него параметры, он должен быть описан соответствующим образом. В данном примере создается класс получателя локального соединения, содержащий метод *example( )*, который принимает три параметра: `str`, `num` и `bool`.

```
package {

  import flash.net.LocalConnection;

  public class ExampleReceiver {

    private var _receiver:LocalConnection;

    public function ExampleReceiver() {
      // Создаем экземпляр получателя локального соединения
      // и слушаем канал "_exampleChannel"
      // в ожидании, когда другие ролики вызовут example.
      _receiver = new LocalConnection();
      _receiver.connect( "_exampleChannel" );
      _receiver.client = this;
    }

    public function example( str:String, num:Number, bool:Boolean):void {
      trace( "Параметры: " + str + "\n" + num + "\n" + bool );
    }
  }
}
```

В следующем фрагменте кода отправляется сообщение, вызывающее метод *example( )* в получающем ролике и передающее в него параметры, аналогично вызову *example(«строка», 6.5, true)*:

```
// Отправляем три параметра в метод example() локального соединения
// получающего ролика. Кстати, это строка, число и Булево значение.
var sender:LocalConnection = new LocalConnection();
sender.send( "_exampleChannel", "example", "строка", 6.5, true);
```

Можно передавать не только элементарные типы данных, но и следующие сложные: *Object*, *Array*, *Date*, *TextFormat* и *XML*.

Кроме того, можно передавать объекты специальных типов. Этот процесс очень похож на сохранение и извлечение специальных объектных типов из совместно используемого объекта. Более подробно теория этого вопроса рассматривается в рецепте 17.6, а здесь мы перечислим необходимые действия:

1. Определение специального класса и, в случае необходимости, его импорт в отправляющий и получающий ролики.
2. Регистрация класса в отправляющем и получающем роликах с помощью метода *flash.net.registerClass( )*, что обеспечивает использование одного псевдонима во всех роликах.
3. Передача экземпляра класса по локальному соединению методом *send( )*.

Далее приведен пример кода отправляющего и получающего роликов, который осуществляет передачу данных специального объектного типа от одного ролика другому.

Первым делом определим класс `Person`:

```
// Создаем класс Person в пакете model
package model {

  public class Person {

    private var _firstName:String;
    private var _age:int;

    public function Person(firstName:String, age:int) {
      _firstName = firstName;
      _age = age;
    }

    public function toString():String {
      return _firstName + _age + " лет";
    }

  }

}
```

Код получающего *.swf*:

```
import flash.net.registerClassAlias;
import model.Person;
```

```
registerClassAlias( "model.Person", model.Person );

package {
  import flash.net.LocalConnection;
  public class ExampleReceiver {

    private var _localConnection:LocalConnection;

    public function ExampleReceiver() {
      _localConnection = new LocalConnection();
      _localConnection.connect("_exampleChannel");
      _localConnection.client = this;
    }

    public function example( person:Person ):void {
      trace( person.toString() );
    }

  }
}
```

И код отправляющего *.swf*:

```
package {
  import flash.net.registerClassAlias;
  import model.Person;
  public class ExampleSender {

    public function ExampleSender() {
      registerClassAlias( "model.Person", Person );

      // Создаем экземпляр Person для передачи
      var person:Person = new Person("Деррон", 24);

      // Создаем локальное соединение и отправляем
      // экземпляр Person в получающий ролик.
      var sender:LocalConnection = new LocalConnection();
      sender.send( "_exampleChannel", "example", person );
    }
  }
}
```

Независимо от типа передаваемых данных действуют одни и те же правила обмена информацией по локальному соединению. Оба ролика должны выполняться одновременно на одном компьютере, получающий ролик должен слушать тот же канал, который используется отправляющим роликом. Также следует отметить, что когда специальный тип определен в обоих роликах, отправляющем и получающем, этот тип данных можно передавать в обоих направлениях (хотя для двунаправленного обмена информацией необходимы два разных канала, что рассмотрено в рецепте 18.3).

См. также

Рецепты 17.6 и 18.3.

18.3. Подтверждение получения при передаче сообщений по локальным соединениям

Задача

Отправляющий ролик должен получать подтверждение об успешной доставке сообщения.

Решение

Сконфигурируйте получающий ролик так, чтобы он возвращал подтверждение получения в отправляющий ролик.

Обсуждение

Для подтверждения доставки сообщения получающий ролик может возвращать ответное сообщение соответствующему отправляющему ролику. Из-за природы локальных соединений один канал не годится для двунаправленного обмена информацией. Поэтому необходимо создать отдельные каналы локального соединения для каждого направления связи. Для того чтобы получающий ролик подтверждал получение сообщения, необходимо предпринять следующие шаги:

1. Настроить отправляющий и принимающий ролики, как описано в рецепте 18.1.
2. В вызываемый метод получающего ролика добавить код, отвечающий за отправку уведомления о получении по новому каналу (например, `"_exampleChannelReceipt"`). Для возвращения отправителю уведомления о получении в методе локального соединения можно использовать метод *this.send( )*.
3. В отправляющем ролике вызвать метод *connect( )*, передавая в него имя канала, по которому получающий ролик возвращает уведомление о получении (`"_exampleChannelReceipt"`).
4. В отправляющем ролике создать метод объекта локального соединения для обработки сообщения-уведомления. Имя этого метода должно совпадать с именем метода, вызываемого получающим роликом при отправке уведомления.

Далее представлен пример кода отправляющего и получающего роликов.

Сначала рассмотрим принимающий ролик, использующий класс *DynamicLocalConnection*:

```
// Создаем получающий код, слушающий канал "_exampleChannel".
var receiver:DynamicLocalConnection = new DynamicLocalConnection( );
receiver.connect( "_exampleChannel" );
receiver.example = function():void {
  // Кроме любого другого кода получающего метода
```

```
  // добавляем этот код, который будет возвращать
  // в отправляющий ролик уведомление о получении
  // по каналу "_exampleChannelReceipt". Ключевое слово this
  // ссылается на текущий объект локального соединения.
  this.send( "_exampleChannelReceipt", "receipt" );
};
```

Код отправляющего ролика:

```
// Создаем объект локального соединения
// для передачи по каналу "_exampleChannel".
var sender:DynamicLocalConnection = new DynamicLocalConnection(  );
seder.send( "_exampleChannel", "example" );

// Указываем локальному соединению слушать канал
// "_exampleChannelReceipt" в ожидании подтверждения
// доставки от получающего ролика.
sender.connect( "_exampleChannelReceipt" );

// Определяем метод receipt(), вызываемый
// из получающего ролика.
sender.receipt = function():void {
  output.text = "Получатель передал уведомление о получении";
};
```

Главное, чтобы имя канала, который слушает объект локального соединения (методом *connect( )*), и имя канала, по которому передает сообщения (методом *send( )*) объект локального соединения другого ролика, было одним и тем же.

Чтобы усовершенствовать решение, сообщим получателю имя канала, по которому будет отправлено уведомление. Это пригодится, если получающий ролик принимает данные от нескольких роликов и не всегда знает, откуда поступили данные. В этом случае имя канала передается в получатель следующим образом:

```
sender.send( "_exampleChannel", "example", "_exampleChannelReceipt1" );
```

Отправляющий ролик передает и имя канала, по которому он ожидает получить уведомление о получении. Тогда получающий ролик посылает уведомление по заданному каналу:

```
receiver.example = function( receiptChannelName:String ):void {
  this.send( receiptChannelName, "receipt" );
}
```

Если имя канала для уведомлений передается в получающий ролик, его не надо жестко прописывать в коде получающего ролика, что повышает гибкость решения.

См. также

Рецепт 18.1.

18.4. Локальный обмен данными с роликами других доменов

Задача

Требуется, чтобы ролик принимал по локальному соединению сообщения от роликов из других доменов.

Решение

Обратитесь к методу *allowDomain( )* получающего объекта локального соединения.

Обсуждение

По умолчанию получающие ролики принимают сообщения только от роликов, находящихся с ними в одном домене. Однако метод *allowDomain( )* (разрешить домен) объекта локального соединения позволяет разрешить или запретить обмен сообщениями с любым доменом. Для этого вызывается *allowDomain( )* каждого получающего объекта локального соединения, для которого необходимо определить специальный список допустимых или запрещенных доменов.

Поведение *allowDomain( )* в ActionScript 3.0 отличается от поведения этого метода в предыдущих версиях языка. Ранее это был метод обратного вызова. Он вызывался автоматически при получении сообщения объектом локального соединения. Исходя из возвращаемого методом значения сообщение принималось или отвергалось.

В ActionScript 3.0 метод *allowDomain( )* вызывается явно в экземпляре *LocalConnection*. Он принимает одну или более строк, определяющих домены, которые могут посылать сообщения в экземпляр локального соединения, *allowDomain( )* которого вызван. В данном примере посылать сообщения в текущий ролик могут ролики из *darronschall.com*.

```
var receiver:LocalConnection = new LocalConnection();
receiver.connect( "_exampleChannel" );

// Разрешаем роликам из darronschall.com передавать данные
// по "_exampleChannel" для выполнения кода этого ролика.
receiver.allowDomain( "darronschall.com" );
```

Чтобы данные могли посылать ролики из нескольких доменов, передаем в метод *allowDomain( )* дополнительные строковые параметры, по одному значению для каждого домена:

```
receiver.allowDomain( "macromedia.com", "adobe.com", "google.com" );
```

В *allowDomain( )* могут задаваться два специальных строковых значения. Строка `*` разрешает прием сообщений от всех доменов. Обычно предоставлять возможность передачи информации всем доменам не-

желательно, потому что это позволяет любому ролику вызывать любой метод получающего ролика. Лучше определить группу доверяемых доменов. Чтобы разрешить передачу данных всем роликам, выполняющимся локально, используется строка *localhost*.

Объекты локального соединения также предоставляют соответствующие средства определения домена получающего ролика. В любом экземпляре локального соединения доступно свойство *domain*, посредством которого можно узнать домен ролика. Это свойство доступно только для чтения, т.е. мы можем только проверять его значение. Попытка задать значение *domain* приводит к ошибке компиляции. Чтобы разрешить обмен информацией с роликами одного домена, можно передать *domain* в метод *allowDomain( )*; например:

```
receiver.allowDomain( receiver.domain );
```

Предыдущий пример обеспечивает такое же поведение, как если бы метод *allowDomain( )* не вызывался вообще; он разрешает обмен информацией только между роликами одного домена. То есть метод *allowDomain( )* и свойство *domain* могут применяться для разрешения обмена информацией как в рамках одного домена, так и между разными доменами.

```
receiver.allowDomain( "darronschall.com", receiver.domain );
```

Этот код разрешает принимать по локальному соединению сообщения от роликов домена *darronschall.com*, а также от роликов того же домена, в котором располагается данный ролик.

Метод *allowInsecureDomain( )* (разрешить небезопасный домен) в основном подобен методу *allowDomain( )*. Разница заключается в использовании HTTPS. Если ролик передается по HTTPS, экземпляры локального соединения не будут принимать данные от роликов, поставляемых по HTTP. По умолчанию, если Flash Player не указано обратное посредством метода *allowInsecureDomain( )*, передача информации от HTTP-ролика HTTPS-ролику запрещена, даже если ролики находятся в одном домене.

В общем использовать *allowInsecureDomain( )* нежелательно, потому что он может нивелировать все преимущества безопасности, предлагаемые HTTPS. Нельзя доверять целостности сообщения, полученного по локальному соединению от роликов, передаваемых по HTTP, поскольку ролик мог быть изменен во время передачи. Тем не менее, если без *allowInsecureDomain( )* не обойтись, то его вызывают так же, как *allowDomain( )*.

```
receiver.allowInsecureDomain( "adobe.com" );
```

См. также

Рецепт 3.12.

19

Отправка и загрузка данных

19.0. Введение

Передача значений в ролики и из них часто применяется во Flash для:

- Передачи значений формы в сценарий, выполняющийся на стороне сервера, для сохранения в базу данных.
- Передачи значений из формы электронного письма в сценарий, выполняющийся на стороне сервера, для отправки электронной почты.
- Загрузки значений из текстового файла (выполняется, когда значения подлежат изменению, как при отображении текущей погоды или для обновления страницы ссылок).
- Загрузки из серверного сценария значений, извлеченных из базы данных, например категории в приложении электронной торговли.
- Отправки значений в серверный сценарий для обработки и возвращения значения в ролик Flash, например при регистрации.

При загрузке данных с URL Flash Player может трактовать их как текст, как необработанные двоичные данные или как URL-кодированные переменные. URL-кодированные переменные подчиняются следующим правилам:

- Имя каждой переменной связано со значением посредством знака равенства (перед и после знака равенства пробелы не ставятся).
- Значения переменных, загружаемые в ролики Flash, всегда рассматриваются как строки, поэтому их не надо заключать в кавычки, как в ActionScript. Пример правильной записи: `artist=Picasso`.
- Пары имя/значение отделяются друг от друга символом амперсанда, например `artist=Picasso&type=painting`.
- Пробелы в значениях заменяются знаками плюс (+), а не `%20`, например: `title=The+Old+Guitarist`. (Пробелы и `%20` не запрещены, но совместимость обеспечивают только знаки «плюс».)

- Любой символ, не являющийся числом, буквой из таблицы символов Latin 1 (без диакритических знаков) или пробелом, должен быть преобразован в шестнадцатеричный код. Например, значение «L'Atelier» должно быть кодировано как `L%27Atelier` (`%27` – код апострофа). В табл. 19.1 представлен список часто используемых кодов.

Таблица 19.1. Распространенные коды URL-кодированного формата

Символ	Код	Символ	Код
<space>	%20 или +	.	%2E
!	%21	/	%2F
"	%22	:	%3A
#	%23	;	%3B
$	%24	<	%3C
%	%25	=	%3D
&	%26	>	%3E
'	%27	?	%3F
(	%28	@	%40
)	%29	|	%7C
*	%2A	~	%7E
+	%2B	\	%5C
,	%2C	^	%5E
-	%2D	_	%5F

Вот пример переменных в URL-кодированном формате:

```
artist=Picasso&type=painting&title=Guernica&room=L%27Atelier
```

В ActionScript 3.0 отправка и загрузка данных подверглись существенным изменениям. Класс *LoadVars* был изъят и заменен более надежным *URLLoader* из пакета *flash.net*. *URLLoader* поддерживается новыми действующими лицами ActionScript, в частности классами *URLRequest*, *URLVariables* и *URLStream* (и некоторыми другими). Сочетание этих классов формирует богатый API, предлагающий большую функциональность и гибкость, чем в предыдущих версиях языка.

Однако загрузка данных во Flash Player по-прежнему подчиняется стандартным правилам безопасной изолированной программной среды. То есть загружать данные из файла или сценария можно, только если он находится в том же домене, что и ролик. Как обойти ограничения, накладываемые системой безопасности доменов, рассказано в рецепте 3.12.

19.1. Загрузка переменных из текстового файла

Задача

Требуется загрузить в ролик Flash переменные из внешнего текстового файла.

Решение

Для чтения URL-кодированных данных в ролик Flash используйте метод *URLLoader.load( )* в сочетании с `URLLoaderDataFormat.VARIABLES` как `dataFormat`.

Обсуждение

Для загрузки URL-кодированных данных в ролик Flash из текстового файла всегда применяется метод *URLLoader.load( )*. Это позволяет ролику Flash иметь доступ к часто изменяющимся значениям, а сам ролик Flash при этом не меняется.

Метод *load( )* принимает экземпляр *URLRequest* как обязательный параметр, указывающий на URL текстового файла. URL может быть абсолютным или относительным адресом. Кроме того, *URLLoader* должен быть сконфигурирован соответственно, чтобы интерпретировать текст как URL-кодированные переменные, а не просто текст. Для этого свойству `dataFormat` экземпляра *URLLoader* присваивается значение `URLLoaderDataFormat.VARIABLES`. Вот код, иллюстрирующий загрузку данных из текстового файла:

```
import flash.net.*;

// Сначала необходимо создать объект URLLoader.
var example:URLLoader = new URLLoader();

// Конфигурируем экземпляр, чтобы он рассматривал
// данные как URL-кодированные переменные
example.dataFormat = URLLoaderDataFormat.VARIABLES;

// Этот пример загружает значения из текстового файла
// по абсолютному URL.
example.load( new URLRequest( "http://www.darronschall.com/example.txt" ) );

// Этот пример загружает значения из текстового файла
// по относительному URL, в данном случае из того же
// каталога, где находится файл .swf.
example.load( new URLRequest( "example.txt" ) );
```

А вот пример текстового файла:

```
someText=testing&someNumber=123
```

При вызове метода *load( )* экземпляра *URLLoader* Flash Player пытается загрузить значения по указанному URL в свойство `data` того же эк-

экземпляра *URLLoader*. Когда данные загружены, Flash декодирует их и формирует событие `complete`, сигнализируя о том, что теперь данные доступны в ролике. Можно добавить обработчик этого события и осуществлять какие-то полезные действия после загрузки и декодирования данных.

Если загрузить данные не удается, *URLLoader* формирует другое событие, тип которого зависит от типа сбоя. Поэтому следует слушать не только событие `complete` об удачном завершении загрузки, но и все события, возникающие в случае сбоя.

Вызов *load( )* в случае сбоя может привести к формированию следующих событий:

`httpStatus`

Формируется, если Flash Player может определить код состояния для неудавшегося HTTP-запроса при попытке загрузить данные.

`ioError`

Формируется, если Flash Player сталкивается с неустранимой ошибкой, в результате которой загрузка прерывается.

`securityError`

Формируется, если осуществляется попытка загрузить данные из домена, не входящего в безопасную изолированную программную среду.

Вот пример обработки различных событий:

```
package {
  import flash.events.*;
  import flash.net.*;
  import flash.utils.trace;

  public class Example {

    public function Example() {
      // Создаем экземпляр URLLoader для загрузки данных
      var loader:URLLoader = new URLLoader();

      // Определяем обработчики событий как на случай
      // успешной, так и неудавшейся загрузки
      loader.addEventListener( IOErrorEvent.IO_ERROR, handleIOError );
      loader.addEventListener( HTTPStatusEvent.HTTP_STATUS, handleHttpStatus
);
      loader.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
                               handleSecurityError );
      loader.addEventListener( Event.COMPLETE, handleComplete );

      // Конфигурируем загрузчик на загрузку
      // URL-кодированных переменных
      loader.dataFormat = URLLoaderDataFormat.VARIABLES;

      // Пытаемся загрузить некоторые данные
```

```
        loader.load( new URLRequest( "example.txt" ) );
    }

    function handleIOError( event:IOErrorEvent ):void {
      trace( "Загрузка не удалась: ошибка ввода/вывода: " + event.text );
    }

    function handleHttpStatus( event:HTTPStatusEvent ):void {
      trace( "Загрузка не удалась: HTTP-статус = " + event.status );
    }

    function handleSecurityError( event:SecurityErrorEvent ):void {
      trace( "Загрузка не удалась: ошибка безопасности: " + event.text );
    }

    function handleComplete( event:Event ):void {
      trace( "Данные успешно загружены" );
    }
}
}
```

Если загрузка прошла успешно, Flash Player сохраняет загруженные переменные как свойства `data` экземпляра *URLLoader* и формирует событие `complete`. Ниже приведен пример обработчика события `complete`, который организует доступ к известным свойствам, загруженным из файла. Предполагается, что в текстовом файле, из которого загружаются данные, хранятся переменные *someText* (некоторый текст) и *someNumber* (некоторое число):

```
function handleComplete( event:Event ):void {
  // Приводим свойство target события к типу URLLoader,
  // потому что событие формирует экземпляр URLLoader.
  var loader:URLLoader = URLLoader( event.target );

  // Организуем доступ к загруженной(-ым)
  // переменной(-ым) через имя переменной
  // свойства data экземпляра URLLoader.
  trace( "someText = " + data.someText );
  trace( "someNumber = " + data.someNumber );
}
```

Вот как мог бы выглядеть текстовый файл:

```
someText=ActionScript+3.0+Cookbook&someNumber=3
```

В большинстве случаев имена переменных, загружаемых из внешнего файла или сценария, известны заранее. Например, экземпляр *URLLoader* используется для загрузки ежедневной статьи из серверного сценария или файла. Независимо от содержимого статьи ожидается, что будут загружены переменные `title` (заголовок), `author` (автор) и `articleBody` (текст статьи). Переменные известны, потому что сценарий, выполняющийся на стороне сервера, и программа загрузки написаны одним разработчиком или одной компанией, или потому что автор программы загрузки имеет доступ к документации сценария. Однако ино-

гда загружаемые переменные неизвестны. В этом случае с помощью выражения *for . . . in* можно перебрать все загруженные переменные:

```
function handleComplete( event:Event ):void {
  var loader:URLLoader = URLLoader( event.target );

  // Используем цикл for...in для перечисления
  // всех загруженных переменных
  for ( var property:String in loader.data ) {
    // Имя свойства сохраняется в переменной property,
    // значение извлекается из свойства data.
    trace( property + " = " + loader.data[property] );
  }
}
```

В предыдущих версиях ActionScript загруженные переменные добавлялись непосредственно в экземпляр класса *LoadVars*. Тогда применять цикл, приведенный в данном коде, было проблематично, потому что `property` не всегда была загруженной переменной. Иногда это было обычное свойство экземпляра *LoadVars*, что давало ошибочные результаты при выводе имен загруженных переменных. В ActionScript 3.0 все загруженные переменные хранятся в унифицированном объекте `data` экземпляра. Это делает приведенный выше цикл безопасным, поскольку `property` всегда является именем переменной, которая была загружена, и никогда свойством самого экземпляра *URLLoader*.

19.2. Загрузка переменных из сценария, выполняющегося на стороне сервера

Задача

В ролик Flash требуется загрузить переменные из сценария, выполняющегося на стороне сервера (ColdFusion, Perl, PHP и др.).

Решение

Для чтения URL-кодированных данных, сгенерированных серверным сценарием, в ролике Flash используйте метод *URLLoader.load( )* в сочетании с `URLLoaderDataFormat.VARIABLES` как `dataFormat`.

Обсуждение

Код на ActionScript для загрузки переменных из серверного сценария абсолютно аналогичен используемому для загрузки переменных из текстового файла. При вызове метода *load( )* в него необходимо передать URL сценария, из которого требуется загрузить переменные, и затем при помощи события `complete` обработать результаты, как показано в рецепте 19.1.

Если значения переменных генерируются из базы данных или другого ресурса, доступного только серверу, сценарии, выполняющиеся на сто-

роне сервера, выступают в качестве источника переменных для Flash Player. Выходные данные сценария должны быть исключительно URL-кодированными, начиная с первого символа. Если создается CGI-сценарий на Perl, то результат URL-кодированный, значит, не надо осуществлять никаких дополнительных настроек; например:

```
#!/usr/bin/perl

# В реальном примере это значение извлекалось бы
# из базы данных или другого ресурса сервера.
$someText = "тест";

# Определяем заголовок Content-Type.
print " Content-Type: text/plain\n\n";

# Выводим пару имя-значение.
print "someText=$someText";
```

Однако если для генерирования переменных, загружаемых в ролик, используется страница ColdFusion, необходимо предпринять некоторые шаги, чтобы гарантировать формирование URL-кодированных данных и вывод с первого символа. В противном случае URL-кодированным данным могут предшествовать дополнительные пробельные символы, что приведет к неверному декодированию значений. Вот что надо сделать:

1. Включить тег `<cfsetting enablecfoutputonly="yes">` в начало документа.
2. Убедиться, что все предназначенные для вывода значения заключены в элемент `<cfoutput>`.
3. Поместить весь документ в тег `<cfprocessingdirective suppresswhitespace="yes">`. Это обеспечит отсутствие лишних пробельных символов в выводе.

Например:

```
<cfsetting enablecfoutputonly="yes">
<cfprocessingdirective suppresswhitespace="yes">
  <cfset someText = "тест">
  <cfoutput>someText=#someText#</cfoutput>
</cfprocessingdirective>
```

Если используется PHP, вывод осуществляется с помощью команд `echo` или `print` в рамках директивы обработки:

```
<?php
$someText = "тест";
echo "someText=$someText";
?>
```

В других интерпретируемых языках программирования дополнительные шаги для обеспечения соответствующего вывода могут и не понадобиться. JSP и ASP (.NET и классический) не требуют никаких дора-

боток. Прием, который, по-видимому, работает в большинстве случаев, состоит в том, чтобы окружить каждую пару имя/значение символами амперсанда. Они выполняют роль разделителей, благодаря чему Flash Player может корректно декодировать переменные. Это решение работоспособно независимо от языка и платформы (ASP, JSP, CFML и т. д.). Например, если теги `<cfprocessingdirective>` и `<cfsetting>` на странице ColdFusion нежелательны, то предыдущий пример с ColdFusion можно переписать следующим образом (обратите внимание на символы амперсанда, выделенные жирным шрифтом):

```
<cfset someText="тест">
<cfoutput>&someText=#someText#&</cfoutput>
```

Во всех приведенных примерах серверный сценарий возвращает во Flash Player всего одну переменную, `someText`. В следующем примере код на ActionScript загружает переменную из сценария и отображает ее в ролике:

```
package {

  import flash.events.*;
  import flash.net.*;
  import flash.utils.*;

  public class Example {

  public function Example() {
      // Создаем экземпляр URLLoader, чтобы
      // иметь возможность загружать данные
      var loader:URLLoader = new URLLoader();

      // Определяем обработчик события,
      // который вызывается по завершении загрузки
      loader.addEventListener( Event.COMPLETE, handleComplete );

      // Конфигурируем загрузчик для загрузки
      // URL-кодированных переменных
      loader.dataFormat = URLLoaderDataFormat.VARIABLES;

      // Пытаемся загрузить некоторые данные
      loader.load( new URLRequest( "getSomeText.cfm" ) );
    }

    private function handleComplete( event:Event ):void {
      // Приводим свойство target события к типу URLLoader,
      // потому что событие формирует экземпляр URLLoader.
      var loader:URLLoader = URLLoader( event.target );

      // Организуем доступ к загруженной(-ым) переменной(-ым)
      // по имени переменной свойства data экземпляра URLLoader.
      trace( "someText = " + loader.data.someText );
    }

  }
}
```

Конечно, в загрузке переменных из сценария есть много общего с загрузкой из файла. Например, если имена переменных неизвестны, их вместе со значениями можно перебрать с помощью выражения *for...in*, как показано в рецепте 19.1.

См. также

Рецепт 19.1.

19.3. Загрузка блока текста (включая HTML и XML)

Задача

Требуется загрузить блок текста, возможно, HTML или XML.

Решение

Для чтения текста из файла или чтения возвращаемых значений сценария, выполняющегося на стороне сервера, используйте метод *URLLoader.load( )* в сочетании с `URLLoaderDataFormat.TEXT` как `dataFormat`.

Обсуждение

В ActionScript 3.0 загрузка текста реализуется иначе, чем в ActionScript 1.0 и 2.0. В предыдущих версиях существовало два разных метода обратного вызова, которые можно было определить в экземпляре *LoadVars* для интерпретации загружаемых данных. Метод обратного вызова *onLoad( )* вызывался при обработке URL-кодированных данных. Активация метода *onData( )* происходила, когда загрузка данных уже была закончена, но они еще не подвергались обработке (перед вызовом *onLoad( )*). Используя *onData( )* вместо *onLoad( )*, можно было получать доступ к загруженному тексту до того, как он был обработан как URL-кодированные данные.

Класс *flash.net.URLLoader* ведет себя иначе, чем его предшественник, класс *LoadVars*. *URLLoader* не различает данные, которые только загружены, и данные, которые уже обработаны и декодированы. Теперь по завершении загрузки рассылается событие `complete`, а свойство `dataFormat` определяет, как должны трактоваться загруженные данные. Значение `URLLoaderDataFormat.TEXT` свойства `dataFormat` указывает на то, что данные должны быть интерпретированы как простой текст. В рецептах 19.1 и 19.2 свойству `dataFormat` было задано значение `URLLoaderDataFormat.VARIABLES`, чтобы данные рассматривались как URL-кодированные переменные. По умолчанию экземпляр *URLLoader* обрабатывает загруженные данные как текст.

Представим, что следующий HTML-текст требуется загрузить из внешнего файла *example.html* и вывести в текстовом поле с разметкой HTML:

```
<b>Title:</b> Сборник рецептов ActionScript 3.0<br />
```

```
<b>Authors:</b> Джои Лотт, Деррон Шалл, Кейт Питерс<br />
<b>URL издателя:</b> <a href="http://www.oreilly.com">www.oreilly.com</a>
```

Следующий класс ActionScript загружает текст из файла и отображает его в текстовом поле с разметкой HTML:

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*
  import flash.net.*;

  public class HTMLLoadingExample extends Sprite {
    private var _output:TextField;

    public function HTMLLoadingExample() {
      initializeOutput();

      loadData();
    }

    private function initializeOutput():void {
      _output = new TextField( );
      _output.width = stage.stageWidth;
      _output.height = stage.stageHeight;

      addChild( _output );
    }

    private function loadData():void {
      var loader:URLLoader = new URLLoader();

      // Указываем загрузчику, что файл читать как простой
      // текст – эту строку можно опустить, поскольку
      // dataFormat по умолчанию URLLoaderDataFormat.TEXT.
      loader.dataFormat = URLLoaderDataFormat.TEXT;

      // Регистрируем обработчик события завершения загрузки данных
      loader.addEventListener( Event.COMPLETE, handleComplete );

      // Загружаем HTML-текст из файла example.html
      loader.load( new URLRequest( "example.html" ) );
    }

    private function handleComplete( event:Event ):void {
      var loader:URLLoader = URLLoader( event.target );

      // Присваиваем свойству htmlText текстового поля
      // HTML-текст из example.html. Свойство data
      // экземпляра URLLoader – это содержимое файла.
      _output.htmlText = loader.data;
    }
  }
}
```

Тип свойства `data` **зависит от значения** `dataFormat`**. Если** `dataFormat` **имеет значение** `URLLoaderDataFormat.TEXT`**, свойство** `data` **экземпляра** *URL-*

Loader представляет собой строку (*String*), содержащую текст загруженного файла. Если `dataFormat` присвоено значение `URLLoaderDataFormat.VARIABLES`, то свойство `data` – это объект (*Object*), который содержит имена переменных и их значения. Если `dataFormat` имеет значение `URLLoaderDataFormat.BINARY`, то свойство `data` становится экземпляром *flash.utils.ByteArray*.

См. также

Рецепты 19.1 и 19.2.

19.4. Контроль за процессом загрузки

Задача

Необходимо знать, какая часть данных загружена.

Решение

Подпишитесь на событие `progress` класса *URLLoader*.

Обсуждение

В классе *URLLoader* есть событие `progress` (прогресс), рассылаемое в процессе загрузки данных. Flash Player передает экземпляр *flash.events.ProgressEvent* в обработчик события, что обеспечивает возможность проверять свойства `bytesLoaded` и `bytesTotal` события. Свойство `bytesLoaded` содержит количество полученных на данный момент байтов. Свойство `bytesTotal` содержит общее количество байтов в загружаемом файле или ноль, если информация недоступна.

Данные полностью загружены, когда значения свойств `bytesLoaded` и `bytesTotal` равны. Однако обычно о завершении загрузки судят не по значениям этих свойств, а по событию `complete`, которое рассылается, когда данные полностью загружены. В рецептах 19.1, 19.2 и 19.3 представлены примеры обработки события `complete`.

Процесс загрузки можно контролировать с помощью следующего кода:

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*
  import flash.net.*;

  public class CheckProgressExample extends Sprite {
    private var _output:TextField;

    public function CheckProgressExample() {
      initializeOutput();
      loadData();
    }
```

```
        private function initializeOutput():void {
          _output = new TextField( );
          _output.width = stage.stageWidth;
          _output.height = stage.stageHeight;

          addChild( _output );
        }

        private function loadData():void {
          var loader:URLLoader = new URLLoader();

          // Слушаем событие progress, чтобы проверить,
          // сколько данных загружено
          loader.addEventListener( ProgressEvent.PROGRESS, handleProgress );

          loader.load( new URLRequest( "example.txt" ) );
        }

        private function handleProgress( event:ProgressEvent ):void {
          // Вычисляем долю загруженных данных, умножая
          // отношение загруженных к общему количеству на 100
          var percent:Number = Math.round( event.bytesLoaded
                                          / event.bytesTotal * 100 );

          _output.text = " Загружено: " + event.bytesLoaded + "\n"
                       + "  Всего: " + event.bytesTotal + "\n"
                       + "Процентов: " + percent;
        }
      }
    }
```

Очень маловероятно, что на практике потребуется отслеживать процесс загрузки посредством события `progress`. Это может пригодиться только при загрузке очень больших объемов данных.

URLLoader не обеспечивает возможности читать данные в процессе загрузки, поэтому событие `progress` всего лишь уведомляет о том, что загрузка выполняется. Для организации доступа к данным в процессе загрузки используется класс *URLStream*, обсуждаемый в рецепте 19.5.

См. также

Рецепты 19.1, 19.2, 19.3 и 19.5.

19.5. Доступ к данным в процессе загрузки

Задача

Требуется организовать доступ к данным в процессе их загрузки.

Решение

Чтобы иметь возможность читать двоичные данные во время их загрузки в ролик, используйте экземпляр *flash.net.URLStream*.

Обсуждение

Рецепт 19.4 рассказывает, как контролировать процесс загрузки данных. Однако из-за природы класса *URLLoader* данные не могут быть прочитаны, пока не завершена загрузка. Чтобы читать данные по мере их поступления, необходимо задействовать класс *URLStream*.

Класс *URLStream* позволяет читать данные в двоичном формате по мере их загрузки во Flash Player подобно использованию *URLLoader*, когда свойству `dataFormat` задано значение `URLLoaderDataFormat.BINARY`. Интерфейс *URLStream* очень похож на *URLLoader*. В обоих классах успешность загрузки контролируют одни и те же события, описанные в рецепте 19.1. Основное отличие в том, как обрабатывается событие `progress`.

В *URLLoader* событие `progress` годится только для представления объема загруженных данных и, возможно, отображения этого количества в процентном соотношении к общему объему данных где-то на экране. В *URLStream* событие `progress` позволяет проверять данные с помощью свойства `bytesAvailable` и одного из методов чтения, скажем, *readInt( )*, *readByte( )*, *readBoolean( )* и др.

URLStream является расширенной возможностью, и его рекомендуется использовать, только если *URLLoader* не удовлетворяет всем требованиям или если разработчик умеет работать с данными на байтовом уровне.

Ниже приведен пример, в котором *URLStream* служит для загрузки файла *.txt* и доступа к данным файла по мере их загрузки во Flash Player:

```
package {
  import flash.display.*;
  import flash.events.*
  import flash.net.*;

  public class CheckProgressExample extends Sprite {

    public function CheckProgressExample() {
      var streamer:URLStream = new URLStream();

      // Слушаем событие progress, чтобы воздействовать
      // на данные при его получении
      streamer.addEventListener( ProgressEvent.PROGRESS, handleProgress );

      streamer.load( new URLRequest( "example.txt" ) );
    }

    private function handleProgress( event:ProgressEvent ):void {
      // Приводим свойство target события к URLStream
      var streamer:URLStream = URLStream( event.target );

      // Перебираем все доступные разряды
```

```
            while ( streamer.bytesAvailable > 0 ) {
              // Читаем значение разряда и выводим его в консоль
              trace( "Читаем разряд: " + streamer.readByte() );
            }
          }
        }
      }
```

В этом примере кода перед осуществлением операции чтения важно с помощью свойства `bytesAvailable` проверять, сколько байт доступно. Попытка считать больше байт, чем доступно во входном буфере, приводит к формированию исключения *EOFError*.

См. также

Рецепты 19.1 и 19.4.

19.6. Отправка данных в сценарий, выполняющийся на стороне сервера

Задача

Требуется отправить данные из ролика Flash в сценарий, выполняющийся на стороне сервера.

Решение

Создайте экземпляр *URLRequest*, содержащий данные, и передайте его в сценарий с помощью метода *flash.net.sendToURL( )*. Результаты выполнения сценария можно отобразить в конкретном окне броузера посредством метода *flash.net.navigateToURL( )*. Если ожидаете получить подтверждение доставки, то вместо метода *sendToURL( )* применяйте метод *URLLoader.load( )*, как описано в рецептах 3.12 и 19.7.

Обсуждение

Метод *sendToUrl( )* из пакета *flash.net* предназначен для отправки данных в серверный сценарий, если результат не требует обработки. Например, необходимо представить данные веб-формы серверному сценарию, не отображая результаты их обработки на стороне сервера. Однако метод *sendToURL( )* никак не подтверждает получение данных и поэтому не подходит в большинстве случаев. Даже при простом выводе на экран статического сообщения, такого как «Спасибо за предоставление данных формы», требуется подтверждение благополучной доставки переменных на сервер. Поэтому, если необходимо подтверждать получение, надо применять метод *URLLoader.load( )*, как описано в рецептах 3.12 и 19.7.

Экземпляр *URLRequest*, переданный в *sendToUrl*, определяет, какие данные (и как) отправляются в серверный сценарий. Обычно для передачи пар имя-значение на сервер свойство `data` запроса содержит экземпляр *URLVariables*. При отправке двоичных данных HTTP-методом `POST` это свойство содержит *flash.utils.ByteArray* или *String* для отправки текста, например запроса XML-RPC (смотрите *http://www.xml-rpc.com*). Следующий фрагмент кода показывает, как создавать *URLRequest* и использовать *URLVariables* для отправки пар имя-значение в сценарий, выполняющийся на стороне сервера:

```
function sendData():void {
  // Создаем запрос, передающий данные на страницу process.cfm
  var request:URLRequest = new URLRequest( "process.cfm" );

  // Создаем некоторые переменные для передачи, someText и someNumber.
  var variables:URLVariables = new URLVariables();
  variables.someText = "Некоторый текст, который необходимо отправить";
  variables.someNumber = 26.2;

  // Присваиваем отправляемые данные созданным ранее переменным
  request.data = variables;

  // Передаем данные в сценарий для обработки
  sendToURL( request );
}
```

URL, передаваемый при создании *URLRequest*, может быть абсолютным или относительным. Возможности работы с ним ограничиваются безопасной изолированной программной средой. Более подробно об этом рассказано в рецепте 3.12.

```
// Задаем абсолютный URL сценария CGI в свойстве url
request.url = "http://www.darronschall.com/cgi-bin/submitVars.cgi";

// Задаем в свойстве url относительный URL сценария
// относительно местоположения файла .swf
request.url = "cgi-bin/submitVars.cgi";
```

Ролик Flash игнорирует ответ сервера на метод *sendToURL( )*, и данные передаются негласно. Если требуется передать данные в сценарий и открыть результаты в определенном окне броузера, используется метод *navigateToURL( )*. Этот метод практически аналогичен *sendToURL( )*, за исключением того, что принимает дополнительный параметр – имя окна, в которое должны быть отправлены результаты. Чтобы вывести результат в новом окне броузера, задается имя окна `blank` (пустой). Результат можно вывести и в другое уже открытое именованное окно или кадр, если известно его имя. Это никак не затрагивает окно броузера с роликом Flash, если только цель не задана как `_self` или `_parent`. В этом случае произойдет замещение страницы, содержащей ролик Flash.

```
// Посылаем переменные, содержащиеся в экземпляре URLRequest,
// в сценарий и отображаем результат в новом окне броузера
navigateToURL( request, "_blank" );
```

Если требуется, чтобы в ролик Flash возвращался ответ, применяется метод *URLLoader.load( )*, рассмотренный в рецептах 3.12 и 19.7.

По умолчанию в случае применения методов *sendToURL( )* или *navigateToURL( )* данные отправляются в заданный сценарий посредством HTTP-метода `POST` (отправить). Это поведение можно изменить, задавая разные значения свойству `method` (метод) экземпляра *URLRequest*, содержащего данные. Значение `URLRequestMethod.GET` определяет применение HTTP-метода `GET`, а значение `URLRequestMethod.POST` – метода `POST`.

```
var request:URLRequest = new URLRequest( "cgi-bin/submit.cgi" );

// Создаем некоторые переменные для передачи
var variables:URLVariables = new URLVariables();
variables.someText = "Отправь меня!";
request.data = variables;

// Конфигурируем переменные, чтобы они передавались HTTP-методом POST
request.method = URLRequestMethod.POST;

// Посылаем запрос и открываем ответ в новом окне
navigateToURL( request, "_blank" );
```

См. также

Рецепты 3.12 и 19.7.

19.7. Отправка переменных и обработка возвращенного результата

Задача

Требуется послать переменные в серверный сценарий и обработать возвращенные сценарием результаты.

Решение

Задача решается с помощью метода *URLLoader.load( )* в сочетании с экземпляром *URLRequest* с заданным свойством `data`.

Обсуждение

Метод *URLLoader.load( )* позволяет отправлять переменные в серверный сценарий и возвращать результаты во Flash. Пример такого сценария – каталог продуктов электронного магазина Flash, данные которого хранятся в базе данных. Как правило, данные распределены по категориям. Когда пользователь выбирает категорию, ролик Flash отправляет ее ID в серверный сценарий и ожидает от него все элементы, входящие в данную категорию.

Метод *URLLoader.load( )* передает переменные в серверный сценарий абсолютно так же, как это делают методы *sendToURL( )* и *navigate-*

ToURL( ). Данные сохраняются в свойство `data` экземпляра *URLRequest*, который передается в метод *load( )* и отправляется в сценарий по заданному URL. Обработка результатов осуществляется точно так же, как описано в рецепте 19.2. Обрабатывается событие `complete`, а свойство `data` класса *URLLoader*, сформировавшего это событие, содержит результат обработки данных сценарием.

Вот полный пример, в котором осуществляется отправка данных в сценарий, возвращение URL-кодированных значений и размещение результата в текстовом поле на экране:

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*
  import flash.net.*;

  public class SendAndLoadExample extends Sprite {
    private var _output:TextField;

    public function SendAndLoadExample() {
      initializeOutput();
      sendData();
    }

    private function initializeOutput():void {
      _output = new TextField();
      _output.width = stage.stageWidth;
      _output.height = stage.stageHeight;

      addChild( _output );
    }

    private function sendData():void {
      // Создаем URLRequest для размещения данных,
      // отправляемых process.cfm
      var request:URLRequest = new URLRequest( "process.cfm" );

      // Создаем пары имя-значение для отправки на сервер
      var variables:URLVariables = new URLVariables();
      variables.method = "getProductDetail"
      variables.productId = 2;
      request.data = variables;

      // Создаем URLLoader для отправки данных
      // и получения ответа
      var loader:URLLoader = new URLLoader();

      // Ожидаем, что сценарий возвратит URL-кодированные переменные
      loader.dataFormat = URLLoaderDataFormat.VARIABLES;

      // Слушаем событие complete,
      // чтобы прочитать ответ сервера
      loader.addEventListener( Event.COMPLETE, handleComplete );
```

```
            // Отправляем данные, находящиеся в URLRequest, в сценарий
            loader.load( request );
        }

        private function handleComplete( event:Event ):void {
            var loader:URLLoader = URLLoader( event.target );

            // Ожидаем, что сценарий возвратит переменные
            // name и description. Выводим эти значения
            // на экран в текстовом поле.
            _output.text = "Имя: " + loader.data.name + "\n"
                         + "Описание: " + loader.data.description;
        }
    }
}
```

См. также

Рецепты 19.1, 19.2, 19.3, 19.4 и 19.6.

20

XML

20.0. Введение

XML – это средство структурированного форматирования и описания данных в текстовом представлении. Изначально этот язык разметки разрабатывался для обеспечения простоты и гибкости, и благодаря своей универсальности, особенно в применении к обмену данными и обеспечению возможности взаимодействия приложений, быстро превратился в промышленный стандарт.

С XML можно столкнуться и при работе с ActionScript. В главе 19 рассказывается, как посылать и загружать данные в URL-кодированном формате. URL-кодирование годится для обмена простыми данными между Flash Player и сценариями, выполняющимися на стороне сервера, но для сложных данных или символов Unicode лучше использовать XML, поскольку он структурирован. Например, из текстового файла требуется загрузить простой тип данных, такой как строка. С помощью экземпляра *URLLoader* могут быть загружены такие URL-кодированные данные:

```
myString=a+string+value
```

Однако если требуется загрузить данные из внешнего источника и использовать их для создания объекта ActionScript, возникает проблема, как представить эти данные в виде URL-кодированной строки. Можно придумывать различные приемы, как например здесь, где каждая пара свойство-значение выделена звездочками (`*`) и каждое свойство отделено от его значения вертикальной чертой (`|`):

```
myObject=prop0|val0*prop1|val1*prop2|val2
```

Получив строковое значение `myObject` (мой объект), с помощью свойства *String.split( )* можно воссоздать элементы, образующие объект. Та-

кой подход имеет право на жизнь, однако, как правило, сложные значения намного проще представлять в формате XML. Например, тот же объект можно представить следующим фрагментом XML:

```
<myObject>
  <prop0>val0</prop0>
  <prop1>val1</prop1>
  <prop2>val2</prop2>
</myObject>
```

XML-данные предлагают ряд преимуществ по сравнению с URL-кодированными данными:

- С помощью XML намного проще представлять сложные данные. XML можно создавать вручную (для статического XML-документа) или программно (из сценария ColdFusion, PHP и т. д.).
- Большинство языков серверных сценариев предлагают встроенную функциональность для чтения и генерирования XML-данных.
- XML – это стандарт, используемый для передачи и хранения данных во всех типах приложений и платформ.

Конечно, XML не единственный способ передачи данных для Flash Player. В главах 19, 21 и 24 обсуждаются средства передачи данных и вне Flash Player. Однако данная глава посвящена исключительно XML, стандартизованной технике обмена данными, которая не требует применения дополнительного серверного ПО (в отличие от Flash Remoting и сетевых соединений). XML стал важной частью ActionScript 3.0, в котором появляется специальное средство работы с XML.

ActionScript 3.0 может похвастать революционно новым синтаксисом для работы с XML. Спецификация *ECMAScript for XML, которой соответствует* ActionScript 3.0, известная также как *E4X*, – это расширение языка *ECMAScript*, обеспечивающее более простой и удобный для чтения подход для работы с объектами XML, чем традиционная, безнадежно устаревшая *объектная модель документов* (Document Object Model, DOM). С появлением E4X работать с XML стало намного проще, чем раньше. Кроме того, если вы впервые сталкиваетесь с XML, E4X сильно ускоряет процесс обучения работе с ним.

В данной главе мы придерживаемся следующей терминологии:

документ XML

Файл, содержащий XML. Данным термином также могут обозначаться загружаемые или отправляемые XML-данные. Не надо путать документ XML с классом *XMLDocument*.

пакет XML

Пакетом XML может быть любой фрагмент XML – от целого документа XML до отдельного узла – если он представляет завершенный, корректный (well-formed) элемент информации в формате XML.

узел XML

Основной строительный блок XML. Узлами могут быть элементы, текстовые узлы, атрибуты и т. д. Обычно элементы и текстовые узлы называют просто «узлами».

элемент XML

Термин «элемент» обычно употребляют как синоним термина «тег». Однако если быть более точным, элемент содержит теги. Элементы должны иметь открывающий и закрывающий теги (`<элемент></элемент>`), или открывающий и закрывающий теги могут быть объединены в один, если элемент не содержит вложенных элементов (`<элемент />`).

Корневой узел

Элемент, находящийся в вершине XML-иерархии элементов.

Текстовый узел

Узел, содержащий текст. Текстовые узлы обычно вложены в элементы.

Атрибут

Это часть элемента. Атрибуты размещаются в тегах элементов в формате имя/значение, например `<элемент имя="значение">`.

объявление XML

Объявление обычно выглядит так: `<?xml version="1.0" ?>`. Это специальный тег, который распознается анализатором XML как тег, содержащий информацию об XML-документе, и не разбирается как элемент.

дерево XML

Иногда его также называют «дерево данных». Дерево XML – это иерархия узлов в XML-документе.

20.1. Изучение структуры XML (чтение и запись XML)

Задача

Научиться понимать, как записывать и читать XML.

Решение

XML основывается на тегах и имеет иерархическую структуру. Если вы знаете HTML, изучение основ XML не должно вызвать трудностей.

Обсуждение

Хотя при работе с ActionScript чтение и написание хорошего XML не главное, но от этого ваш ActionScript только выиграет. Если вы еще не знакомы с XML, не волнуйтесь, в нем нет ничего страшного.

XML – это способ представления структурированных данных. То есть содержимое данных описывается явно. Например, есть строка данных:

```
Джерри, Кэролин, Лаура
```

С помощью XML можно сообщить, *кто* эти люди:

```
<family>
  <father>Джерри</father>
  <mother>Кэролин</mother>
  <sister>Лаура</sister>
</family>
```

Теперь, как видите, XML предоставляет намного больше информации о данных. Вот еще несколько моментов, которые необходимо отметить об XML:

- Структура документа XML образована главным образом узлами. Узел – это общий термин, который применим ко многим частям XML. Например, `<family>` (семья) – это узел предыдущего фрагмента XML; такие узлы называют *элементами*. Значения `Джерри`, `Кэролин` и `Лаура` тоже представляют собой узлы; такие узлы называют *текстовыми*.
- Каждый элемент XML должен иметь соответствующие открывающий и закрывающий теги. Пример открывающего тега: `<family>`. Закрывающий тег идентичен открывающему, за исключением наличия в нем прямого слэша, обозначающего, что это закрывающий элемент: `</family>`. Если элемент не содержит вложенные узлы, открывающий и закрывающий теги могут быть объединены. Например, `<emptyElement />` (пустой элемент) – элемент, сочетающий в себе открывающий и закрывающий теги. Обратите внимание на пробел между именем элемента, `emptyElement`, и прямым слэшем. Пробел может и отсутствовать – это дело вкуса.
- Элементы могут содержать вложенные узлы (будь то другие элементы или текстовые узлы). Несколько примеров этому можно найти в приведенном выше фрагменте XML-документа `<family>`. Элемент `<family>`, который является *корневым узлом* в данном примере, содержит три вложенных элемента: `<father>` (отец), `<mother>` (мать) и `<sister>` (сестра). Такие вложенные узлы еще называют *дочерними узлами*. Каждый из этих дочерних узлов также содержит вложенный узел. Однако эти вложенные узлы являются текстовыми узлами, а не элементами. Несмотря на это они по-прежнему рассматриваются как дочерние узлы.

Нам хотелось бы рассмотреть здесь еще один тип узлов. *Атрибут* – это специальный тип узла, который задается элементу и во многих случаях может даже использоваться как альтернатива вложенному узлу. Если вы когда-нибудь работали с HTML, то уже знакомы с атрибутами. Среди самых распространенных атрибутов HTML можно назвать атрибут `href` элемента `<a>` и атрибут `colspan` элемента `<td>`. Вот как из-

менится рассматриваемый ранее XML-документ, если заменить в нем вложенные узлы атрибутами:

```
<family father="Джерри" mother="Кэролин" sister="Лаура" />
```

Обратите внимание, что мы смогли убрать вложенные элементы и записать те же данные в одном элементе. Также заметьте, что поскольку в <family> больше нет вложенных узлов, открывающий и закрывающий теги можно объединить.

Возникает вопрос, когда и почему использовать атрибуты, а когда – вложенные узлы. Часто это дело вкуса. Иногда проще или понятнее выглядит запись данных XML с помощью атрибутов. Как правило, если необходимо представить сравнительно небольшое количество относительно коротких значений, лучше выбрать атрибуты. Имена атрибутов должны быть уникальными в рамках элемента. Если требуется представить много данных, если данные довольно длинные (более нескольких слов) или если невозможно обеспечить уникальность имен атрибутов в элементе, используются вложенные элементы.

Атрибуты и вложенные узлы можно сочетать. Вот пример элемента <article> (статья), который включает атрибуты title и author, но использует вложенный текстовый узел для представления текста статьи. Это хороший пример того, когда одно из значений (текст статьи) просто слишком длинное, чтобы быть атрибутом.

```
<article title="XML: не только для чокнутых" author="Сэмюел Р.
Шимовитц">
Мои друзья не могли поверить в то, что я занялся XML.
Я стал изгоем, заключенным в темноте офиса, освещаемого только моим верным
монитором.
</article>
```

При составлении XML-данных для отправки во внешний сценарий XML-строки можно создавать прямо во Flash. Например:

```
var dataToSend:String = "<feedback name='Анон' comments='хорошо'>";
```

Статический документ XML вне Flash создается в любом текстовом редакторе и сохраняется как простой текст. Динамическое создание XML-документа с использованием серверного сценария можно найти в соответствующей документации языка программирования.

См. также

В данном рецепте приведены очень простые примеры. XML может быть намного более сложным и включать пространства имен, объявления типов документов и т. д. Большинство из этих элементов выходят за рамки данной книги и за рамки того, что требуется знать для работы с XML в ActionScript. Более подробную информацию об XML можно найти на сайте *http://www.xml.com*. Чтобы получить общее представление об XML, рекомендуем обратиться к книге Эллиотта Расти

Гарольда (Elliotte Rusty Harold) и В. Скотта Минса «XML in a Nutshell» (O'Reilly)[1] и/или книге Саймона Ст. Лорента (Simon St.Laurent) и Майкла Фицджеральда (Michael Fitzgerald) «XML Pocket Reference» (O'Reilly).

20.2. Создание объекта XML

Задача

Требуется создать объект XML с древовидной структурой и данными.

Решение

Заполненный XML-объект содержит данные, как город, заселенный людьми.

Заполненный XML-объект можно создать одним из следующих способов:

- Создайте объект XML и заполните его, присвоив ему XML-литерал.
- Заполните дерево объекта XML, передавая в конструктор *XML* строку XML.
- Создайте пустой объект XML и с помощью E4X заполняйте узлы дерева XML под одному.
- Создайте пустой объект и загрузите в него XML-данные из внешнего источника.

Обсуждение

Заполненный объект XML в ActionScript можно создавать по-разному. Каждый метод предлагает свои преимущества, поэтому при выборе необходимо исходить из нужд конкретного проекта.

Самый простой способ создать заполненный XML-объект – создать новый XML-объект и присвоить ему XML-литерал:

```
var example:XML = <abc><a>eh</a><b>bee</b><c>see</c></abc>;
```

Это пример E4X в действии. Обратите внимание, что пакет XML используется прямо в коде в правой части равенства. Если бы XML был заключен в кавычки, он интерпретировался бы как строка, но поддержку типов данных XML встроена в ActionScript 3.0. Компилятор ActionScript понимает, что выражение в правой части – это XML, и заполняет XML-экземпляр `example` предоставленным пакетом XML.

Такая техника подходит, если структура XML известна еще до создания объекта. Она даже допускает наличие динамических данных в ли-

[1] Гарольд Э., Минс С. «XML. Справочник». – Пер. с англ. – СПб: Символ-Плюс, 2002.

терале XML. Имя переменной в литерале XML заключается в фигурные скобки ({and }). Например, для простого XML-объекта, предназначенного для передачи на сервер имени пользователя и счета, был бы создан такой XML-литерал:

```
// Предположим, существует две переменные,
// username и score
var username:String = "Деррон";
var score:int = 1000;

// Заключаем имя переменной в фигурные скобки,
// чтобы использовать ее значение при создании
// объекта XML с помощью XML-литерала
var example:XML = <gamescore>
                    <username>{username}</username>
                    <score>{score}</score>
                  </gamescore>;
```

Также можно создать строку и передать ее как параметр в конструктор XML. Предыдущий пример тогда записывается следующим образом:

```
// Создаем структуру XML с помощью строки с использованием
// значений username и score в пакете XML.
var str:String = "<gamescore><username>" + username + "</username>"
               + "<score>" + score + "</score></gamescore>";

// Передаем строку в конструктор для создания объекта XML
var example:XML = new XML( str );
```

Структура XML-объектов `example`, созданных в двух предыдущих примерах кода, идентична. XML-объекты можно создавать с использованием фигурных скобок, если на момент компиляции известна их структура, но данные поставляются во время выполнения (как счет, который формируется после окончания игры).

Иногда не все данные известны сразу. В этом случае и дерево объекта XML создается лишь через некоторое время. Так, если XML-объект предназначен для хранения информации о корзине покупок пользователя, данные объекта должны меняться при каждом изменении набора товаров в корзине. В таких случаях создается объект XML, в который затем с помощью E4X добавляются или удаляются узлы, как описывается в рецептах 20.3 и 20.10.

И не будем забывать о заполнении объектов XML данными, извлеченными из внешнего источника, такого как статический XML-документ или сценарий, который генерирует динамический XML. До сих пор мы рассматривали создание XML-объектов только для отправки данных на сервер или для использования в собственных программах, но можно и загружать данные с сервера, например при извлечении пользовательских данных, информации каталога или настроечной информации ролика. Для таких сценариев в рецепте 20.11 рассказывается, как загружать документ XML с сервера.

См. также

Рецепты 20.3, 20.4, 20.5 и 20.10, в которых более подробно рассказывается о создании дерева XML с использованием E4X. О загрузке информации из внешних источников рассказывает рецепт 20.11.

20.3. Добавление элементов в объект XML

Задача

Требуется создать объект *XML* и добавить в него элементы.

Решение

Используйте синтаксис E4X для создания дочерних элементов и добавления их в дерево XML. Кроме того, больший контроль за добавлением новых элементов обеспечат методы *insertChildBefore( )* и *insertChildAfter( )*.

Обсуждение

В объект XML можно добавлять элементы для создания структуры данных XML и ее передачи в другое приложение. Это можно сделать несколькими способами, как обсуждается в рецепте 20.2.

Синтаксис E4X позволяет без труда добавлять элементы в объект XML. Элементы добавляются в экземпляр XML с помощью оператора «точка» (.), практически аналогично добавлению типичного свойства в обычный *Object*:

```
// Создаем экземпляр XML, в который будем добавлять элементы
var example:XML = <example />;

// Создаем новый узел XML newElement
// и добавляем его в экземпляр example
example.newElement = <newElement />;

/* На экран выводится:
  <example>
    <newElement/>
  </example>
*/
trace( example );
```

В предыдущем примере создается литерал XML для пустого узла `newElement` (новый элемент), который затем добавляется в экземпляр `example` посредством оператора «точка» и присвоения свойства `newElement` узлу `newElement`. В данной ситуации не обязательно, чтобы имя свойства и имя элемента совпадали. Однако было бы странно, если бы имена были разными. При добавлении узла XML в существующий объект XML имя элемента замещает имя свойства.

Также новый элемент можно ввести, создавая свойство экземпляра XML и присваивая ему значение:

```
// Создаем экземпляр XML
var example:XML = <example />;

// Создаем новое свойство emptyElement
// и присваиваем ему пустую строку
example.emptyElement = "";

/* На экран выводится:
  <example>
    <emptyElement/>
  </example>
*/
trace( example );
```

Предыдущий пример демонстрирует, что дочерние элементы XML-объекта можно создавать, не добавляя XML-узлы фактически, как в первом примере. Задавая пустую строку как значение свойства `emptyElement` (пустой элемент), мы создаем пустой узел элементов `emptyElement`.

Если имя добавляемого элемента неизвестно или если оно должно основываться на значении переменной, можно использовать альтернативную запись с квадратными скобками. Вместо применения оператора «точка» заключаем строку в квадратные скобки (`[` и `]`). Строка вычисляется и полученное значение выступает в качестве имени свойства. Это позволяет динамически создавать имя свойства и, следовательно, имя элемента:

```
// Создаем экземпляр XML
var example:XML = <example />;

var id:int = 10;

// Создаем строку, чтобы включить значение id в имя узла
example[ "user" + id ] = "";

/* На экран выводится:
  <example>
    <user10/>
  </example>
*/
trace( example );
```

Иногда без синтаксиса квадратных скобок не обойтись. Например, в XML в имени узла элементов допускается дефис. Попытка создать имя элемента, включающее дефис, с помощью оператора «точка» приводит к ошибке компиляции:

```
example.some-element = ""; // Формируется ошибка компиляции
```

Дефис имеет специальное значение для компилятора: он является знаком вычитания. Предыдущий фрагмент кода интерпретируется как

выражение вычитания переменной *element* из *example.some* и последующего присваивания пустой строки этому значению. Такое выражение вычитания не является синтаксически правильным, что объясняет появление ошибки компилятора. Более того, даже если бы это выражение было скомпилировано, оно бы не обеспечило необходимого поведения создания узла элементов. Применение квадратных скобок устраняет сложности:

```
example[ "some-element" ] = "";
```

Простота E4X, однако, имеет и неприятную оборотную сторону: дело в том, что новый элемент всегда добавляется только в конец дерева XML. Методы *insertChildBefore( )* (вставить дочерний элемент перед) и *insertChildAfter( )* (вставить дочерний элемент после) обеспечивают больший контроль над добавлением новых узлов. Первый из них вставляет новый элемент перед элементом дерева XML, а второй – после элемента. Оба метода принимают два одинаковых параметра: узел, отмечающий точку вставки, и вставляемые данные. Следующий фрагмент кода иллюстрирует изменение дерева XML с помощью обоих методов:

```
// Создаем экземпляр XML
var example:XML = <example/>;

// Создаем пустой узел элемента two
example.two = "";

// Вставляем узел one перед узлом two
example = example.insertChildBefore( example.two, <one /> );

// После узла two вставляем узел three
example = example.insertChildAfter( example.two, <three /> );

/* На экран выводится:
<example>
  <one/>
  <two/>
  <three/>
</example>
*/
trace( example );
```

Обратите внимание, что методы *insertChildBefore( )* и *insertChildAfter( )* не меняют экземпляр XML, для которого они вызваны, а возвращают новый экземпляр, содержащий изменения. Чтобы зафиксировать изменения, результат вызова метода необходимо опять присвоить экземпляру XML, как показано выше.

См. также

Рецепты 20.2 и 20.4.

20.4. Добавление текстовых узлов в объект XML

Задача

Требуется добавить текстовые узлы в объект *XML*.

Решение

Используя синтаксис E4X, создайте текстовые узлы и добавьте их в дерево XML. Для получения большего контроля над добавлением текстовых узлов обратитесь к методам *appendChild( )*, *prependChild( )*, *insertChildAfter( )* и *insertChildBefore( )*.

Обсуждение

Создавать текстовые узлы с помощью E4X, как вы уже догадались, просто. Процесс аналогичен тому, который был описан в рецепте 20.3. Как и в случае узлов элементов, для создания текстовых узлов используется оператор «точка» (.), с помощью которого создается свойство экземпляра XML и ему присваивается любое значение, которое может быть преобразовано в строку:

```
// Создаем экземпляр XML
var example:XML = <example/>;

// Создаем текстовый узел из строки
example.firstname = "Деррон";

// Создаем текстовый узел из числа
example.number = 24.9;

// Создаем текстовый узел из Булева значения
example.boolean = true;

// Создаем текстовый узел из массива
example.abc = ["a", undefined, "b", "c", null, 7, false];

/* На экран выводится:
<example>
  <firstname>Деррон</firstname>
  <number>24.9</number>
  <boolean>true</boolean>
  <abc>a,,b,c,,7,false</abc>
</example>
*/
trace( example );
```

В этом примере в XML-экземпляр добавляются и текстовые узлы, и узлы элементов. Значение в правой части выражения присваивания преобразуется в строку и становится текстовым узлом. Затем имя свойства преобразуется в узел элемента с текстовым узлом в качестве дочернего элемента, и весь узел элемента добавляется в экземпляр XML.

Для управления размещением текстовых узлов в узлах элементов используются методы *appendChild( )* (добавить дочерний элемент в конец), *prependChild( )* (добавить дочерний элемент в начало), *insertChildBefore( )* или *insertChildAfter( )*. Эти методы позволяют точно определить, как текстовый узел вставляется в дерево XML:

```
// Создаем экземпляр XML
var example:XML = <example/>;

// Добавляем в конец дерева узел элемента two, содержащий
// дочерний текстовый узел со значением 2
example.appendChild( <two>2</two> );

// Добавляем в начало дерева узел элемента one, содержащий
// дочерний текстовый узел со значением "номер 1"
example.prependChild( <one>"номер 1"</one> );

// После узла элемента one вставляем текстовый узел
// со значением 1.5
example.insertChildAfter( example.one[0], 1.5 );

// Перед узлом элемента two вставляем узел элемента part,
// содержащий дочерний текстовый узел со значением 1.75
example.insertChildBefore( example.two[0], <part>1.75</part> );

/* На экран выводится:
<example>
  <one>"номер 1"</one>
  1.5
  <part>1.75</part>
  <two>2</two>
</example>
*/
trace( example );
```

В предыдущем примере кода был создан *смешанный элемент*. Смешанный элемент – это элемент, содержащий и вложенные текстовые узлы, и вложенные элементы. Элемент `example` смешанный, потому что содержит не только дочерние элементы (`<one>`, `<part>` и `<two>`), но и вложенный текстовый узел (со значением 1,5). Это вполне допустимо в XML и демонстрирует, что у элемента может быть много дочерних элементов, а не только один текстовый узел.

См. также

Рецепт 20.3.

20.5. Добавление атрибутов в элемент XML

Задача

Требуется добавить атрибуты в элемент XML.

Решение

Присваивание атрибутов узлам элементов в синтаксисе E4X осуществляется с помощью оператора @.

Обсуждение

Для присваивания атрибутов элементу в E4X предназначен оператор @; базовый синтаксис такой:

```
узелЭлемента.@имяАтрибута = "значение";
```

После переменной узла элемента ставится оператор «точка» (.), за которым следует оператор @. Сразу после символа @ задается имя атрибута и простым выражением присваивания задается значение атрибута. Значение в правой части выражения присваивания перед присваиванием преобразуется в строку:

```
// Создаем экземпляр XML
var example:XML = <example><someElement/></example>;

// Добавляем атрибуты в узел элемента someElement
example.someElement.@number = 12.1;
example.someElement.@string = "пример";
example.someElement.@boolean = true;
example.someElement.@array = ["a", null, 7, undefined, "c"];

/* На экран выводится:
<example>
  <someElement number="12.1" string="пример" boolean="true"
  array="a,,7,,c"/>
</example>
*/
trace( example );
```

Этот синтаксис требует, чтобы имя атрибута было действительным именем переменной. Это означает, что имена атрибутов могут включать числа, буквы и символы подчеркивания, первый символ не может быть числом. Однако имена XML-атрибутов могут содержать и другие символы, которые при использовании с оператором @ приводят к ошибке компиляции. Атрибуты, имена которых содержат недопустимые для имени переменной символы, создаются при помощи синтаксиса с квадратными скобками. Например:

```
example.someElement.@["bad-variable-name"] = "да";
```

Для динамического создания атрибутов можно применять и квадратные скобки, компонуя имя атрибута из значений переменных. Например:

```
example.someElement.@["color" + num] = "красный";
```

См. также

Рецепт 20.9.

20.6. Чтение элементов дерева XML

Задача

Требуется извлечь дочерние элементы объекта *XML*.

Решение

Обратитесь к методу *elements( )*, который возвратит все элементы в виде *XMLList*, и цикл *for each* – для перебора элементов списка.

Обсуждение

Часто возникает необходимость «пройти» (обойти) по дереву XML, чтобы извлечь или проверить один или более его элементов. Это удобный способ поиска заданного элемента или обработки элементов, порядок расположения которых неизвестен или неважен.

E4X предоставляет удобный метод *elements( )* (элементы), возвращающий все дочерние узлы элементов XML-объекта. Поместив этот метод в цикл *for each*, можно совершать обход дерева XML:

```
var menu:XML = <menu>
                 <menuitem label="Файл">
                   <menuitem label="Новый"/>
                 </menuitem>
                 <menuitem label="Помощь">
                   <menuitem label="О программе"/>
                 </menuitem>
                 Это текстовый узел
               </menu>;

for each ( var element:XML in menu.elements() ) {
  /* На экран выводится:
  Файл
  Помощь
  */
  trace( element.@label );
}
```

Данный пример показывает, что метод *elements( )* возвращает только дочерние узлы объекта XML типа *element* (игнорируя узлы другого типа, например текстовые). Он также возвращает только элементы, являющиеся прямыми потомками XML-объекта. Вызов *menu.elements( )* в предыдущем фрагменте кода не возвращает элементы меню `Новый` и `О программе`, потому что они не являются прямыми дочерними элементами `menu`. Чтобы обойти все дерево, необходимо создать рекурсивную функцию:

```
var menu:XML = <menu>
                 <menuitem label="Файл">
                   <menuitem label="Новый"/>
```

```
                </menuitem>
                <menuitem label="Помощь">
                  <menuitem label="О программе"/>
                </menuitem>
                Это текстовый узел
              </menu>;

/* На экран выводится:
Файл
Новый
Помощь
О программе
*/
walk( menu );

// Рекурсивная функция, выбирающая все элементы дерева XML
function walk( node:XML ):void {
  // Перебираем все дочерние элементы узла
  for each ( var element:XML in node.elements() ) {
    // Выводим атрибут label
    trace( element.@label );
    // Рекурсивно обходим дочерний элемент,
    // чтобы выбрать его дочерние элементы
    walk( element );
  }
}
```

См. также

Рецепты 20.7, 20.8 и 20.9.

20.7. Поиск элементов по имени

Задача

Требуется найти элемент по имени узла, а не по положению в иерархии XML.

Решение

Обратитесь к синтаксису E4X и посредством оператора «точка» переходите («dot down») к конкретному элементу.

Обсуждение

E4X позволяет работать с XML-объектами так же просто, как и с обычными объектами. Доступ к узлам элементов осуществляется так же, как к свойствам объекта; например:

```
var fruit:XML = <fruit><name>Яблоко</name></fruit>;

// На экран выводится: Яблоко
trace( fruit.name );
```

Как видите, чтобы получить доступ к элементу name (имя) пакета XML, используется оператор «точка» (.) и прямая ссылка на name XML-экземпляра fruit (фрукт). Для более глубоко вложенных элементов XML-дерева указывается последовательность точек и имен элементов:

```
var author:XML = <author><name><firstName>Деррон</firstName></name></author>;

// На экран выводится: Деррон
trace( author.name.firstName );
```

Краткая запись для организации доступа к глубоко вложенным узлам элементов, когда точный путь к узлу неизвестен, – двойной оператор «точка». Например, можно опустить путь и напрямую обратиться к узлу firstName (имя):

```
var author:XML = <author><name><firstName>Деррон</firstName></name></author>;

// На экран выводится: Деррон
trace( author..firstName );
```

Двойной оператор «точка» применим для любого уровня вложенности, хоть второго, хоть десятого.

Если существует несколько узлов элементов с одинаковыми именами, доступ к ним осуществляется посредством целочисленных индексов, которые указываются в квадратных скобках, как при работе с массивами. Например:

```
var items:XML = <items>
                  <item>
                    <name>Яблоко</name>
                    <color>красный</color>
                  </item>
                  <item>
                    <name>Апельсин</name>
                    <color>оранжевый</color>
                  </item>
                </items>;

// На экран выводится: Яблоко
trace( items.item[0].name );
// На экран выводится: Апельсин
trace( items.item[1].name );
```

items.item возвращает *XMLList* с двумя элементами, каждый из которых представляет узел элементов item XML-литерала. К первому узлу элементов item можно обратиться по индексу 0, ко второму – по индексу 1. Индекс первого элемента всегда равен 0, индекс последнего элемента всегда на единицу меньше, чем общее число элементов. Для подсчета числа выбранных элементов применяется метод *length( )*:

```
// На экран выводится: 2
trace( items.item.length() );
```

Если требуется проверить все узлы элементов с определенным именем и количество таких элементов не известно, используется цикл *for each*:

```
var items:XML = <items>
                  <item>
                    <name>Яблоко</name>
                    <color>красный</color>
                  </item>
                  <item>
                    <name>Апельсин</name>
                    <color>оранжевый</color>
                  </item>
                </items>;

// Используем синтаксис двойной точки,
// таким образом, можно опустить путь
for each ( var name:XML in items..name ) {
  /* На экран выводится:
  Яблоко
  Апельсин
  */
  trace( name );
}
```

Как и в других случаях применения оператора «точка», он может быть заменен квадратными скобками. Это полезно, когда в качестве имени искомого элемента выступает значение переменной:

```
var nodeName:String = "color";

var fruit:XML = <fruit><color>красный</color></fruit>;

// На экран выводится: красный
trace( fruit[nodeName] );
```

Квадратные скобки не могут использоваться с двойным оператором «точка». Например, следующий код приводит к ошибке:

```
trace( fruit..[nodeName] ); // Формируется ошибка компиляции
```

См. также

Рецепт 20.14.

20.8. Чтение текстовых узлов и их значений

Задача

Требуется извлечь значение текстового узла.

Решение

Используйте синтаксис E4X или, в качестве альтернативы, метод *text( )* для возвращения объекта *XMLList*, содержащего текстовые уз-

лы элемента. Затем с помощью метода *toString( )* преобразуйте значение текстового узла в строку. Обратитесь к функциям преобразования, например *int( )* или *Number( )*, для преобразования значения в другой тип данных или предоставьте Flash Player возможность автоматически преобразовать значение в определенный тип.

Обсуждение

Рецепт 20.4 рассматривает текстовые узлы и их создание в объектах XML. Данный рецепт объясняет, как извлекать значение текстового узла. Сначала необходимо получить ссылку на текстовый узел, во многом аналогично узлам элементов, что обсуждалось в рецептах 20.6 и 20.7. Имея ссылку на текстовый узел, можно получить его значение, вызывая метод *toString( )* или другой метод преобразования. Рассмотрим следующий пакет XML:

```
<book>
  <title>ActionScript 3.0. Сборник рецептов</title>
</book>
```

Корневым узлом данного XML-пакета является `<book>` (книга), содержащий узел `<title>` как дочерний элемент. Элемент `<title>`, в свою очередь, включает вложенный узел: текстовый узел, имеющий значение `ActionScript 3.0. Сборник рецептов`.

Если этот XML-пакет присвоен объекту *XML*, можно использовать синтаксис E4X и оператор «точка» для доступа к узлу элемента `title` по имени, а также метод *toString( )* для доступа к его дочернему текстовому узлу:

```
var book:XML = <book>
                 <title>ActionScript 3.0. Сборник рецептов</title>
               </book>;

// Используем синтаксис E4X для организации доступа
// к элементу title и присваиваем его значение переменной.
var title:String = book.title.toString();

// На экран выводится: ActionScript 3.0. Сборник рецептов
trace( title );
```

В этом примере преобразование узла элемента `title` в тип *String* осуществляется явно вызовом метода *toString( )*. На самом деле вызов *toString( )* можно опустить, потому что Flash Player проводит преобразование автоматически. Но явный вызов *toString( )* хорош тем, что делает код более прозрачным и понятным.

Аналогично можно преобразовать текстовый узел в другой тип данных. В следующем примере демонстрируется преобразование текстовых узлов в различные типы с помощью функций преобразования *Boolean( )* и *int( )* и автоматически осуществляемое Flash Player преобразование в тип *Number* (без явного вызова метода *Number( )*):

```
var example:XML = <example>
                    <bool>true</bool>
                    <integer>12</integer>
                    <number>.9</number>
                  </example>;

// Преобразуем текстовый узел "true" в Булево true
var bool:Boolean = Boolean( example.bool );

// Преобразуем текстовый узел "12" в целое
var integer:int = int( example.integer );

// Преобразуем текстовый узел ".9" в число
var number:Number = example.number;

/* На экран выводится:
true
12
0.9
*/
trace( bool );
trace( integer );
trace( number );
```

Этот пример немного неточен. Функция преобразования *Boolean( )* может не обеспечить того результата, который предполагается. Изменение значения текстового узла в элементе `<bool>` с `true` на `false` приводит к тому, что переменной `bool` будет задано значение `true`, а не `false`, как ожидается. Такое поведение заложено при проектировании функции, хотя и кажется немного странным. Простой прием для преобразования строки `false` в Булево `false` – вызвать метод *toLowerCase( )* и сравнить возвращаемое значение со строкой `true`:

```
var bool:Boolean = example.bool.toLowerCase() == "true";
```

До сих пор мы работали с узлами элементов, содержащими всего один дочерний текстовый узел. А что же смешанные элементы? При преобразовании смешанного элемента в строку получается форматированная строка XML:

```
var fruit:XML = <fruit>
                  <name>Яблоко</name>
                  По яблоку в день...
                </fruit>;

// Здесь необходимо явно вызвать toString()
var value:String = fruit.toString();

/* На экран выводится:
<fruit>
  <name>Яблоко</name>
  По яблоку в день...
</fruit>
*/
trace( value );
```

В таких ситуациях возвращение только текстовых узлов элемента обеспечит метод *text( )*. Как показано в рецепте 20.6, для перебора всех узлов организуется цикл *for each*:

```
var fruit:XML = <fruit>
                  <name>Яблоко</name>
                  По яблоку в день...
                </fruit>;

for each ( var textNode:XML in fruit.text() ) {
  // На экран выводится: По яблоку в день...
  trace( textNode );
}
```

См. также

Рецепты 20.4, 20.6 и 20.7.

20.9. Чтение атрибутов элемента

Задача

Требуется извлечь атрибуты элемента.

Решение

Список атрибутов элемента возвращает метод *attributes( )*. В качестве альтернативы может выступать оператор @ синтаксиса E4X или метод *attribute( )* для организации доступа к атрибуту по имени.

Обсуждение

Список атрибутов определенного узла элементов в виде объекта *XMLList* возвращается методом *attributes( )*. Возвращенный *XMLList* является индексируемым, как экземпляр *Array*. По индексу можно получить значение атрибута:

```
var fruit:XML = <fruit name="Яблоко" color="красный" />;

// Используем метод attributes() и сохраняем результаты как XMLList
var attributes:XMLList = fruit.attributes();

// На экран выводится: Яблоко
trace( attributes[0] );
// На экран выводится: красный
trace( attributes[1] );
```

В данном примере извлекаются только значения атрибутов. Имя атрибута позволяет получить метод *name( )*. В следующем примере проверяется имя второго атрибута (расположенного под индексом 1):

```
var fruit:XML = <fruit name="Яблоко" color="красный" />;

// На экран выводится: color
trace( fruit.attributes()[1].name() );
```

Чтобы выбрать все атрибуты элемента, используется цикл *for each*. Чтобы получить имя и значение атрибута, применяется метод *name( )* в сочетании со ссылкой на атрибут. В следующем примере метод *toString( )* вызывается для ссылки на атрибут явно, чтобы преобразовать значение атрибута в строку, как в рецепте 20.8:

```
var fruit:XML = <fruit name="Яблоко" color="красный" />;

for each ( var attribute:XML in fruit.attributes() ) {
  /* На экран выводится:
  name = Яблоко
  color = красный
  */
  trace( attribute.name() + " = " + attribute.toString() );
}
```

Если необходимо получить конкретный атрибут и известно его имя, не составит никакого труда сделать это с помощью синтаксиса E4X. Для этого посредством оператора «точка» необходимо обратиться к конкретному узлу элемента и затем с помощью оператора `@` указать имя атрибута, чтобы получить его значение:

```
var fruit:XML = <fruit name="Яблоко" color="красный" />;

// На экран выводится: красный
trace( fruit.@color );
```

Кроме того, можно вызвать метод *attribute( )*, передав в него имя атрибута:

```
var fruit:XML = <fruit name="Яблоко" color="красный" />;

// На экран выводится: красный
trace( fruit.attribute("color") );
```

Также, чтобы организовать доступ ко всем атрибутам элемента, аналогично способу с методом *attributes( )*, можно прибегнуть к оператору `@` с групповым символом (`*`):

```
var fruit:XML = <fruit name="Яблоко" color="красный" />;

// На экран выводится: Яблоко
trace( fruit.@*[0] );

// На экран выводится: красный
trace( fruit.@*[1] );

// На экран выводится: 2
trace( fruit.@*.length() );
```

Атрибуты всегда возвращаются как *XMLList*, поэтому они проиндексированы, что упрощает доступ к ним.

Синтаксис E4X – исключительно мощное средство. Рассмотрим пакет XML, содержащий ряд элементов, причем некоторые из них имеют атрибут `price` (цена). Следующий фрагмент кода показывает, как можно

использовать E4X для вычисления суммарной стоимости элементов, имеющих атрибут `price`:

```
// Создаем воображаемую корзину покупок
var cart:XML = <cart>
                  <item price=".98">мелки</item>
                  <item price="3.29">карандаши</item>
                  <group>
                    <item price=".48">синяя ручка</item>
                    <item price=".48">черная ручка</item>
                  </group>
                </cart>;

// Создаем переменную total для представления
// общей стоимости товаров в корзине
var total:Number = 0;

// Выбираем каждый атрибут price и добавляем
// его значение к промежуточной сумме
for each ( var price:XML in cart..@price ) {
  total += parseFloat(price);
}

// На экран выводится: 5.23
trace( total );
```

Двойной оператор «точка» перед оператором `@` в синтаксисе E4X обеспечивает поиск атрибутов `price` по всему XML-дереву узла элементов `cart`.

Как показано в рецепте 20.5, для динамического доступа к атрибуту по имени либо в случае, если имя атрибута содержит недопустимые для имен переменных символы, можно использовать синтаксис квадратных скобок:

```
var example:XML = <example bad-variable-name="да" color12="синий" />;
var num:Number = 12;

// На экран выводится: да
trace( example.@["bad-variable-name"] );

// На экран выводится: синий
trace( example.@["color" + num] );
```

См. также

Рецепты 20.5 и 20.8.

20.10. Удаление элементов, текстовых узлов и атрибутов

Задача

Требуется удалить из объекта *XML* узел элементов, текстовый узел или атрибут.

Решение

Это делается при помощи ключевого слова `delete`.

Обсуждение

В предыдущих рецептах мы научились добавлять узлы элементов, текстовые узлы и атрибуты в объекты *XML*. Также было показано, как организовывать работу с ними (чтение и перебор). Но что если требуется удалить определенный элемент или атрибут? Достаточно указать ключевое слово `delete`, а за ним – элемент (атрибут), подлежащий удалению:

```
var example:XML = <example>
                    <fruit color="Красный">Яблоко</fruit>
                    <vegetable color="Зеленый">Брокколи</vegetable>
                    <dairy color="Белый">Молоко</dairy>
                  </example>;

// Удаляем атрибут color из элемента fruit
delete example.fruit.@color;

// Полностью удаляем элемент dairy
delete example.dairy;

// Удаляем текстовый узел из узла элементов vegetable
delete example.vegetable.text()[0];

/* На экран выводится:
<example>
  <fruit>Яблоко</fruit>
  <vegetable color="Зеленый"/>
</example>
*/
trace( example );
```

Особый интерес в предыдущем примере вызывает удаление текстового узла. В случае применения определенных методов объекта *XML*, таких как *text( )* и *elements( )*, или, в определенных ситуациях, синтаксиса E4X возвращается экземпляр *XMLList* с множеством элементов. Ключевое слово `delete` применимо только к одному элементу. Таким образом, чтобы указать конкретный элемент *XMLList*, который следует удалить, необходимо использовать запись с квадратными скобками. Чтобы удалить все элементы *XMLList*, организуется цикл *for* и перебор элементов в обратном порядке:

```
var example:XML = <example>
                    <fruit color="красный" name="Яблоко" />
                  </example>;

// Получаем XMLList атрибутов для fruit
var attributes:XMLList = example.fruit.@*;

// Перебираем элементы в обратной последовательности,
```

```
// чтобы удалить все атрибуты. Удаляя элементы с конца
// массива, мы избегаем проблемы с изменением индексов.
for ( var i:int = attributes.length() - 1; i >= 0; i-- ) {
    delete attributes[i];
}

/* На экран выводится:
<example>
  <fruit/>
</example>
*/
trace( example );
```

20.11. Загрузка XML

Задача

Требуется загрузить данные XML из документа XML или серверного сценария, который генерирует XML.

Решение

Метод *URLLoader.load( )*, свойству `dataFormat` которого задано значение `URLLoaderDataFormat.TEXT`, обеспечит загрузку данных в виде простого текста. Используйте обработчик события `complete` и преобразуйте текст в экземпляр *XML*.

Обсуждение

В предыдущих версиях ActionScript загрузка XML-файла осуществлялась посредством вызова метода *load( )* непосредственно для объекта *XML*. В ActionScript 3.0 отправка и загрузка данных была выделена в класс *URLLoader* и родственные ему классы. В ActionScript 3.0 нет специальных средств загрузки XML, что предоставляет свободу действий разработчикам для реализации собственных решений.

Процесс загрузки файла XML хотя и многошаговый, но относительно безболезненный. Сначала необходимо создать экземпляр *URLLoader* для загрузки данных с URL. Чтобы *URLLoader* загружал данные как простой текст, его свойству `dataFormat` должно быть задано значение `URLLoaderDataFormat.Text`. Добавляем слушателя события `complete`, чтобы получать уведомление о завершении загрузки данных. В обработчике события `complete` для преобразования загруженных данных в объект *XML* задействуется одна из техник, описанных в рецепте 20.2. И наконец, для запуска процесса загрузки вызывается метод *URLLoader.load( )*, в который передается экземпляр *URLRequest*, указывающий на URL файла XML. Законченный пример выглядит так:

```
package {
  import flash.display.*;
  import flash.events.*;
```

```
import flash.net.*;
import flash.utils.*;

public class LoadXMLExample extends Sprite {

  public function LoadXMLExample() {
    var loader:URLLoader = new URLLoader();
    loader.dataFormat = URLLoaderDataFormat.TEXT;
    loader.addEventListener( Event.COMPLETE, handleComplete );
    loader.load( new URLRequest( "example.xml" ) );
  }

  private function handleComplete( event:Event ):void {
    try {
      // Преобразуем загруженный текст в экземпляр XML
      var example:XML = new XML( event.target.data );
      // Здесь example готов к использованию с E4X
      trace( example );

    } catch ( e:TypeError ) {
      // Если мы попали сюда, значит, не получилось преобразовать
      // загруженный текст в экземпляр XML, вероятно, потому что
      // он отформатирован несоответствующим образом.
      trace( "Не получилось преобразовать текст в XML" );
      trace( e.message );
    }
  }
}
```

В предыдущем примере обратите внимание на использование блока *try...catch* в методе `handleComplete`. Если XML-файл не содержит действительной XML-разметки, при попытке преобразования загруженных данных в экземпляр *XML* формируется *TypeError* (ошибка приведения типов данных). Ошибка перехватывается блоком *catch*, что позволяет красиво обработать сбой при синтаксическом разборе.

Более подробно об работе с *URLLoader* рассказано в главе 19.

См. также

Рецепт 20.2 и главу 19, в частности рецепт 19.3.

20.12. Загрузка XML из других доменов

Задача

Требуется загружать XML из других доменов, а не только из того, в котором находится файл *.swf*.

Решение

Сделайте в *файле политики crossdomain.xml* удаленного домена запись, разрешающую доступ к нему из домена файла *.swf*.

Обсуждение

Об использовании *файла политики crossdomain.xml* рассказано в рецепте 3.12.

См. также

Рецепт 3.12.

20.13. Отправка XML

Задача

Требуется отправить данные XML в сценарий, выполняющийся на стороне сервера.

Решение

Создайте экземпляр *URLRequest*, содержащий данные XML, подлежащие отправке. Обратитесь к методу *flash.net.sendToURL( )*, если хотите отправить данные и проигнорировать ответ сервера. Метод *flash.net.navigateToURL( )* позволит отправить данные и открыть ответ сервера в определенном окне броузера, а метод *URLLoader.load( )* – отправить данные и загрузить ответ в файл *.swf*.

Обсуждение

XML используется для передачи данных в приложения и из них, в данном случае – роликов Flash. Поэтому было бы странно, если бы объекты XML создавались в ролике Flash для использования только в нем. Обычно данные XML загружаются из другого источника, во Flash создаются данные XML для отправки в другое приложение или и то, и другое.

В данном рецепте рассматривается отправка данных XML из Flash в другое приложение, что может потребоваться во многих случаях. Например, во Flash-игре XML может задействоваться для передачи на сервер имени пользователя и счета. Иногда пакеты XML посылаются на сервер для активизации какой-либо функциональности сервера. Такой процесс иногда называют *удаленным вызовом процедур* (Remote Procedure Call, RPC), и здесь для передачи на сервер информации вызова функции (имени функции, параметров и т. п.) может применяться XML. Существует формальная спецификация использования XML в таких случаях – XML-RPC (см. *http://www.xmlrpc.com*). Так что, как видите, причины отправки XML на сервер весьма разнообразны.

Как обсуждалось в рецепте 20.11 и главе 19, ActionScript 3.0 объединил отправку и загрузку данных в методах пакета *flash.net*. Ранее класс *XML* содержал и метод *send( )*, и метод *sendAndLoad( )* для отправки XML на сервер, но в ActionScript 3.0 место этих методов занял

экземпляр *URLRequest*. В рецептах 19.6 и 19.7 рассматриваются основные приемы использования экземпляра *URLRequest* для отправки и получения данных.

Рассмотрим законченный рабочий пример. Ниже создается необходимый клиентский код на ActionScript, после чего требуется выбрать одно из серверных решений. Выбирайте то решение, которое поддерживается вашим сервером (или персональным компьютером, если он выступает в качестве сервера). Сценарии, выполняющиеся на стороне сервера, могут быть написаны на Perl, PHP и ColdFusion.

Прежде всего создается новый класс ActionScript 3.0:

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.filters.*;
  import flash.events.*;
  import flash.net.*;

  public class XMLSendLoadExample extends Sprite {

    private var _message:TextField;
    private var _username:TextField;
    private var _save:SimpleButton;

    public function XMLSendLoadExample() {
      initializeDisplay();
    }

    private function initializeDisplay():void {
      _message = new TextField();
      _message.autoSize = TextFieldAutoSize.LEFT;
      _message.x = 10;
      _message.y = 10;
      _message.text = "Enter a user name";

      _username = new TextField();
      _username.width = 100;
      _username.height = 18;
      _username.x = 10;
      _username.y = 30;
      _username.type = TextFieldType.INPUT;
      _username.border = true;
      _username.background = true;

      _save = new SimpleButton();
      _save.upState = createSaveButtonState( 0xFFCC33 );
      _save.overState = createSaveButtonState( 0xFFFFFF );
      _save.downState = createSaveButtonState( 0xCCCCCC );
      _save.hitTestState = _save.upState;
      _save.x = 10;
      _save.y = 50;
      // По щелчку кнопки save вызывается метод handleSave
```

```
    _save.addEventListener( MouseEvent.CLICK, handleSave );

    addChild( _message );
    addChild( _username );
    addChild( _save );
  }

  // Создаем состояние кнопки с определенным цветом фона
  private function createSaveButtonState( color:uint ):Sprite {
    var state:Sprite = new Sprite();

    var label:TextField = new TextField();
    label.text = "Сохранить";
    label.x = 2;
    label.height = 18;
    label.width = 30;
    var background:Shape = new Shape();
    background.graphics.beginFill( color );
    background.graphics.lineStyle( 1, 0x000000 );
    background.graphics.drawRoundRect( 0, 0, 32, 18, 9 );
    background.filters = [ new DropShadowFilter( 1 ) ];

    state.addChild( background );
    state.addChild( label );
    return state;
  }

  private function handleSave( event:MouseEvent ):void {
    // Формируем случайный счет, сохраняемый с username
    var score:int = Math.floor( Math.random() * 10 );

    // Создаем новый экземпляр XML, содержащий данные,
    // которые должны быть сохранены
    var dataToSave:XML = <gamescore>
                           <username>{_username.text}</username>
                           <score>{score}</score>
                         </gamescore>;

    // Направляем запрос в сценарий, который будет обрабатывать XML
    var request:URLRequest = new URLRequest( "/gamescores.cfm" );
    // Присваиваем свойству data экземпляр XML dataToSave,
    // чтобы отправить данные XML на сервер
    request.data = dataToSave;
    // Задаем contentType, чтобы показать, что передаются данные XML
    request.contentType = "text/xml";
    // Используем метод post для отправки данных
    request.method = URLRequestMethod.POST;

    // Создаем URLLoader для обработки отправки и загрузки данных XML
    var loader:URLLoader = new URLLoader();
    // По завершении загрузки ответа сервера вызываем handleResponse
    loader.addEventListener( Event.COMPLETE, handleResponse );
    // Наконец, отправляем данные XML по заданному URL
    loader.load( request );
```

```
    }

    private function handleResponse( event:Event ):void {
      try {
        // Делаем попытку преобразовать ответ сервера в XML
        var success:XML = new XML( event.target.data );

        // Проверяем значение узла элементов success
        if ( success.toString() == "1" ) {
          _message.text = "Сохранено успешно.";
        } else {
          _message.text = "В процессе сохранения произошла ошибка.";
        }

      } catch ( e:TypeError ) {
        // Выводим сообщение об ошибке, поскольку ответ сервера не понят
        _message.text = "Не удалось разобрать XML-ответ сервера.";
      }
    }
  }
}
```

Свойство `contentType` (тип содержимого) экземпляра *URLRequest* по умолчанию имеет значение `application/x-www-form-urlencoded`, поэтому важно при отправке данных XML присвоить ему значение `text/xml`. Кроме того, чтобы отправлять данные на сервер посредством HTTP-метода `POST`, свойству `method` запроса необходимо присвоить значение *URLRequestMethod.POST*.

Конечно, в реальном примере счет генерировался бы в зависимости от результатов игры. В данном случае требовалось продемонстрировать отправку и прием XML, поэтому счет генерировался случайным образом.

Следующий шаг – создать сценарий, выполняющийся на стороне сервера. Первым приводим вариант на Perl. Если вы будете использовать его, поместите следующий код в текстовый файл *gamescores.cgi* (или *gamescores.pl*) в каталог вашего веб-сервера, для которого активирован доступ к CGI (обычно *cgi* или *cgi-bin*).

```
#!/usr/bin/perl

# Flash/Perl+CGI XML interaction demo
# Arun Bhalla (arun@groogroo.com)

use strict;
use XML::Simple;
use CGI;

my $ScoreFile = "scores.txt";

# Здесь мы предполагаем, что данный сценарий CGI
# получает XML в форме text/xml посредством POST.
# Поэтому XML предоставляется в сценарий через STDIN.
```

```
my $input = XMLin(join('',<STDIN>));

# Отправляем HTTP-заголовок
print CGI::header('text/xml');

# Пытаемся открыть файл счета для записи или возвращаем сообщение об ошибке.
open(SCORES, ">> $ScoreFile") || (printMessage(0) &&
                 die "Ошибка при открытии $ScoreFile");

# Сохраняем счет в текстовом файле с разделяющими символами "|".
print SCORES join('|', $input->{username}, $input->{score}), "\n";

# Возвращаем результат в виде XML.
printMessage(1);

# Подпрограмма для вывода результата в виде XML.
sub printMessage {
  my $value = shift;
  my $message = {};
  $message->{success} = $value;
  print XMLout($message, keeproot => 1, rootname => 'success');
}
```

Следующий блок кода представляет пример сценария на ColdFusion. Поместите данный код в ColdFusion-страницу *gamescores.cfm* в каталог веб-сервера, который поддерживает данный тип приложений:

```
<cfsilent>
<cfsetting enablecfoutputonly="Yes">
<cfset success = 0>
<cftry>
  <!--- пакет XML, посланный Flash. --->
  <cfset scores_xml = XmlParse( getHTTPRequestData(  ).content ) >

  <!--- Синтаксический разбор пакета XML, присланного Flash. --->

  <!--- Из документа XML выбирается username и score и сохраняются как
        локальные переменные, чтобы упростить работу с ними. // --->
  <cfset username = scores_xml.gamescore.username.XmlText >
  <cfset score = scores_xml.gamescore.score.XmlText >

  <!--- Последний счет добавляется в файл счетов. Он также
        может быть сохранен в базе данных или другом документе XML. // --->
  <cffile action="APPEND" file="#ExpandPath( 'scores.txt' )#"
  output="#username#|#score#|#getHTTPRequestData(  ).content#" addnewline="
Yes">
  <cfset success = 1 >
  <cfcatch type="Any">
    <cfset success = 0 >
    <cffile action="APPEND" file="#ExpandPath( 'attempts.txt' )#
                                                    " output="ERROR"
    addnewline="Yes">
  </cfcatch>
</cftry>
</cfsilent>
<cfcontent type="text/xml">
```

```
<cfoutput><?xml version="1.0" ?><success>#success#</success></cfoutput>
<cfsetting showdebugoutput="no" enablecfoutputonly="No">
```

Если на вашем сервере используется PHP, поместите следующий код в PHP-страницу *gamescores.php* в каталог веб-сервера, который сможет обработать запрос к данной странице:

```
<?php

// Читаем XML из Raw Post Data.
$xml = $GLOBALS['HTTP_RAW_POST_DATA'];

// Обрабатываем XML, используя расширение DOM PHP.
$document = xmldoc($xml);

// Читаем корневой элемент <gameinfo>.
$rootElement = $document->root(  );

// Читаем дочерние узлы <username> и <score>.
$childNodes = $rootElement->children(  );

$data = "";

// Перебираем дочерние узлы и помещаем их в массив.
foreach($childNodes as $childNode){
  // Добавляем данные в массив;
  $name = $childNode->tagName(  );
  $value = $childNode->get_content(  );
  $data[$name] = $value;
}

// Добавляем данные в scores.txt ( формат: username|score)
$fp = fopen("scores.txt","a+");
$dataString = $data['username'] . "|" . $data['score'] . "\n";
fputs($fp,$dataString,strlen($dataString));
fclose($fp);

// Возвращаем код успешности операции во Flash
echo "<success>1</success>";
?>
```

У всех трех сценариев есть общая черта: они возвращают XML-ответ с корневым узлом `<success>`, содержащий текстовый узел со значением 1 для обозначения успешности операции сохранения и 0, который является признаком ошибки.

Создав ролик Flash и серверный сценарий, чтобы протестировать их, достаточно запустить ролик и нажать кнопку. Ролик должен получить ответ об успешности операции, а в каталоге сервера, в котором находится сценарий, появится файл *scores.txt*, содержащий данные, предоставленные роликом Flash.

См. также

Рецепты 19.6, 19.7 и 20.11.

20.14. Поиск в XML-документах

Задача

Требуется найти в объекте *XML* узлы или атрибуты, отвечающие определенным критериям.

Решение

Для выбора определенных значений из дерева XML примените к объекту *XML* синтаксис E4X вместе с *условием фильтрации*.

Обсуждение

В данной главе рассмотрен вопрос о том, как синтаксис E4X с объектами *XML* упрощает чтение и запись значений XML-дерева. Кроме того, что E4X прост в применении, он обладает исключительной мощью. Синтаксис E4X аналогичен использованию *XPath* для поиска в документах XML. У тех, кто хорошо знаком с основами *XPath*, работа с расширенными возможностями E4X (такими как фильтрация по условию) не должна вызвать затруднений. Фильтрация по условию позволяет выбирать узлы элементов, отвечающие условию определенного Булева выражения, используя синтаксис `.( условие )`, как будет показано ниже в данном рецепте.

Начнем с создания *XML* из литерала XML:

```
var foodgroup:XML = <foodgroup>
                      <fruits>
                        <fruit color="красный">Яблоко</fruit>
                        <fruit color="оранжевый">Апельсин</fruit>
                        <fruit color="зеленый">Груша</fruit>
                        <fruit color="красный">Арбуз</fruit>
                        <servings>3</servings>
                      </fruits>
                      <vegetables>
                        <vegetable color="красный">Помидор</vegetable>
                        <vegetable color="brown">Картошка</vegetable>
                        <vegetable color="зеленый">Брокколи</vegetable>
                        <servings>2</servings>
                      </vegetables>
                    </foodgroup>;
```

Если имена узлов элементов известны, обратиться к ним можно с помощью оператора «точка». Например, список всех узлов элементов `<fruit>` можно возвратить посредством следующего выражения E4X:

```
var fruitList:XMLList = foodgroup.fruits.fruit;
```

Если требуется какой-то определенный узел `<fruit>`, доступ к нему можно организовать, задавая в квадратных скобках значение его индекса:

```
var theApple:XML = foodgroup.fruits.fruit[0];
```

Если полный путь от корневого узла к искомому узлу (или узлам) неизвестен (или не имеет значения), применяется двойной оператор «точка», выбирающий все соответствующие узлы на любом уровне дерева XML. Приведенная ниже строка кода возвращает все узлы <vegetable> независимо от их положения в иерархии:

```
var vegetableList:XMLList = foodgroup..vegetable;
```

Звездочка (*) – это групповой символ для выбора «любого узла». Следующее выражение E4X присваивает переменной *servings* (порции) значение экземпляра *XMLList*, содержащего все узлы элементов <servings>, являющиеся дочерними элементами любых узлов, которые, в свою очередь, представляют собой дочерние элементы узла <foodgroup> (группа продуктов):

```
var servings:XMLList = foodgroup.*.servings;
```

Знак @ служит для обозначения атрибута. Следующий пример формирует *XMLList*, содержащий значения атрибутов color узлов <fruit>:

```
var colorValues:XMLList = foodgroup.fruits.fruit.@color;
```

Теперь рассмотрим фильтрацию по условию. Фильтрация по условию посредством синтаксиса .(*условие*) выбирает узлы элементов, отвечающие заданному условию. Условие определяется Булевым выражением. Фильтрации подвергается объект *XML* или *XMLList*, предшествующий выражению фильтра.

Допустим, что требуется выбрать все узлы <fruit>, где атрибут color имеет значение красный. Для этого сначала с помощью синтаксиса E4X и оператора «точка» формируется экземпляр *XMLList*, включающий все узлы элементов <fruit>, после чего проводится фильтрация посредством выражения @color == "красный":

```
/* На экран выводится:
<fruit color="красный">Яблоко</fruit>
<fruit color="красный">Арбуз</fruit>
*/
trace( foodgroup..fruit.( @color == "красный" ) );
```

Здесь в выражении, переданном в trace, происходят две вещи:

- Часть foodgroups..fruit возвращает *XMLList* всех узлов элементов <fruit> дерева XML.
- К *XMLList* узлов <fruit> применяется фильтрация по условию и создается новый *XMLList*, содержащий только те элементы <fruit>, которые соответствуют выражению фильтра, в данном случае это элементы, атрибуты color которых имеют значение красный. Как видите, оба узла элементов <fruit>, атрибут color которых равен красный, присутствуют в выводе trace.

Приведенный выше код выбирает узлы, представляющие красные фрукты. А что если требуется выбрать любой узел, атрибут color которого имеет значение красный? На помощь приходит звездочка, обеспечивающая выбор любого узла, в сочетании с фильтром, который проверяет наличие атрибута color, и если color существует, проверяет его значение:

```
/* На экран выводится:
<fruit color="красный">Яблоко</fruit>
<fruit color="красный">Арбуз</fruit>
<vegetable color="красный">Помидор</vegetable>
*/
trace( foodgroup..*.( hasOwnProperty( "@color" ) && @color == "красный" )  );
```

Булево выражение, выступающее в качестве условия, может быть любым выражением, возвращающим Булевы значения true или false. В предыдущем примере метод hasOwnProperty (имеет свойство) проверяет элемент на наличие атрибута color и, если находит его, проверяет его значение. Элемент добавляется в *XMLList*, возвращаемый выражением E4X, только если условие выполняется.

До сих пор фильтрация по условию применялась только к атрибутам. Однако она же позволяет выбирать узел элементов, имеющий текстовый узел с определенным значением. Это особенно полезно, если имеется документ XML с повторяющимися узлами элементов, содержащими дочерние узлы. Например, вот как можно вывести на экран значение атрибута color любого узла элементов <fruit>, узел элемента <name> которого содержит текстовый узел Яблоко:

```
var fruits:XML = <fruits>
                   <fruit color="красный">
                     <name>Яблоко</name>
                   </fruit>
                   <fruit color="оранжевый">
                     <name>Апельсин</name>
                   </fruit>
                   <fruit color="зеленый">
                     <name>Груша</name>
                   </fruit>
                   <fruit color="красный">
                     <name>Арбуз</name>
                   </fruit>
                 </fruits>;

// На экран выводится: красный
trace( fruits.fruit.(name == "Яблоко").@color );
```

Как видите, фильтрация по условию – это довольно мощное средство, особенно в сочетании с регулярными выражениями. В следующем примере с помощью регулярного выражения выбираются все узлы элементов <fruit>, имеющие дочерний узел <name>, в которых содержится текстовый узел, начинающийся с гласной:

```
var fruits:XML = <fruits>
                   <fruit color="красный">
                     <name>Яблоко</name>
                   </fruit>
                   <fruit color="оранжевый">
                     <name>Апельсин</name>
                   </fruit>
                   <fruit color="зеленый">
                     <name>Груша</name>
                   </fruit>
                   <fruit color="красный">
                     <name>Арбуз</name>
                   </fruit>
                 </fruits>;
/* На экран выводится:
<fruit color="красный">
  <name>Яблоко</name>
</fruit>
<fruit color="оранжевый">
  <name>Апельсин</name>
</fruit>
*/
trace( fruits.fruit.( /^[аеиоуэюяАЕИОУЭЮЯ].*/.test( name ) ) );
```

В предыдущем фрагменте кода создается регулярное выражение (между / и /), которое можно прочитать так: «начинается с гласной, в верхнем или нижнем регистре, за которой следует любой символ, повторяющийся любое количество раз». Для регулярного выражения вызывается метод *test( )*, чтобы сравнить его с передаваемым параметром (в данном случае, узлом элемента `<name>`, преобразованным в значение его текстового узла для рассматриваемого элемента `<fruit>`). Поскольку метод *test( )* возвращает Булево значение, его можно использовать как часть Булева условия фильтра.

Регулярные выражения подробно рассмотрены в главе 13.

См. также

Рецепты 20.7, 20.8, 20.9 и главу 13

20.15. Использование HTML и специальных символов в XML

Задача

Требуется использовать HTML или другой текст, содержащий символы, имеющие специальное значение в контексте XML.

Решение

Заключите текст в тег `CDATA`.

Обсуждение

Хотя в данном рецепте речь не идет об ActionScript как таковом, он имеет отношение к делу, потому что затрагивает вопросы, с которыми разработчики наверняка столкнутся при работе с XML во Flash или в других средствах разработки. В XML определенные символы имеют специальное значение. Например, знаки «больше чем» и «меньше чем» обозначают границы тегов XML. Любой из этих символов в тексте XML-документа породит ошибки при его синтаксическом разборе. Например:

```
<example>a < b</example>
```

Как видите, здесь текстовый узел содержит текст `a < b`. Но поскольку знак «меньше чем» имеет специальное значение в контексте документа XML, данное выражение приводит к ошибке синтаксического анализатора. Другой распространенный пример – размещение HTML в XML-документе; например:

```
<htmlExample><a href="http://www.darronschall.com">Деррон</a></htmlExample>
```

В этом случае HTML трактуется как XML, а не как строка. В предыдущий пакет XML входят корневой узел `<htmlExample>` с дочерним узлом `<a>`, содержащим текстовый узел со строковым значением `Деррон`.

В подобном случае текст можно заключить в тег `CDATA` (Character Data – символьные данные), который не обрабатывается анализатором XML. Он просто воспринимает эти данные как строку.

Тег `CDATA` начинается с `<![CDATA[` и заканчивается `]]>`. Следовательно, первый пример можно изменить так:

```
<example><![CDATA[a < b]]></example>
```

Второй пример можно переписать следующим образом:

```
<htmlExample><![CDATA[<a href="http://www.darronschall.com">Деррон</a>]]></
htmlExample>
```

При синтаксическом разборе XML символьные данные, заключенные в тег `CDATA`, трактуются как одна строка, а сам тег `CDATA` не рассматривается. Таким образом, в первом примере получаем текстовый узел `a < b`, а не `<![CDATA[a < b]]>`. Во втором примере пакет XML интерпретируется как корневой узел `<htmlExample>` с одним текстовым узлом, строковым значением которого является вся HTML-ссылка. Из-за тега `CDATA` `<a>` рассматривается не как дочерний узел, как это было раньше, а как часть строкового значения текстового узла.

21

Веб-сервисы и удаленное взаимодействие во Flash

21.0. Введение

Удаленные вызовы процедур (Remote procedure calls, RPC) являют собой мощное средство построения распределенных и надежных приложений. RPC задействуются в большинстве приложений Flash-платформы. RPC позволяют отправлять данные из Flash Player на сервер или извлекать данные с сервера и отображать их пользователю. Есть масса различных способов удаленного вызова процедур из Flash Player, но среди них особо выделяются веб-сервисы и Flash Remoting.

Говоря о веб-сервисах в данной книге, основное внимание мы сосредотачиваем на простом протоколе доступа к объектам (Simple Object Access Protocol, SOAP). Веб-сервисы используют SOAP для обмена информацией между сервером и клиентом. SOAP может сериализовать сложные типы данных. Это означает, что можно вызывать серверные методы из клиента и передавать в них параметры как простых типов (числа, строки и Булевы значения), так и сложных (объекты). Серверный метод может возвращать клиенту сложные данные, такие как массивы, объекты дат и даже специальные типы данных. SOAP веб-сервисы поддерживаются практически всеми платформами, среди которых можно назвать Java, ColdFusion, PHP, .NET и Perl. Однако Flash Player не имеет встроенной поддержки веб-сервисов и SOAP.

Flash Player может обмениваться информацией посредством HTTP и проводить синтаксический разбор данных XML. Поскольку SOAP веб-сервисы обмениваются информацией через HTTP, а SOAP – это основанный на XML протокол, ActionScript может применяться для вызова методов веб-сервисов.

Технология Flash Remoting очень близка к веб-сервисам, но обладает несколькими существенными преимуществами:

- Flash Remoting тоже использует HTTP, но вместо SOAP здесь применяется протокол двоичной синхронной передачи данных Active Messaging Format (AMF). Поскольку AMF-пакеты двоичные, этот формат обеспечивает возможность посылать намного больше данных с меньшими затратами. Поэтому Flash Remoting обеспечивает более быстрый обмен информацией, чем другие веб-сервисы.
- Flash Remoting поддерживается Flash Player.
- Flash Remoting также доступна для Java, ColdFusion, .NET и Perl. Однако есть платформы, не такие распространенные на сегодняшний день, которые поддерживают веб-сервисы и не поддерживают Flash Remoting.

И Flash Remoting, и веб-сервисы осуществляют асинхронные вызовы методов сервиса и могут применяться для создания сложных приложений клиент-сервер. Рецепты данной главы рассматривают работу с этими технологиями.

21.1. Вызов методов веб-сервисов

Задача

Требуется вызвать метод веб-сервиса.

Решение

Вызывайте метод посредством объекта *mx.rpc.soap.WebService*.

Обсуждение

Как упоминалось ранее, Flash Player не имеет встроенной поддержки веб-сервисов, но инфраструктура Flex предоставляет решение этой проблемы. В данном рецепте показано, как организовать работу с веб-сервисами при помощи решения, основанного на Flex 2, а вариант с платформой Flash можно найти по адресу *http://www.rightactionscript.com/ascb*.

Инфраструктура Flex включает *mx.rpc.soap.WebService* – класс, упрощающий вызов методов веб-сервисов в ActionScript. В первую очередь создадим новый объект *WebService*:

```
var webService:WebService = new WebService();
```

У каждого веб-сервиса должен быть ресурс в виде данных на языке описания веб-сервисов WSDL (Web Service Description Language). С помощью свойства `wsdl` можно указать объекту *WebService*, где располагается WSDL:

```
webService.wsdl = "http://www.rightactionscript.com/webservices/
FlashSurvey.php?wsdl";
```

Прежде чем вызывать методы, необходимо загрузить данные WSDL по указанному URL методом *loadWSDL( )*:

```
webService.loadWSDL();
```

Метод *loadWSDL( )* осуществляет запрос данных WSDL асинхронно. То есть, чтобы знать, когда данные WSDL загружены, необходимо слушать соответствующее событие. Когда данные загружены, объект *WebService* рассылает событие `load` типа *mx.rpc.soap.LoadEvent*:

```
webService.addEventListener(LoadEvent.LOAD, onWSDL);
```

Как только данные WSDL корректно загружены, можно вызывать методы веб-сервиса непосредственно из объекта *WebService*. Например, используемый в данном примере URL WSDL указывает на реальный веб-сервис, имеющий метод *getAverages( )* (получить среднее), который можно вызвать из объекта *WebService*:

```
webService.getAverages();
```

Если метод принимает параметры, они передаются в него при вызове, как это делается для обычных методов ActionScript. Например, в том же веб-сервисе есть метод *takeSurvey( )*, который ожидает два целых значения. В следующей строке при вызове этого метода в него передаются значения 10 и 15:

```
webService.takeSurvey(10, 15);
```

Вызовы методов веб-сервисов происходят асинхронно, это означает, что ответ не возвращается немедленно. В рецепте 21.2 подробнее рассказано о том, как получить значение свойства `return` метода веб-сервиса.

См. также

Рецепт 21.2.

21.2. Обработка ответов веб-сервисов

Задача

Требуется получить возвращаемое значение метода веб-сервиса.

Решение

Добавьте слушателя события `result` в объект метода веб-сервиса.

Обсуждение

Методы веб-сервисов – это сложные объекты типа *mx.rpc.soap.Operation*, для которых можно определять слушателей событий. Когда метод веб-сервиса возвращает значение, объект метода формирует событие `result` (результат) типа *mx.rpc.events.ResultEvent*. Если требуется

обработать это событие, в объект метода добавляется слушатель для него. Например, если webService – это объект *WebService*, который соответствует веб-сервису, определяющему метод *getAverages( )*, то слушатель добавляется так:

```
webService.getAverages.addEventListener(ResultEvent.RESULT, onGetAverages);
```

Затем можно вызывать метод обычным способом:

```
webService.getAverages();
```

При вызове слушателя в него передается параметр *ResultEvent* (событие «результат»). Класс *ResultEvent* определяет свойство result, содержащее значение return (возврат). Предположим, метод веб-сервиса *getAverages( )* возвращает ассоциативный массив с двумя свойствами, flash и actionscript. Следующий фрагмент кода обеспечивает вывод их значений в текстовой области textArea:

```
private function onGetAverages(event:ResultEvent):void {
    textArea.text = "Среднее для Flash и ActionScript – " +
event.result.flash + " и " + event.result.actionscript;
}
```

21.3. Обработка ошибок веб-сервисов

Задача

Требуется обработать ошибки веб-сервиса.

Решение

Слушайте событие fault.

Обсуждение

Когда происходит ошибка веб-сервиса, операция формирует событие fault (сбой) типа *mx.rpc.events.FaultEvent*. Слушателя этого события можно добавить непосредственно в объект *Operation* (операция). Однако рекомендуется обрабатывать события сбоя на уровне объекта *WebService*. Если события fault операции не обрабатываются на уровне *Operation*, то передаются вверх по иерархии объектов (bubble up) в *WebService*. Следующий код добавляет слушателя события в объект *WebService*:

```
webService.addEventListener(FaultEvent.FAULT, onWebServiceFault);
```

Класс *FaultEvent* определяет свойство fault типа *mx.rpc.Fault*. Объекты *Fault* в своих свойствах faultCode (код сбоя), faultDetail (детали сбоя), faultString (строка сбоя) и rootCause (причина сбоя) возвращают информацию об ошибке. Следующий код обеспечивает отображение предупреждения (*Alert*) при возникновении ошибки:

```
private onWebServiceFault(event:FaultEvent):void {
    var fault:Fault = FaultEvent.fault;
    var message:String = "Произошла ошибка. Информация об ошибке\nкод: "
                                                + fault.faultCode;
    message += "\nдетали: " + faul.faultDetail;
    Alert.show("Ошибка веб-сервиса", message);
}
```

21.4. Flash Remoting: вызов методов

Задача

Требуется вызвать метод Flash Remoting.

Решение

Задействовав класс *NetConnection*, соединитесь с сетевым интерфейсом Flash Remoting и вызывайте метод с помощью метода *call( )*.

Обсуждение

Во Flex и Flash есть основанные на ActionScript программные интерфейсы, призванные упростить вызов методов Flash Remoting. Однако в данном рецепте обсуждается низкоуровневое решение, эксплуатирующее только встроенные возможности Flash Player.

Класс *flash.net.NetConnection* используется всеми операциями Flash Remoting. Первый шаг при вызове метода Flash Remoting – создать новый объект *NetConnection*. Каждому сервису Flash Remoting (обычно сервис – это класс) должен соответствовать отдельный объект *NetConnection*:

```
var connection:NetConnection = new NetConnection();
```

Далее вызываем метод *connect( )*, передавая в него URL сетевого интерфейса Flash Remoting. Например, к приложению на ColdFusion, размещенному на локальном сервере, можно обратиться так:

```
connection.connect("http://localhost:8500/flashservices/gateway/");
```

Приведенная ниже строка кода обеспечивает соединение с сетевым интерфейсом Flash Remoting рабочего примера AMFPHP:

```
connection.connect("http://www.rightactionscript.com/flashremoting/
gateway.php");
```

Если URL сетевого интерфейса неизвестен, то надо обратиться или к документации серверного продукта, реализующего Flash Remoting, или к своему сетевому администратору, отвечающему за серверную часть Flash Remoting.

Метод *connect( )* не сразу делает попытку соединиться с сетевым интерфейсом по заданному URL. Если URL недействительный или если

возникает ошибка на стороне сервера, ошибка не будет возвращена до тех пор, пока не произойдет фактический вызов метода.

Следующий шаг – вызвать метод Flash Remoting посредством метода *call( )* объекта *NetConnection*. Метод *call( )* принимает два обязательных параметра. Первый определяет имя и путь сервиса и метода. Второй указывает объекту *NetConnection*, как обрабатывать ответ, возвращаемый в результате вызова метода. Как обрабатываются ответы сервиса Flash Remoting, рассказывает рецепт 5. В данном предварительном обсуждении предполагается, что приложению не надо обрабатывать ответ; таким образом, для второго параметра передается значение `null`:

Первый параметр метода *call( )* определяет вызываемый метод, специфицируя разделенный точками путь к нему в таком формате:

```
пакет.ОбъектСервиса.метод
```

Например, если используется компонент ColdFusion (CFC), доступный по адресу *http://localhost:8500/com/oreilly/as3cb/Example.cfc*, и CFC определяет метод *test( )*, то вызов этого метода осуществляется следующим образом:

```
connection.call("com.oreilly.as3cb.Example.test", null);
```

Для того чтобы передать параметры в вызываемый метод, надо добавить их в список параметров метода *call( )*. Например, если метод *test( )* принимает числовой и строковый параметры, то передать их значения можно так:

```
connection.call("com.oreilly.as3cb.Example.test", null, 15, "abcd");
```

См. также

Рецепты 21.2 и 21.5.

21.5. Flash Remoting: обработка ответов

Задача

Требуется получить значение `return` метода Flash Remoting.

Решение

Задействуйте объект *Responder* с методом *call( )*.

Обсуждение

В рецепте 4 показано, как с помощью метода *call( )* объекта *NetConnection* вызвать метод Flash Remoting. Второй параметр этого метода предназначен для определения того, как Flash Player должен обрабатывать ответы. Если передается значение `null` (как в примерах того ре-

цепта), ответы не обрабатываются. Если ответы необходимо обработать, задействуется объект *flash.net.Responder*.

Конструктор *Responder* (ответчик) позволяет передавать в него две ссылки на функции, которые обрабатывают значения `return` и ошибки:

```
var responder:Responder = new Responder(onResult, onError);
```

В вызываемую функцию обработки результата передается один параметр – значение, возвращенное методом Flash Remoting:

```
private function onResult(returnValue:Datatype):void {
}
```

В метод-обработчик ошибок передается объект, свойства которого подробно описывают возникшую ошибку.

В следующем примере вызывается метод Flash Remoting и для отображения значений применяется метод *trace( )*. Метод *getAverages( )* возвращает ассоциативный массив с двумя свойствами, `flash` и `actionscript`:

```
package {
    import flash.net.NetConnection;
    import flash.net.Responder;

    public class Example {
        private var _connection:NetConnection;
        public function Example() {
            _connection = new NetConnection();
            _connection.connect("http://www.rightactionscript.com/
                                                        flashremoting/
                gateway.php");
            var responder:Responder = new Responder(onResult, onError);
            _connection.call("FlashSurvey.getAverages", responder);
        }

        private function onResult(result:Object):void {
            trace(result.flash + " " + result.actionscript);
        }

        private function onError(error:Object):void {
            trace(error.description);
        }
    }
}
```

Типы значений, возвращаемые методами Flash Remoting, всегда должны быть хорошо сериализуемыми. Поэтому если требуется возвратить наборы записей, их необходимо преобразовывать в ассоциативные массивы. Это обеспечит наилучшую функциональную совместимость.

См. также

Рецепт 21.4.

22

Создание интегрированных приложений

22.0. Введение

Класс *ExternalInterface* (внешний интерфейс) обеспечивает возможность Flash-проигрывателю синхронно обмениваться информацией с приложением, в которое он встроен. Во многих случаях роль приложения-оболочки выполняет веб-броузер. Собственно, данная глава всецело посвящена построению приложений, объединяющих ActionScript и JavaScript, когда Flash Player встроен в броузер.

22.1. Вызов функций JavaScript

Задача

Требуется вызвать из ActionScript функцию JavaScript.

Решение

Это делается при помощи метода *ExternalInterface.call( )*.

Обсуждение

Синхронные вызовы функций JavaScript из ActionScript реализуются посредством метода *ExternalInterface.call( )*, принимающего один обязательный параметр – строку с именем вызываемой функции:

```
ExternalInterface.call("changeTitle");
```

На HTML-странице должна быть определена одноименная функция:

```
<script language="JavaScript">
    function changeTitle(title) {
        if(title == undefined) {
            title = "Новый заголовок";
        }
```

```
        window.title = title;
    }
</script>
```

Если функция JavaScript принимает параметры, то их можно передать в нее как дополнительные при вызове метода *call( )*. Так, ниже передается значение в функцию *changeTitle( )* (изменить заголовок):

```
ExternalInterface.call("changeTitle", "ActionScript 3.0 Cookbook");
```

Поскольку *call( )* – это синхронный метод, все значения, возвращаемые функцией JavaScript, немедленно возвращаются в ActionScript. Так возвращаемое значение можно присвоить переменной:

```
var title:String = ExternalInterface.call("getTitle");
```

Функция JavaScript для данного вызова ActionScript могла бы выглядеть следующим образом:

```
<script language="JavaScript">
    function getTitle() {
        return window.title;
    }
</script>
```

Класс *ExternalInterface* применим в следующих броузерах:

- Internet Explorer 5.0+ (Windows)
- Netscape 8.0+ (Windows и Mac OS X)
- Mozilla 1.7.5+ (Windows и Mac OS X)
- Firefox 1.0+ (Windows и Mac OS X)
- Safari 1.3+ (Mac OS X)

Если броузер не поддерживает *ExternalInterface*, то все равно существует возможность вызывать функции JavaScript. Для этого предназначена функция *flash.net.navigateToURL( )*.

Функция *navigateToURL( )*асинхронная, т. е. она не возвращает значение. Чтобы с помощью *navigateToURL( )* вызвать функцию JavaScript, необходимо задействовать объект *flash.netURLRequest*, который имеет значение, использующее протокол `javascript`. Ниже приведен пример вызова функции JavaScript *alert( )*:

```
var request:URLRequest = new URLRequest("javascript:alert('example');");
navigateToURL(request);
```

См. также

Рецепт 22.2.

22.2. Вызов функций ActionScript

Задача

Требуется вызвать функцию ActionScript из JavaScript.

Решение

С помощью метода *ExternalInterface.addCallback( )* зарегистрируйте функцию ActionScript и затем вызывайте ее из JavaScript по ссылке на объект Flash Player.

Обсуждение

Программный интерфейс *ExternalInterface* позволяет регистрировать функции ActionScript, что делает их доступными для JavaScript. Для регистрации функций ActionScript применяется статический метод *addCallback( )*, принимающий два обязательных параметра: имя, по которому эта функция вызывается из JavaScript, и ссылку на функцию. В следующем примере регистрируется функция `displayMessage` (вывести сообщение) с использованием идентификатора (имя, по которому эта функция может вызываться из JavaScript) `showMessage` (показать сообщение):

```
ExternalInterface.addCallback("showMessage", displayMessage);
```

Из JavaScript необходимо получить ссылку на объект Flash Player. Существует две основные версии Flash Player для воспроизведения Flash в броузерах: ActiveX-версия и подключаемая версия. Internet Explorer задействует ActiveX-компонент, тогда как все остальные броузеры используют подключаемую версию.

Для описания ActiveX-проигрывателя в HTML-странице используется тег `<object>`. Ссылку на него в JavaScript можно получить посредством выражения `window.`*`objectId`*, где *`objectId`* – значение атрибута `id` тега `<object>`. Например, если атрибут `id` тега `<object>` имеет значение `example`, то ссылка на ActiveX-проигрыватель записывается как `window.example`.

Подключаемый проигрыватель описывается в HTML-странице тегом `<embed>`. Получить JavaScript-ссылку можно через `window.document.`*`embedName`*, где *`embedName`* (встроенное имя) – значение атрибута `name` тега `<embed>`. Например, если атрибут `name` тега `<embed>` имеет значение `example`, то ссылка на подключаемый проигрыватель записывается как `window.document.example`.

В большинстве случаев версия Flash Player, установленная у пользователя, неизвестна. Определить броузер пользователя в JavaScript можно с помощью свойства `navigator.appName`:

- Если `navigator.appName` возвращает строку `Microsoft`, пользователь работает с Internet Explorer, т. е. задействуется ActiveX-проигрыватель.
- Если `navigator.appName` не возвращает строку `Microsoft`, значит, пользователь работает с подключаемой версией проигрывателя.

Следующий код на JavaScript позволяет определить версию проигрывателя и получить соответствующую ссылку на него:

```
<script language="JavaScript">
```

```
var flashPlayer;
function detectFlashPlayer() {
    if(navigator.appName.indexOf("Microsoft") != -1) {
        flashPlayer = window.objectId;
    }
    else {
        flashPlayer = window.document.embedName;
    }
}
</script>
```

Далее, чтобы вызвать метод *detectFlashPlayer( )* при загрузке страницы, в теге `<body>` задается атрибут `onLoad`, например:

```
<body onLoad="detectFlashPlayer">
```

Переменная `flashPlayer` содержит корректную ссылку на Flash Player, установленный в системе пользователя.

Любую зарегистрированную функцию ActionScript можно вызывать прямо по ссылке на Flash Player. Так, если функция зарегистрирована с идентификатором `showMessage`, следующей строкой кода эта функция вызывается из JavaScript (предполагается, что `flashPlayer` – переменная, указывающая на Flash Player):

```
flashPlayer.showMessage();
```

Также можно передавать параметры в функции. Если функция ActionScript, имеющая в *ExternalInterface* идентификатор `showMessage`, принимает строковый параметр, то из JavaScript ее можно вызвать так:

```
flashPlayer.showMessage("пример сообщения");
```

См. также

Рецепт 22.1.

22.3. Передача параметров из HTML

Задача

Требуется передать параметры из HTML в SWF.

Решение

Укажите параметр `FlashVars`.

Обсуждение

`FlashVars` предоставляет возможность передавать параметры в SWF из HTML. Это может пригодиться для передачи простых данных в SWF, когда данные меняются в зависимости от того, где располагается приложение. Например, требуется передать в SWF URL веб-сервисов, и эти URL меняются в зависимости от того, на каком сервере располагается

SWF. Это позволяет повторно размещать SWF с новыми значениями без необходимости повторной компиляции SWF.

Решение, основанное на FlashVars, состоит из двух частей: одна часть реализуется в HTML, а вторая – в ActionScript.

HTML-часть решения FlashVars требует добавления тега <param name="FlashVars"> в тег <object> и атрибута FlashVars в тег <embed>. Значение тега и атрибута – это URL-кодированная последовательность пар имя-значение. Например, следующий код представляет действительное значение FlashVars, определяющее две пары имя-значение, url1 и url2:

```
url1=http://www.example.com&url2=http://www.sample.com
```

В ActionScript ссылаться на переменные и значения, переданные посредством FlashVars, можно через свойство root.loaderInfo.parameters любого отображаемого объекта. Свойство root.loaderInfo.parameters – это ассоциативный массив, ключи которого являются именами переменных, переданных в SWF через FlashVars. Например, если взять значение из приведенного выше примера, свойство root.loaderInfo.parameters будет иметь два ключа: url1 и url2.

FlashVars в сочетании с JavaScript позволяет передавать строку запроса из HTML-страницы в SWF. Следующий код JavaScript иллюстрирует, как с помощью JavaScript создать теги <object> и <embed>, чтобы они передавали данные строки запроса из HTML-страницы в SWF, используя FlashVars:

```
// Извлекаем строку запроса и присваиваем ее переменной.
var parameters = window.location.search.substr(1);
var objectEmbed = '<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-
444553540000" id="Example" width="100%" height="100%" codebase="http://
download.macromedia.com/pub/shockwave/cabs/flash/ swflash.cab">';
objectEmbed += '<param name="movie" value="Example.swf" />';
objectEmbed += '<param name="quality" value="high" />';
objectEmbed += '<param name="bgcolor" value="#869ca7" />';
objectEmbed += '<param name="allowScriptAccess" value="sameDomain" />';
objectEmbed += '<param name="FlashVars" value="' + parameters + '" />';
objectEmbed += '<embed src="Example.swf" quality="high" bgcolor="#869ca7"
width="100%" height="100%" name="Example" align="middle" play="true" loop="
false" quality="high" allowScriptAccess="sameDomain" type="application/x-
shockwave-flash" pluginspage="http://www. macromedia.com/go/getflashplayer"
FlashVars="' + parameters + '"></embed>';
objectEmbed += '</object>';
document.write(objectEmbed);
```

Параметр FlashVars – это мощное средство Flash Player. Однако важно правильно его применять. FlashVars не является универсальным механизмом передачи данных в SWF. Он разработан только для работы с простыми данными, которые могут меняться в зависимости от того, где размещается SWF. Для загрузки во время выполнения более сложных настроечных данных надо работать с объектом *URLLoader*.

23

Работа с файлами

23.0. Введение

До версии 8 Flash Player не поддерживал никакого механизма, обеспечивающего пользователю возможность просматривать файлы на своем компьютере. Более того, Flash Player не имел средства загрузки и передачи файлов. Поэтому в большинстве веб-приложений для этих целей применялись решения, основанные на HTML. Оффлайновым Flash-приложениям для передачи и загрузки файлов часто приходилось прибегать к специальным решениям. Flash Player 8 и последующие версии поддерживают новые программные интерфейсы, позволяющие Flash Player просматривать файлы системы пользователя, что существенно упрощает передачу и загрузку файлов.

Возможность просматривать файлы на локальных дисках, а также передавать и загружать файлы во Flash Player обеспечивают классы *FileReference* и *FileReferenceList*. В данной главе подробно рассматривается работа с этими API.

23.1. Загрузка файлов

Задача

Требуется обеспечить пользователям возможность загружать файлы на их компьютер.

Решение

Задача решается посредством метода *download( )* объекта *FileReference*.

Обсуждение

Класс *flash.net.FileReference* определяет метод *download( )* (загрузить), посредством которого пользователи могут загружать файл на свой компьютер по заданному URL. При вызове метода *download( )* Flash Player открывает диалоговое окно «Select location for download» (Выбрать место для загрузки). Это диалоговое окно предоставляет возможность использовать стандартный диалог операционной системы для просмотра локальных дисков и выбора места для сохранения файлов.

Первый шаг перед вызовом метода *download( )* – создание объекта *FileReference* (ссылка на файл):

```
var fileReference:FileReference = new FileReference();
```

Метод *download( )* принимает один обязательный параметр – объект *URLRequest*, определяющий URL, по которому требуется загрузить файл. В самом простом случае URL указывает на статический файл, который может быть загружен пользователем. Следующий пример открывает диалоговое окно, в котором пользователь может выбрать место для сохранения копии файла *example.txt*, находящегося на сервере в том же каталоге, что и SWF:

```
var urlRequest:URLRequest = new URLRequest("example.txt");
fileReference.download(urlRequest);
```

При создании объекта *URLRequest* можно задать и полный путь к файлу:

```
var urlRequest:URLRequest = new URLRequest("http://www.myexamplesite.com
                                                        /example.txt");
fileReference.download(urlRequest);
```

В диалоговом окне сохранения можно также изменять имя загружаемого файла. По умолчанию в поле имени файла отображается то имя, под которым файл хранится на сервере (предыдущем примере – *example.txt*). Однако если статический файл был создан сценарием, выполняющимся на стороне сервера, то в качестве имени будет выступать уникальный идентификатор (например, *R7AS82892KHWI014.jpg*), который непонятен пользователю. Кроме того, файлы могут создаваться сценарием динамически во время запроса, и тогда объект *URLRequest* указывает на этот сценарий. В этом случае значение по умолчанию, отображаемое в поле имени файла, соответствует имени сценария (например, *script.cgi*), что не только неудобно для пользователя, но также может привести к неправильному толкованию файла системой Windows из-за несоответствующего расширения файла.

Метод *download( )* принимает второй параметр (необязательный), которым можно определить для файла другое имя по умолчанию, отображаемое в диалоговом окне сохранения. Следующий пример предлагает пользователю загрузить файл, динамически создаваемый сце-

нарием (*script.cgi*) во время запроса. Сценарий возвращает файл изображения JPEG. Вместо имени файла по умолчанию (*script.cgi*), код определяет для файла имя *example.jpg*:

```
var urlRequest:URLRequest = new URLRequest("script.cgi");
fileReference.download(urlRequest, "example.jpg");
```

Работая с методом *download( )*, не обойтись без блока `try...catch`, потому что метод может формировать ошибки. Необработанная ошибка приводит к сбою (если код предполагает, что операция проведена успешно) и выводу стандартного сообщения об ошибке. Все ошибки надо обрабатывать, потому стандартное сообщение заставляет пользователя думать, что что-то сломалось. Если ошибка обрабатывается в коде ActionScript, приложение реагирует на нее намного изящнее, чем на необработанную. Чаще всего возникают ошибки *IllegalOperationError* (недопустимая операция) и *SecurityError* (нарушение безопасности). Ошибка *IllegalOperationError* возникает, когда вызов *download( )* осуществляется при уже открытом диалоге для сохранения. Одновременно может быть открыто только одно такое диалоговое окно. Ошибка *SecurityError* возникает, когда у SWF нет прав на загрузку файла, потому что SWF – это ненадежный файл:

```
try {
    fileReference.download(urlRequest, fileName);
}
catch (illegalOperation:IllegalOperationError) {
    // код для обработки ошибки недопустимой операции
}
catch (security:SecurityError) {
    // код для обработки ошибки безопасности
}
```

Хотя не так часто, но метод *download( )* может также формировать ошибки типа *ArgumentError* (недопустимый аргумент) и *MemoryError* (ошибка распределения памяти). Ошибки *ArgumentError* возникают, когда значение свойства `data` параметра *URLRequest* имеет тип, отличный от *URLVariables* или *String*. Хотя *URLRequest* допускает использование байтового массива в качестве значения свойства `data`, метод *download( )* работает только с типами *URLVariables* или *String*. Ошибка распределения памяти возникает в случае выполнения одного из двух условий: если *URLRequest* настроен на осуществление операции `GET` и `System.useCodePage` имеет значение `true`, Flash Player не может преобразовать символы из Unicode в формат многобайтовых символов, как это требуется. Или Flash Player не может выделить достаточный объем памяти для открытия диалогового окна сохранения.

Метод *download( )*, не только формирует ошибки, но и рассылает события ошибок. Если Flash Player не смог загрузить файл по причинам нарушения безопасности, метод *download( )* рассылает событие нарушения безопасности типа *SecurityErrorEvent*. Такие события возникают,

если безопасная изолированная программная среда Flash Player не предоставила SWF разрешения на загрузку файла по заданному URL. Метод *download( )* может рассылать событие ошибки ввода/вывода типа *IOErrorEvent*, если произошел сбой при загрузке файла по одной из следующих причин:

- Ошибка сети.
- Сервер требует аутентификации, и Flash Player выполняется не в броузере.
- Недопустимый протокол (разрешены только HTTP, HTTPS и FTP).

Для обработки событий ошибок, таких как *SecurityErrorEvent* и *IOErrorEvent*, необходимо добавить слушателей. Это делается так же, как и для обычных событий. В следующем примере добавляется слушатель событий типа *IOErrorEvent*.

```
fileReference.addEventListener(IOErrorEvent.IO_ERROR, onIOError);
```

См. также

Рецепт 23.2.

23.2. Определение момента выбора файла пользователем

Задача

Чтобы начать загрузку, требуется отследить момент выбора файла пользователем.

Решение

Слушайте событие `select`. Слушайте событие `cancel`, чтобы определить момент нажатия пользователем кнопки Cancel (Отменить).

Обсуждение

Метод *download( )* не прерывает выполнение кода ActionScript. При вызове *download( )* Flash Player делает попытку открыть диалоговое окно для сохранения. Независимо от того, удачно ли открыт диалог или сформирована ошибка, Flash Player продолжает выполнение со следующей строки кода ActionScript. То есть не ожидается, что пользователь выберет файл и нажмет кнопку Save сразу после вызова метода *download( )*. Чтобы установить факт нажатия кнопки, мы слушаем событие `select`. Событие `select` имеет тип *Event*; чтобы добавить слушателя этого события, можно использовать константу `Event.SELECT`:

```
fileReference.addEventListener(Event.SELECT, onSelectFile);
```

Когда событие `select` произошло, можно извлекать имя выбранного пользователем файла, читая свойство `name` объекта *FileReference*:

```
private function onSelectFile(event:Event):void {
    trace(event.target.name);
}
```

Пользователь также может щелкнуть по кнопке Cancel диалогового окна сохранения. В этом случае файл не загружается, а диалоговое окно закрывается. Когда пользователь нажимает кнопку Cancel, объект *FileReference* рассылает событие `cancel` типа *Event*. Для добавления слушателя этого события можно использовать константу `Event.CANCEL`:

```
fileReference.addEventListener(Event.CANCEL, onCancelDialog);
```

См. также

Рецепт 23.1.

23.3. Отслеживание процесса загрузки

Задача

Требуется отслеживать процесс загрузки.

Решение

Слушайте событие `progress`.

Обсуждение

Процесс загрузки файла(-ов) можно отслеживать, работая с событием `progress`. Каждый раз при загрузке определенной части файла на компьютер пользователя объект *FileReference* формирует событие `progress` типа *ProgressEvent*. Для добавления слушателя этого события используется константа `ProgressEvent.PROGRESS`:

```
fileReference.addEventListener(ProgressEvent.PROGRESS, onFileProgress);
```

Событие `progress` имеет два свойства, `bytesLoaded` и `bytesTotal`, возвращающие количество загруженных байтов и общее количество байтов соответственно. В следующем примере значения этих свойств анализируются для отображения процесса загрузки в текстовом поле `fileProgressField` (поле процесса загрузки файла):

```
private function onFileProgress(event:ProgressEvent):void {
    fileProgressField.text = event.bytesLoaded + " из " + event.bytesTotal
                                                    + " байтов";
}
```

По завершении загрузки файла объект *FileReference* рассылает событие `complete` типа *Event*. Для добавления слушателя этого события используется константа `Event.COMPLETE`:

```
fileReference.addEventListener(Event.COMPLETE, onFileComplete);
```

См. также

Рецепт 23.1.

23.4. Просмотр файлов

Задача

Требуется предоставить пользователю возможность просматривать локальный(-е) диск(-и) и выбирать файл(-ы) для передачи.

Решение

Обратитесь к методу *browse( )* объекта *FileReference* или *FileReferenceList*.

Обсуждение

Метод *browse( )* объектов *FileReference* или *FileReferenceList* позволяет открывать диалоговое окно, в котором можно просматривать локальные диски в поисках файла или файлов. Методы *browse( )* классов *FileReference* и *FileReferenceList* отличаются тем, что первый позволяет выбирать только один файл, тогда как второй – несколько файлов, при нажатой клавише Control или Shift (или Command на Макинтоше):

```
fileReference.browse();
```

Как и метод *download( )*, *browse( )* может формировать ошибки, которые лучше обрабатывать с помощью блоков `try/catch`. Ошибка недопустимой операции возникает в одном из двух случаев:

- Одновременно может быть открыто только одно диалоговое окно просмотра файлов. Если такой диалог уже открыт и происходит вызов *browse( )*, формируется ошибка типа *IllegalOperationError*.
- Если пользователь в глобальных настройках Flash Player запретил просмотр файлов, вызов метод *browse( )* приведет к формированию ошибки.

Данный фрагмент кода иллюстрирует применение блока `try/catch` для обработки ошибки *IllegalOperationError*, сформированной при выборе метода *browse( )*.

```
try {
    fileReference.browse();
}
catch (illegalOperation:IllegalOperationError) {
    // код обработки ошибки
}
```

Обработка ошибки гарантирует, что Flash Player не будет отображать стандартное сообщение об ошибке (что может приводить пользователя в замешательство).

См. также

Рецепт 23.5.

23.5. Фильтрация файлов, которые будут отображаться в окне просмотра файлов

Задача

Требуется фильтровать типы файлов, отображаемые в диалоговом окне, чтобы пользователь мог видеть только те файлы, которые разрешено передавать.

Решение

Передайте массив объектов *FileFilter* в метод *browse( )*.

Обсуждение

По умолчанию метод *browse( )* выводит диалоговое окно, отображающее все файлы файловой системы пользователя. Однако можно фильтровать отображаемые типы файлов. Например, требуется показывать только файлы изображений или только текстовые файлы. Это можно реализовать, передавая в метод *browse( )* массив объектов *flash.net.FileFilter*.

Конструктор *FileFilter* принимает два обязательных параметра:

- Первый определяет, что отображается в выпадающем меню «Files of type» (файлы типа) диалогового окна просмотра файлов.
- Второй определяет, какие расширения файлов отбираются.

Расширения файлов, записанные со звездочкой и точкой впереди, должны быть перечислены через точку с запятой. Следующий код создает объект *FileFilter*, выбирающий файлы с расширениями *.png*, *.gif* и *.jpg*:

```
var fileFilter:FileFilter = new FileFilter("Images", "*.png;*.gif;*.jpg");
```

Кроме того, можно задать третий параметр для типов файлов на Макинтоше. Как и расширения файлов, типы файлов перечисляются через точку с запятой. Если параметр типов файлов Макинтоша опущен, то расширения файлов выступают в качестве фильтра для файлов и Windows и Mac OS. Если же этот параметр задан, то расширения файлов служат для фильтрации файлов Windows, но игнорируются для файлов Макинтоша, выбор которых осуществляется согласно типам файлов Макинтоша. Сейчас операционная система Mac OS поддерживает расширения файлов как способ определения типа файла, но также продолжает использовать коды типов файлов.

Фильтр (или фильтры) можно задать, передавая их в метод *browse( )* как элементы массива:

```
fileReference.browse([fileFilter]);
```

В предыдущем примере в метод *browse( )* передается только один фильтр. Каждый передаваемый в метод *browse( )* фильтр добавляет новый пункт в выпадающее меню «Files of type» диалогового окна выбора файлов. Например, следующий код обеспечивает четыре пункта меню «Files of type»: `Images`, `Documents`, `Archives` и `All`.

```
var fileFilter1:FileFilter = new FileFilter("Images", "*.png;*.gif;*.jpg");
var fileFilter2:FileFilter = new FileFilter("Documents",
                                            "*.txt;*.doc;*.pdf;*.rtf");
var fileFilter3:FileFilter = new FileFilter("Archives", "*.zip;*.tar;*.hqx");
var fileFilter4:FileFilter = new FileFilter("All", "*.*");
_fileReference.browse([fileFilter1, fileFilter2, fileFilter3, fileFilter4]);
```

При передаче массива объектов *FileFilter* в метод *browse( )* существует вероятность того, что *browse( )* сформирует ошибку недопустимости аргумента типа *ArgumentError*. Такие ошибки возникают, если любой из объектов *FileFilter* содержит неправильно заданные значения.

Параметр фильтра файлов можно применять и в методе *browse( )* объекта *FileReferenceList*.

См. также

Рецепт 23.4.

23.6. Определение момента выбора пользователем файла для передачи

Задача

Требуется отследить момент, когда пользователь выбрал файл в диалоговом окне просмотра файлов.

Решение

Слушайте событие `select`. Слушайте событие `cancel`, чтобы отслеживать нажатие пользователем кнопки Cancel.

Обсуждение

Когда пользователь выбирает файл и нажимает кнопку Open диалогового окна просмотра файлов, объект *FileReference* формирует событие `select` типа *Event*. Слушателя этого события можно добавить, задействовав константу `Event.SELECT`:

```
fileReference.addEventListener(Event.SELECT, onSelectFile);
```

Когда пользователь выбирает файл, сразу становится доступной информация о нем (`name`, `size`, `createdDate` и т. д.). Например, как только объект *FileReference* рассылает событие *SELECT*, можно получить имя выбранного файла, как в следующем примере:

```
selectedFileTextField.text = fileReference.name;
```

Когда пользователь нажимает кнопку Cancel диалогового окна просмотра файлов, объект *FileReference* рассылает событие `cancel` типа *Event*. Слушатель этого события добавляется посредством задания константы `Event.CANCEL`:

```
fileReference.addEventListener(Event.CANCEL, onCancelBrowse);
```

Объекты *FileReferenceList* также используют события `select` и `cancel`.

См. также

Рецепт 23.7.

23.7. Передача файлов

Задача

Требуется разрешить пользователям передачу файлов.

Решение

Обратитесь к методу *upload( )* объекта *FileReference*.

Обсуждение

Метод *upload( )* объекта *FileReference* позволяет загружать файл на сервер с помощью серверного сценария, сконфигурированного на прием файлов через HTTP(S). Метод *upload( )* принимает как минимум один обязательный параметр – объект *URLRequest*, определяющий URL сценария, которому требуется передать данные файла:

```
var urlRequest:URLRequest = new URLRequest("uploadScript.cgi");
fileReference.upload(urlRequest);
```

При передаче файлов методом *upload( )* всегда задействуется метод `POST`, параметр `Content-Type` которого имеет значение `multipart/form-data`. По умолчанию `Content-Disposition` имеет значение `Filedata`. Сценарий должен знать значение `Content-Disposition`, чтобы иметь возможность читать данные файла. Если используется иное, не применяемое по умолчанию значение `Content-Disposition`, оно должно быть задано как второй, необязательный параметр метода *upload( )*:

```
fileReference.upload(urlRequest, "UploadFile");
```

Файл можно передавать, только если выбор его пользователем реализован с участием метода *browse( )*. Когда пользователь выбирает файл методом *browse( )* объекта *FileReference*, метод *upload( )* можно вызвать после возникновения события `select`. Если выбор файлов осуществляется посредством объекта *FileReferenceList*, то метод *upload( )* следует вызвать для каждого файла, содержащегося в свойстве `file-`

`List` объекта. Свойство `fileList` объекта *FileReferenceList* – это массив объектов *FileReference*, каждый из которых соответствует отдельному выбранному пользователем файлу.

Метод *upload( )* может формировать ошибки. Эти ошибки идентичны формируемым методом *download( )*. Кроме того, как и *download( )*, метод *upload( )* может рассылать события нарушения безопасности и события ошибок ввода/вывода. Более подробно обработка этих ошибок и событий рассмотрена в рецепте 23.1.

См. также

Рецепт 23.1.

23.8. Отслеживание процесса передачи файла

Задача

Требуется отслеживать процесс передачи файла на сервер.

Решение

Слушайте событие `progress`.

Обсуждение

Так же как мы отслеживали процесс загрузки файла, можно отслеживать и процесс передачи файла. Объект *FileReference* при передаче файла рассылает события `progress`. По завершении передачи он формирует событие `complete`. Подробнее все это описано в рецепте 23.3.

См. также

Рецепт 23.3.

24

Работа с сокетами

24.0. Введение

Использование сокетов позволяет Flash-проигрывателю обмениваться данными с сервером через заданный сетевой порт. Основное отличие сокетного соединения от серверного, описанного в главе 20, заключается в том, что серверное соединение автоматически закрывается после завершения передачи данных.

После установления сокетное соединение остается открытым, пока клиент (Flash Player) или сервер не закроет его явно. Благодаря этому существует особый тип связи, при которой сервер может передавать данные в любой момент времени, не дожидаясь запроса на них от Flash Player (это называют push-технологией).

Сокетные соединения обычно служат для создания многопользовательских приложений. Сетевой чат – это один из примеров такого приложения. Программа чата обычно представляет собой центральный сервер, к которому подключены различные Flash Player-клиенты. Получив сообщение от *.swf*-клиента, сервер определяет, кому это сообщение адресовано и передает его этому клиенту через открытое соединение. В данном случае клиент-получатель не запрашивает сообщение, сервер просто передает его этому клиенту. Когда клиент закрывает соединение, сервер сообщает об этом другим клиентам.

Flash Player может устанавливать с сервером сокетные соединения двух типов. Их поведение и работа очень схожи, но имеются и отличия. Первое – сокетные соединение для обмена XML-данными, которое подобно *XMLSocket* из предыдущих версий Flash Player. Новым для Flash Player 9 является сокетное соединение для обмена двоичными данными.

Для создания сокетного соединения обмена XML-данными задействуется класс *flash.net.XMLSocket*. Для создания сокетного соединения обмена двоичными данными – класс *flash.net.Socket*.

Сокетные соединения для обмена XML-данными специализируются на передаче текста. Обмен информацией между клиентом и сервером осуществляется за счет обмена пакетами XML, содержащими данные. Действия предпринимаются исходя из результатов анализа содержимого пакетов XML.

Сокетные соединения для передачи двоичных данных появились в ActionScript 3.0 и обеспечивают возможность передачи необработанных двоичных данных. Такие соединения немного сложнее, чем соединения для передачи XML, потому что требуют от разработчика знания двоичных типов данных, но также и более мощные, потому что позволяют устанавливать соединения с более широким диапазоном серверов и вообще предлагают большие возможности. Например, соединения для передачи двоичных данных обеспечивают возможность соединяться с почтовыми серверами (через POP3, SMTP и IMAP), серверами новостей (через NNTP), чат-серверами или даже реализовывать совместный просмотр рабочего стола и работу с удаленными настольными приложениями посредством соединения с VNC-сервером (через RFB).

Независимо от типа соединения, будь это соединение для передачи двоичных данных или XML, оно является асинхронным. Это означает, что пока от обработчика события не получено уведомление о поступлении данных, они недоступны для чтения. Любое чтение данных должно осуществляться в обработчике события, который выполняется автоматически, когда данные становятся доступными. Синхронные сетевые соединения проще разрабатывать, но они не так эффективны, поскольку программе приходится приостанавливать выполнение и ожидать поступления данных. Рецепты данной главы подробно рассказывают о работе с асинхронными сокетными соединениями Flash Player.

24.1. Соединение с сокет-сервером

Задача

Требуется установить соединение с сокет-сервером.

Решение

Задействуйте метод *Socket.connect( )* или *XMLSocket.connect( )* и слушайте событие `connect`, сигнализирующее об установлении соединения.

Обсуждение

Прежде чем приступить к установлению сокетного соединения с сервером, необходимо знать, во-первых, доменное имя или IP-адрес сервера и, во-вторых, порт, через который будет осуществляться соединение.

Независимо от того, используется ли экземпляр *Socket* или *XMLSocket*, процесс установления соединения одинаковый. Оба класса определяют метод *connect( )*, принимающий два параметра:

`host`

Строковое значение, определяющее узел сети, с которым устанавливается соединение. Это может быть доменное имя, например `www.example.com`, или IP-адрес, например `192.168.1.101`. Чтобы установить соединение с сервером, на котором располагается ролик Flash, вместо строки с именем узла сети надо передать значение `null`.

`port`

Значение типа *int*, определяющее номер порта, через который будет устанавливаться соединения с узлом сети. Это значение должно быть не меньше 1024, если только файл политики безопасности сервера не разрешает порты ниже 1024.

Из-за асинхронной природы сокетных соединений во Flash метод *connect( )* не ожидает установления соединения, а продолжает выполнение кода ActionScript со следующей строки. Попытка начать взаимодействие с соединением до его полного установления приводит к непредвиденным результатам и некорректной работе кода.

Для правильного установления сокетного соединения, прежде чем вызывать метод *connect( )*, необходимо добавить слушателя события `connect`. Событие `connect` формируется объектами *Socket* и *XMLSocket* при успешном установлении соединения и информирует о том, что соединение готово к работе.

Код для установления соединения экземпляра *Socket* с сервером, выполняющимся на `localhost`, через порт 2900 выглядит так:

```
package {
  import flash.display.Sprite;
  import flash.events.*;
  import flash.net.Socket;

  public class SocketExample extends Sprite {

    private var socket:Socket;

    public function SocketExample() {
      socket = new Socket();

      // Добавляем слушателя события, чтобы получить
      // уведомление, когда соединение будет установлено.
      socket.addEventListener( Event.CONNECT, onConnect );

      // Соединяемся с сервером
      socket.connect( "localhost", 2900 );
    }

    private function onConnect( event:Event ):void {
      trace( "Соединение установлено..." );
    }

  }
}
```

Для объекта *XMLSocket* код будет практически аналогичным: сначала создается слушатель события `connect`, затем вызывается метод *connect( )*. Единственное отличие в том, что место *Socket* везде занимает *XMLSocket*:

```
package {
  import flash.display.Sprite;
  import flash.events.*;
  import flash.net.XMLSocket;

  public class SocketExample extends Sprite {

    private var socket:XMLSocket;

    public function SocketExample() {
      socket = new XMLSocket();

      // Добавляем слушателя события, чтобы получить
      // уведомление, когда соединение будет установлено.
      socket.addEventListener( Event.CONNECT, onConnect );

      // Соединяемся с сервером
      socket.connect( "localhost", 2900 );
    }

    private function onConnect( event:Event ):void {
      trace( "Соединение для передачи xml установлено..." );
    }

  }
}
```

Сбой при установлении соединения может возникнуть по двум причинам: или сразу не удается установить соединение и формируется ошибка времени выполнения, или возникает событие `ioError` или `securityError`, свидетельствующее о невозможности успешно завершить установление соединения. Более подробно об обработке событий ошибок рассказывает рецепт 24.6.

Помните, что при установлении сетевого соединения с узлом сети действуют следующие правила безопасной изолированной программной среды Flash Player:

1. Для успешного установления соединения *.swf* и узел сети должны располагаться в одном и том же домене.
2. Удаленный файл *.swf* не может устанавливать соединение с локальным сервером.
3. Локальный ненадежный файл *.swf* не может получить доступа ни к одному сетевому ресурсу.
4. Чтобы разрешить междоменный доступ или соединение с портом ниже 1024, используется файл политики.

Нарушение рамок безопасной изолированной программной среды при попытках соединения с ненадежным доменом или портом ниже 1024

приводит к формированию события `securityError`. Справиться с этими ограничениями поможет файл междоменной политики безопасности, обсуждаемый в рецепте 3.12. Чтобы использовать такой файл политики с объектами *Socket* или *XMLSocket*, необходимо загрузить его методом *flash.system.Security.loadPolicyFile( )*, как показано в данном фрагменте кода:

```
Security.loadPolicyFile("http://www.rightactionscript.com/crossdomain.xml");
```

При создании файла междоменной политики необходимо определить не только разрешенные домены, но и разрешенные порты. Они должны быть перечислены через запятую в атрибуте `port` тега `<allow-access-from>`. Если порты не указаны, Flash Player предполагает, что порт 80 (стандартный HTTP-порт) является единственным разрешенным портом. Следующий файл политики разрешает всем доменам соединение с портами 80 и 110 (стандартные порты HTTP и почтовый POP):

```
<?xml version="1.0"?>

<!DOCTYPE cross-domain-policy SYSTEM "http://www.macromedia.com/xml/dtds
                                                    /cross-domain-policy.dtd">
<cross-domain-policy>
   <allow-access-from domain="*" to-ports="80,110" />
</cross-domain-policy>
```

См. также

Рецепты 3.12, 24.5 и 24.6.

24.2. Отправка данных

Задача

Требуется отправить данные на сервер.

Решение

Для объектов *Socket* с помощью методов для записи (*writeByte( )*, *writeUTFBytes( )* и др.) запишите данные в буфер и вызовите метод *flush( )*, чтобы отправить эти данные. Для объектов *XMLSocket* следует применять метод *send( )*.

Обсуждение

Классы *Socket* и *XMLSocket* определяют разные программные интерфейсы для отправки данных на сервер. Начнем с API класса *Socket*.

При отправке данных на сервер с использованием объекта *Socket* необходимо сначала записать эти данные в буфер. Класс *Socket* определяет массу методов для записи данных. Каждый метод предназначен для записи разных типов данных (или записывает данные по-разному). К этим

методам относятся *writeBoolean( )*, *writeByte( )*, *writeBytes( )*, *writeDouble( )*, *writeFloat( )*, *writeInt( )*, *writeMultiByte( )*, *writeObject( )*, *writeShort( )*, *writeUnsignedInt( )*, *writeUTF( )* и *writeUTFBytes( )*. Большинство методов принимают один параметр, тип которого обозначен в имени метода. Например, метод *writeBoolean( )* принимает Булев параметр, а методы *writeByte( )*, *writeDouble( )*, *writeFloat( )*, *writeInt( )*, *writeShort( )* и *writeUnsignedInt( )* принимают числовые значения. Метод *writeObject( )* принимает в качестве параметра объект, который должен быть сериализован в формат AMF. Метод *writeBytes( )* принимает параметр типа *ByteArray* и допускает параметры *offset* и *length*. Например, в следующей строке кода вызывается *writeBytes( )*, в который передается объект *ByteArray* и определяется, что он должен записать все байты (начиная со смещения 0, и в количестве, равном длине *ByteArray*):

```
socket.writeBytes(byteArray, 0, byteArray.length);
```

Методы *writeUTF( )* и *writeUTFBytes( )* позволяют записывать строки. Каждый из них принимает строковый параметр. Метод *writeUTFBytes( )* просто записывает строку в виде байтов. Метод *writeUTF( )* указывает сохраняемое количество байт, прежде чем записать побайтное представление данных.

Метод *writeMultiByte( )* также записывает строковые данные, но использует нестандартный набор символов. Метод принимает два обязательных параметра: строку, которую необходимо записать, и имя набора символов. В документации по Flash и Flex перечислены поддерживаемые наборы символов, их названия и псевдонимы. При определении набора символов в *writeMultiByte( )* указывается строка его названия. В следующем примере записывается строка `example` в Unicode:

```
socket.writeMultiByte("example", "unicode");
```

Какой метод выбрать для записи данных в объекте *Socket*, всецело зависит от того, какие данные требуется записать и какие данные ожидает сервер. С помощью объекта *Socket* можно написать почтовый клиент Telnet или POP, используя только ActionScript. Оба протокола ожидают текстовые команды в кодировке ASCII. Например, после соединения с POP-сервером с помощью команды `USER` можно передать имя пользователя. Следующая строка передает такую команду в объект *Socket*:

```
// POP-серверы ожидают символ перевода строки (\n), чтобы выполнить
предыдущую команду.
socket.writeUTFBytes("USER exampleUsername\n");
```

Записывая данные в объект *Socket*, мы не передаем их на сервер. При каждом вызове метода *write* данные добавляются в объект *Socket*. Например, следующий фрагмент кода записывает в объект *Socket* четыре байта, но ни один из них не отправляется на сервер:

```
socket.writeByte(1);
socket.writeByte(5);
```

```
socket.writeByte(4);
socket.writeByte(8);
```

Когда приходит время передать накопленные данные на сервер, вызывается метод *flush( )*. Он просто отправляет все записанные данные и очищает буфер:

```
socket.flush();
```

Программный интерфейс для отправки данных класса *XMLSocket* намного проще. Запись и передача данных осуществляются одним методом с подходящим именем *send( ) (отправить)*. Метод *send( )* принимает параметр любого типа. Он преобразует этот параметр в строку и отправляет ее на сервер. Как правило, в качестве параметра фигурирует XML-объект или строка, содержащая данные, структурированные как XML:

```
xmlSocket.send(xml);
```

Однако конкретный формат данных всецело зависит от ожидаемого сервером формата. Если сервер ожидает XML-данные, необходимо отправлять XML-данные. Если сервер ожидает URL-кодированные данные, должны передаваться URL-кодированные данные.

См. также

Рецепты 24.1 и 24.3.

24.3. Получение данных

Задача

Требуется читать данные с сервера.

Решение

Работая с экземплярами *Socket*, надо подписаться на событие `socketData` и вызывать в обработчике события один из методов чтения, например *readByte( )* или *readInt( )*. Это гарантирует, что будет прочитано количество байтов, не превышающее доступного их числа, которое определяется свойством `bytesAvailable`.

Для экземпляров *XMLSocket* подпишитесь на событие `data` и обрабатывайте полученные XML-данные в обработчике этого события.

Обсуждение

Процесс получения данных через сокетное соединение зависит от его типа. И *Socket*, и *XMLSocket* могут получать данные с сервера, но делают они это с помощью немного разных техник. Начнем с класса *Socket*.

Как говорилось во введении к данной главе, сокетные соединения во Flash являются асинхронными. Поэтому нельзя просто создать сокет-

ное соединение и сразу же читать данные из него. Методы *read*, не дожидаются, пока данные будут переданы с сервера. Данные становятся доступными для чтения только после того, как будут загружены клиентом с сервера. При попытке чтения данных из *Socket* до того, как они станут доступными, возникает ошибка.

О том, что данные доступны для чтения, извещает событие `socketData`, рассылаемое экземплярами *Socket*. Добавляя слушателя этого события, мы обеспечиваем вызов обработчика события при каждом получении новых данных с сервера. Код для чтения и интерпретации полученных данных размещается в обработчике события.

Для чтения поступающих с сервера данных класс *Socket* предоставляет множество разных *read*-методов для разных типов данных. Например, посредством метода *readByte( )* можно читать байты, а метод *readUnsignedInt( )* обеспечивает чтение целых чисел без знака. В табл. 24.1 представлен список различных типов данных, которые могут быть прочитаны с сервера, тип возвращаемого значения и количество байтов, считываемых методом *read*.

Таблица 24.1. Методы чтения для различных типов данных

Метод : Возвращаемый тип	Описание	Количество считываемых байт
`readBoolean():Boolean`	Читает Булево значение	1
`readByte():int`	Читает знаковый байт	1
`readDouble():Number`	Читает число с плавающей точкой двойной точности в формате IEEE 754	8
`readFloat():Number`	Читает число с плавающей точкой одинарной точности в формате IEEE 754	4
`readInt():int`	Читает 32-разрядное целое со знаком	4
`readObject():*`	Читает объект, кодированный в формате AMF	*n*
`readShort():int`	Читает 16-разрядное целое со знаком	2
`readUnsignedByte():uint`	Читает беззнаковый байт	1
`readUnsignedInt():uint`	Читает 32-разрядное целое без знака	4
`readUnsignedShort():uint`	Читает 16-разрядное целое без знака	2
`readUTF():String`	Читает строку в кодировке UTF-8	*n*

Есть еще два метода чтения, не внесенные в табл. 24.1. Это *readBytes( )* и *readUTFBytes( )*. Метод *readBytes( )* – единственный метод чтения *Socket*, который не возвращает значение. Он принимает следующие три параметра:

bytes

Экземпляр *flash.utils.ByteArray*, в который происходит чтение данных из экземпляра *Socket*.

offset

Значение типа *uint*, определяющее смещение в *bytes*, с которым должны размещаться данные, считанные из *Socket*. По умолчанию равно 0.

length

Значение типа *uint*, определяющее, сколько байтов должно быть считано. По умолчанию равно 0, т. е. в *ByteArray* будут считаны все доступные в *Socket* данные.

Другой метод, *readUTFBytes( )*, принимает единственный параметр *length*, определяющий, сколько байтов в кодировке UTF-8 должно быть считано, и возвращает соответствующую строку (*String*).

Перед чтением данных из объекта *Socket* важно проверить сначала его свойство `bytesAvailable`. Попытка прочитать больше данных, чем доступно, приводит к ошибке *flash.errors.EOFError*.

В следующем примере кода осуществляется соединение с сервером, побайтное чтение и отображение данных, присылаемых с сервера:

```
package {
  import flash.display.Sprite;
  import flash.events.ProgressEvent;
  import flash.net.Socket;

  public class SocketExample extends Sprite {

    private var socket:Socket;

    public function SocketExample() {
      socket = new Socket();

      // Ожидаем события получения данных с сервера
      socket.addEventListener( ProgressEvent.SOCKET_DATA, onSocketData );

      // Соединяемся с сервером
      socket.connect( "localhost", 2900 );
    }

    private function onSocketData( event:ProgressEvent ):void {
      trace( "Экземпляр Socket получил " + socket.bytesAvailable +
                                              + " байт(ов) данных:" );

      // Читаем полученные данные по одному байту, пока таковые доступны
```

```
        while ( socket.bytesAvailable ) {
          // Читаем байт из socket и выводим на экран
          var data:int = socket.readByte();
          trace( data );
        }
      }
    }
  }
```

Если сервер возвратит сообщение (например, «Hello»), то вывод кодов символов при подключении клиента выглядел бы так:

```
Экземпляр Socket получил 5 байт(ов) данных:
72
101
108
108
111
```

Данные, прочитанные из *Socket*, не могут быть прочитаны вновь. Например, если байт прочитан, он не может быть «положен обратно» и прочитан вновь позже как часть значения типа *int*.

Если получаемые объектом *Socket* данные представляют собой текст в формате ASCII, то с помощью метода *readUTFBytes( )* можно восстановить строку. Методу *readUTFBytes( )* необходимо указать количество байтов, которое необходимо прочитать и преобразовать в строку. Чтобы прочитать все байты, надо задать значение `bytesAvailable`:

```
var string:String = socket.readUTFBytes(socket.bytesAvailable);
```

Класс *XMLSocket* ведет себя аналогично классу *Socket* в том, что касается получения данных с сервера. В обоих случаях из-за того, что во Flash реализованы асинхронные сокетные соединения, используется слушатель события, обеспечивающий уведомление о получении данных. Однако процесс фактического чтения данных в этих классах сильно отличается.

Когда загрузка данных с сервера завершена, экземпляр *XMLSocket* рассылает событие `data`. Событие `data`, определяемое константой *flash.events.DataEvent.DATA*, содержит свойство `data` типа *String*, в котором располагается информация, полученная с сервера.

При работе с *XMLSocket* возвращенные с сервера данные всегда трактуются как *String*. Здесь нет специальных методов чтения для разных типов данных.

Значение свойства `data` представляет собой необработанный ответ сервера. Поэтому при работе с *XMLSocket* могут передаваться не только экземпляры *XML*, но и простая строковая информация. Однако если ожидается, что сервер должен возвратить *XML*, перед работой с полученными данными они должны быть преобразованы в экземпляр *XML*.

В следующем примере кода посредством *XMLSocket* устанавливается соединение с сервером, выполняющимся на порту 290 на локальном компьютере. После успешного установления соединения на сервер отправляется сообщение *<test>*. Слушатель события *onData* обрабатывает ответ сервера, в данном случае это строка `<response><test success='true'/></ response>`. Как видите, свойство `data` события, передаваемого в *onData*, является просто строкой (*String*), и для преобразования этой *String* в экземпляр *XML* используется конструктор *XML*. Наконец, для вывода части преобразованного *XML* используется синтаксис *E4X* (более подробно о работе с *XML* и применении *E4X* рассказано в главе 21):

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.DataEvent;
  import flash.net.XMLSocket;

  public class SocketExample extends Sprite {

    private var xmlSocket:XMLSocket;

    public function SocketExample() {
      xmlSocket = new XMLSocket();

      // Слушатель события Connect для отправки сообщения
      // на сервер после успешного установления соединения
      xmlSocket.addEventListener( Event.CONNECT, onConnect );

      // Слушаем событие, свидетельствующее
      // о получении данных с сервера
      xmlSocket.addEventListener( DataEvent.DATA, onData );

      // Соединяемся с сервером
      xmlSocket.connect( "localhost", 2900 );
    }

    private function onConnect( event:Event ):void {
      xmlSocket.send( "<test/>" );
    }

    private function onData( event:DataEvent ):void {
      // Необработанная строка, возвращенная с сервера.
      // Она может выглядеть следующим образом:
      // <response><test success='true'/></response>
      trace( event.data );

      // Преобразуем строку в XML
      var response:XML = new XML( event.data );

      // Используя E4X, организовываем доступ к атрибуту
      // success узла элементов "test" ответа. Вывод: true
      trace( response.test.@success );
    }
  }
}
```

Событие `data` может быть сформировано только после того, как экземпляр *XMLSocket* получит нулевой байт (`'\0'`) от сервера. То есть чтобы клиент получил строку от сервера, недостаточно просто отправить ее. Она еще должна завершаться нулевым байтом.

См. также

Рецепты 18.2, 24.1 и 24.4.

24.4. Подтверждение установления связи с сервером

Задача

Требуется подтвердить установление связи с сервером и выяснить контекст полученных данных, чтобы знать, как их обрабатывать.

Решение

Создаем константы для представления состояний протокола. Используем константы для сопоставления конкретных функций обработки с определенным состоянием. В обработчике события `socketData` посредством сопоставления состояния вызываем соответствующую функцию.

Обсуждение

Процесс подтверждения установления соединения представляет собою обычный сценарий при работе с сокетными соединениями. Как правило, сначала сервер отправляет данные клиенту, клиент отвечает на эти данные определенным образом, после чего отвечает сервер, тоже соответствующим образом. Весь этот процесс повторяется до тех пор, пока не будет установлено «нормальное» соединение и не будет подтверждено установление связи.

Сложность обработки ответа сервера заключается в том, что обработчик события `socketData` не отслеживает контекст. То есть ответ сервера не сопровождается информацией о причинах поступления таких данных или директивой «эти данные предоставляются в ответ на...». Не всегда из ответа можно понять, как его следует обрабатывать, особенно если поступают разные ответы. Один ответ может возвращать два байта, другой – целое, за которым следует значение типа *double*. Как видите, все не так-то просто.

Решение заключается в том, чтобы создать константы различных состояний, представляющие различные контексты, в которых сервер отправляет данные клиенту. Ассоциируя каждую из таких констант с определенной функцией обработки данных, можно без труда обеспечить вызов соответствующей функции исходя из текущего состояния протокола.

Рассмотрим следующий сценарий подтверждения установления связи, реализуемый при соединении с сервером:

1. При установлении соединения сервер немедленно отправляет целое, представляющее самую высокую из поддерживаемых версию протокола.
2. Клиент отвечает целым, обозначающим версию протокола, которая фактически будет использоваться при обмене информацией.
3. Сервер возвращает 8-байтный запрос аутентификации.
4. Клиент возвращает запрос аутентификации серверу.
5. Сервер закрывает соединение, если ответ клиента не соответствует ожидаемому или протокол переходит в обычный рабочий режим и процесс подтверждения установления связи завершается.

Но на шаге 4 на запрос аутентификации дается более защищенный ответ. Запрос не просто возвращается как есть, он шифруется определенным образом с применением предоставленного пользователем ключа. Например, клиент может запросить у пользователя пароль, который затем выступает в качестве ключа для шифрования 8-байтного запроса. После этого шифрованный запрос возвращается на сервер. Если этот ответ соответствует ожиданиям сервера, значит, клиент знает необходимый пароль, и соединение может быть разрешено.

Для реализации описанного процесса подтверждения установления связи сначала необходимо создать константы, представляющие разные типы данных, возвращаемых сервером. Во-первых, при определении версии (шаг 1). Во-вторых, при получении запроса (шаг 3). Наконец, обычный рабочий режим (шаг 5). Все это может быть представлено следующими константами:

```
public const DETERMINE_VERSION:int = 0;
public const RECEIVE_CHALLENGE:int = 1;
public const NORMAL:int = 2;
```

Неважно, какие значения присваиваются константам. Главное, чтобы все они были разными.

Следующий этап – создание функций для обработки данных. Создадим три функции: *readVersion( )* (читать версию), *readChallenge( )* (читать запрос) и *readNormalProtocol( )* (читать обычный протокол). После описания функций необходимо сопоставить созданные ранее константы состояния с функциями, которые будут использоваться для обработки данных, получаемых в этом состоянии. Код выглядит так:

```
stateMap = new Object();
stateMap[ DETERMINE_VERSION ] = readVersion;
stateMap[ RECEIVE_CHALLENGE ] = readChallenge;
stateMap[ NORMAL            ] = readNormalProtocol;
```

Последний шаг – создать такой обработчик события `socketData`, который обеспечивал бы вызов соответствующей функции обработки исхо-

дя из текущего состояния протокола. Для этого создается переменная currentState (текущее состояние) типа *int*. Затем с помощью stateMap (карта состояний) путем поиска функции, ассоциированной с currentState, вызывается необходимая функция обработки:

```
var processFunc:Function = stateMap[ currentState ];
processFunc(); // Вызываем соответствующую функцию обработки
```

Здесь надо быть внимательным и не забывать обновлять переменную currentState, чтобы она правильно отражала текущее состояние протокола.

Весь пример кода описанного выше сценария подтверждения установления связи выглядит так:

```
package {
  import flash.display.Sprite;
  import flash.events.ProgressEvent;
  import flash.net.Socket;
  import flash.utils.ByteArray;

  public class SocketExample extends Sprite {

    // Константы состояний для описания протокола
    public const DETERMINE_VERSION:int = 0;
    public const RECEIVE_CHALLENGE:int = 1;
    public const NORMAL:int = 2;

    // Сопоставляем состояние и функцию обработки
    private var stateMap:Object;

    // Отслеживаем текущее состояние протокола
    private var currentState:int;

    private var socket:Socket;

    public function SocketExample() {
      // Сопоставляем состояния и константы состояний
      stateMap = new Object();
      stateMap[ DETERMINE_VERSION ] = readVersion;
      stateMap[ RECEIVE_CHALLENGE ] = readChallenge;
      stateMap[ NORMAL            ] = readNormalProtocol;

      // Инициализируем текущее состояние
      currentState = DETERMINE_VERSION;

      // Создаем socket и устанавливаем соединение
      socket = new Socket();
      socket.addEventListener( ProgressEvent.SOCKET_DATA, onSocketData );
      socket.connect( "localhost", 2900 );
    }

    private function onSocketData( event:ProgressEvent ):void {
      // Ищем функцию обработки исходя из текущего состояния
      var processFunc:Function = stateMap[ currentState ];
      processFunc();
    }
```

```
        private function readVersion():void {
          // Шаг 1 – читаем версию с сервера
          var version:int = socket.readInt();

          // Когда версия получена, следующее состояние –
          // получение запроса аутентификации от сервера
          currentState = RECEIVE_CHALLENGE;

          // Шаг 2 – возвращаем версию на сервер
          socket.writeInt( version );
          socket.flush();
        }

        private function readChallenge():void {
          // Шаг 3 - читаем 8-байтный запрос в байтовый массив
          var bytes:ByteArray = new ByteArray();
          socket.readBytes( bytes, 0, 8 );

          // Следующее состояние после получения запроса –
          // обычная работа протокола
          currentState = NORMAL;

          // Шаг 4 – возвращаем ответ на сервер
          socket.writeBytes( bytes );
          socket.flush();
        }

        private function readNormalProtocol():void {
          // Шаг 5 – после завершения процесса подтверждения
          // установления связи обрабатываем обычные сообщения
        }
      }
    }
```

См. также

Рецепты 18.2, 24.1 и 24.3.

24.5. Разрыв соединения с сервером

Задача

Требуется разорвать соединение с сервером или получать уведомление, когда сервер разрывает соединение с вами.

Решение

Вызывайте методы *Socket.close( )* или *XMLSocket.close( )*, чтобы явно закрыть соединение, или слушайте событие `close`, уведомляющее о том, что сервер закрыл соединение для вас.

Обсуждение

Общее правило программирования – убирать после себя. Если ранее созданный объект больше не нужен, он должен быть уничтожен. В на-

шем случае соединение с сервером должно быть явно закрыто после окончания работы с ним. Открытые неиспользуемые сокетные соединения – пустая трата ресурсов. Если соединение не закрыто, сервер может продолжать поддерживать его открытым, в результате чего быстро будет достигнуто максимально допустимое для сервера количество сетевых соединений.

Закрытие сетевых соединений для экземпляров *Socket* и *XMLSocket* происходит одинаково. Достаточно вызвать метод *close( )* экземпляра:

```
// Предполагаем, что socket является соединенным экземпляром Socket
socket.close();     // Разрываем соединение с сервером
```

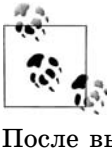
С экземпляром *XMLSocket* все то же самое:

```
// Предполагаем, что xmlSocket является соединенным экземпляром XMLSocket
xmlSocket.close();  // Разрываем соединение с сервером
```

Метод *close( )* сообщает серверу, что клиент хочет разорвать соединение. Чтобы получать уведомление о том, что сервер сам закрывает соединение, в экземпляре *Socket* или *XMLSocket* необходимо слушать событие `close`. Для этого методом *addEventListener( )* добавляем слушателя события типа `Event.CLOSE`, например:

```
var socket:Socket = new Socket();

// Добавляем слушателя события, чтобы получать уведомление,
// когда сервер разрывает соединение с клиентом
socket.addEventListener( Event.CLOSE, onClose );
```



Вызов метода *close( )* не формирует события `close`. Событие `close` возникает, только когда разрыв соединения инициируется сервером.

После вызова метода *close( )* экземпляр соединения не может читать или писать данные. Если требуется повторно использовать этот экземпляр, необходимо снова установить соединение, как описывается в рецепте 24.1.

См. также

Рецепт 24.1.

24.6. Обработка ошибок сокетного соединения

Задача

Требуется обрабатывать ошибки, которые могут возникать при использовании сокетных соединений.

Решение

Организуйте блоки `try/catch` для обработки ошибок ввода/вывода и выхода за пределы файла (EOF).

Обсуждение

Что касается ошибок и событий ошибок, классы *Socket* и *XMLSocket* ведут себя одинаково. При вызове метода *connect( )* объекты *Socket* и *XMLSocket* могут формировать ошибку типа *SecurityError* при выполнении одного из следующих условий:

- *.swf* классифицируется как ненадежный в данном домене.
- Номер порта выше, чем 65535.

Если соединение не установлено, при вызове метода *send( )* (*XMLSocket*) или *flush( )* (*Socket*) может формироваться ошибка типа *IOError*. Хотя можно (и следует) осуществлять вызовы методов *send( )* или *flush( )* в рамках блоков `try/catch`, не стоит использовать `try/catch` для реализации логики приложения. Если в приложении перед вызовом *send( )* или *flush( )* требуется реализовать проверку свойства *connected* объекта соединения, лучше прибегнуть к выражению `if`. Например, в следующем фрагменте кода выражение `if` играет роль фрагмента логики приложения для вызова метода *connectToSocketServer( )* в случае, если объект *Socket* на данный момент не соединен с сервером. Здесь также организован блок `try/catch` для регистрации ошибок, формируемых методом *flush( )*:

```
if ( socket.connected ) {
    try {
        socket.flush();
    }
    catch( error:IOError ) {
        logInstance.write( "socket.flush error\n" + error );
    }
}
else {
    connectToSocketServer();
}
```

Все `read`-методы *Socket* могут формировать ошибки типа *EOFError* и *IOError*. Ошибки EOF (конец файла) возникают, если осуществляется попытка чтения данных в отсутствие доступных для чтения данных. Ошибки I/O (ввода/вывода) возникают при попытке чтения из уже закрытого сетевого соединения.

Кроме ошибок, формируемых методами классов *Socket* и *XMLSocket*, объекты этих классов также рассылают события ошибок. Два основных типа таких событий, возникающие при работе с сетевыми соединениями: *IOError* и *securityError*. Событие *IOError* типа *IOErrorEvent* возникает в случае сбоя при отправке или загрузке данных. Событие *securityError* типа *SecurityErrorEvent* – если не удается установить сетевое соединение с сервером из-за того, что сервер находится вне безопасной изолированной программной среды или номер порта ниже 1024.

В обоих случаях этих ошибок можно избежать, используя файл междоменной политики.

A

Таблица кодов Unicode для символов Latin 1

В табл. A.1 перечислены символы Latin 1 с эквивалентными им кодами Unicode в диапазоне от U+0000 до U+00FF (т. е. управляющие символы C0, базовые символы Latin, управляющие символы C1 и дополнительные символы Latin 1). Таблица воспроизводит приложение B книги «ActionScript for Flash MX: The Definitive Guide» с разрешения автора, Колина Мука (Colin Moock).[1]

В первом столбце таблицы (под названием **Dec**) перечислены коды символов в десятичном виде (стандартное значение ASCII или Latin 1), второй столбец представляет код Unicode символа и в третьем столбце описывается или показывается сам символ.

Таблица A.1. Символы и соответствия Unicode согласно стандарту ISO 8859-1 (Latin 1)

Dec	Unicode	Описание	Dec	Unicode	Описание
0	\u0000	[пустое значение]	6	\u0006	[подтверждение]
1	\u0001	[начало заголовка]	7	\u0007	[звуковой сигнал]
2	\u0002	[начало текста]	8	\u0008	[возврат]
3	\u0003	[конец текста]	9	\u0009	[горизонтальная табуляция]
4	\u0004	[конец передачи]	10	\u000a	[перевод строки]
5	\u0005	[запрос]	11	\u000b	[вертикальная табуляция]

[1] Колин Мук «ActionScript для Flash MX. Подробное руководство». – Пер. с англ. – СПб: Символ-Плюс, 2004.

Dec	Unicode	Описание	Dec	Unicode	Описание
12	\u000c	[перевод страницы]	42	\u002a	*
13	\u000d	[возврат каретки]	43	\u002b	+
14	\u000e	[shift выкл.]	44	\u002c	, (запятая)
15	\u000f	[shift вкл.]	45	\u002d	– (минус)
16	\u0010	[смена канала данных]	46	\u002e	. (точка)
17	\u0011	[элемент управления устройством 1]	47	\u002f	/
18	\u0012	[элемент управления устройством 2]	48	\u0030	0
19	\u0013	[элемент управления устройством 3]	49	\u0031	1
20	\u0014	[элемент управления устройством 4]	50	\u0032	2
21	\u0015	[отрицание]	51	\u0033	3
22	\u0016	[синхронизация]	52	\u0034	4
23	\u0017	[конец блока передачи]	53	\u0035	5
24	\u0018	[отмена]	54	\u0036	6
25	\u0019	[конец носителя]	55	\u0037	7
26	\u001a	[замена]	56	\u0038	8
27	\u001b	[переключение кода]	57	\u0039	9
28	\u001c	[разделитель файлов]	58	\u003a	:
29	\u001d	[разделитель групп]	59	\u003b	;
30	\u001e	[разделитель записей]	60	\u003c	<
31	\u001f	[разделитель сегментов]	61	\u003d	=
32	\u0020	[пробел]	62	\u003e	>
33	\u0021	!	63	\u003f	?
34	\u0022	'' (прямые кавычки)	64	\u0040	@
35	\u0023	#	65	\u0041	A
36	\u0024	$	66	\u0042	B
37	\u0025	%	67	\u0043	C
38	\u0026	&	68	\u0044	D
39	\u0027	' (прямой апостроф)	69	\u0045	E
40	\u0028	(	70	\u0046	F
41	\u0029	)	71	\u0047	G

Dec	Unicode	Описание	Dec	Unicode	Описание
72	\u0048	H	106	\u006a	j
73	\u0049	I	107	\u006b	k
74	\u004a	J	108	\u006c	l
75	\u004b	K	109	\u006d	m
76	\u004c	L	110	\u006e	n
77	\u004d	M	111	\u006f	o
78	\u004e	N	112	\u0070	p
79	\u004f	O	113	\u0071	q
80	\u0050	P	114	\u0072	r
81	\u0051	Q	115	\u0073	s
82	\u0052	R	116	\u0074	t
83	\u0053	S	117	\u0075	u
84	\u0054	T	118	\u0076	v
85	\u0055	U	119	\u0077	w
86	\u0056	V	120	\u0078	x
87	\u0057	W	121	\u0079	y
88	\u0058	X	122	\u007a	z
89	\u0059	Y	123	\u007b	{
90	\u005a	Z	124	\u007c	|
91	\u005b	[	125	\u007d	}
92	\u005c	\	126	\u007e	~
93	\u005d	]	127	\u007f	[удалить]
94	\u005e	^	128	\u0080	управляющий символ
95	\u005f	_ (подчерк)	129	\u0081	управляющий символ
96	\u0060	` (тупое ударение)	130	\u0082	управляющий символ
97	\u0061	a	131	\u0083	управляющий символ
98	\u0062	b	132	\u0084	управляющий символ
99	\u0063	c	133	\u0085	управляющий символ
100	\u0064	d	134	\u0086	управляющий символ
101	\u0065	e	135	\u0087	управляющий символ
102	\u0066	f	136	\u0088	управляющий символ
103	\u0067	g	137	\u0089	управляющий символ
104	\u0068	h	138	\u008a	управляющий символ
105	\u0069	i	139	\u008b	управляющий символ

Dec	Unicode	Описание	Dec	Unicode	Описание
140	\u008c	управляющий символ	172	\u00ac	¬
141	\u008d	управляющий символ	173	\u00ad	-
142	\u008e	управляющий символ	174	\u00ae	®
143	\u008f	управляющий символ	175	\u00af	Ї
144	\u0090	управляющий символ	176	\u00b0	°
145	\u0091	управляющий символ	177	\u00b1	±
146	\u0092	управляющий символ	178	\u00b2	I
147	\u0093	управляющий символ	179	\u00b3	i
148	\u0094	управляющий символ	180	\u00b4	r
149	\u0095	управляющий символ	181	\u00b5	µ
150	\u0096	управляющий символ	182	\u00b6	¶
151	\u0097	управляющий символ	183	\u00b7	·
152	\u0098	управляющий символ	184	\u00b8	ё
153	\u0099	управляющий символ	185	\u00b9	№
154	\u009a	управляющий символ	186	\u00ba	є
155	\u009b	управляющий символ	187	\u00bb	»
156	\u009c	управляющий символ	188	\u00bc	j
157	\u009d	управляющий символ	189	\u00bd	S
158	\u009e	управляющий символ	190	\u00be	s
159	\u009f	управляющий символ	191	\u00bf	ї
160	\u00a0	[неразрывный пробел]	192	\u00c0	А
161	\u00a1	Ў	193	\u00c1	Б
162	\u00a2	ў	194	\u00c2	В
163	\u00a3	J	195	\u00c3	Г
164	\u00a4	¤	196	\u00c4	Д
165	\u00a5	Ґ	197	\u00c5	Е
166	\u00a6	¦	198	\u00c6	Ж
167	\u00a7	§	199	\u00c7	З
168	\u00a8	Ё	200	\u00c8	И
169	\u00a9	©	201	\u00c9	Й
170	\u00aa	ª	202	\u00ca	К
171	\u00ab	«	203	\u00cb	Л

Dec	Unicode	Описание	Dec	Unicode	Описание
204	\u00cc	М	230	\u00e6	ж
205	\u00cd	Н	231	\u00e7	з
206	\u00ce	О	232	\u00e8	и
207	\u00cf	П	233	\u00e9	й
208	\u00d0	Р	234	\u00ea	к
209	\u00d1	С	235	\u00eb	л
210	\u00d2	Т	236	\u00ec	м
211	\u00d3	У	237	\u00ed	н
212	\u00d4	Ф	238	\u00ee	о
213	\u00d5	Х	239	\u00ef	п
214	\u00d6	…	240	\u00f0	р
215	\u00d7	Ч	241	\u00f1	с
216	\u00d8	Ш	242	\u00f2	т
217	\u00d9	Щ	243	\u00f3	у
218	\u00da	Ъ	244	\u00f4	ф
219	\u00db	Ы	245	\u00f5	х
220	\u00dc	Ь	246	\u00f6	ц
221	\u00dd	Э	247	\u00f7	ч
222	\u00de	Ю	248	\u00f8	ш
223	\u00df	Я	249	\u00f9	щ
224	\u00e0	а	250	\u00fa	ъ
225	\u00e1	б	251	\u00fb	ы
226	\u00e2	в	252	\u00fc	ь
227	\u00e3	г	253	\u00fd	э
228	\u00e4	д	254	\u00fe	ю
229	\u00e5	е			

Алфавитный указатель

Специальные символы

& (амперсанд)
  и конкатенация строк, 340
  код, 484
' (апостроф), 340
&&, логическое И, 50, 51
$&, код подстановки, 381
` (кавычки)
  код URL-кодирования, 484
! (восклицательный знак), код, 484
!, логическое НЕ, 50, 51
!=, логический оператор неравенства, 42, 51
!==, оператор неравенства, 42
" (двойные кавычки), код, 484
(знак фунта), 108
  код, 484
(), круглые скобки
  и условные операторы, 50
  как метасимволы регулярных выражений, 371
  код URL-кодирования, 484
+ (знак плюс)
  как метасимвол регулярного выражения, 370
    нежадные шаблоны, 383
  код URL-кодирования, 484
  оператор конкатенации, 337
++, оператор инкремента, 40
+=
  оператор, добавление текста, 270
  оператор конкатенации с присваиванием, 337
  составной оператор присваивания, 39
- (дефис)
  код URL-кодирования, 484
* (звездочка)
  в полях для ввода пароля, 267
  и загрузка данных из внешних источников, 501
  как метасимвол регулярных выражений, 370
    нежадные шаблоны, 383
  код URL-кодирования, 484
@ (знак «эт»)
  добавление атрибутов в элементы XML, 513
  чтение атрибутов XML, 520
$ (знак доллара)
  как метасимвол регулярных выражений, 370
  код, 484
$$, код подстановки, 381
$`, код подстановки, 381
$', код подстановки', 381
% (знак процента), код, 484
––, оператор декремента, 40
_ (символ подчеркивания), имена закрытых/защищенных свойств, 73
– (тире), сопоставление с шаблоном, 269
. (точка), 108
  как метасимвол регулярных выражений, 370
  объекты XML, добавление элементов, 508, 513
  сохранение файлов классов, 70
, (запятая), 108
  код URL-кодирования, 484
/ (прямой слэш),
  и хранение данных, 466
/* */, комментарии, 32
//, комментарии, 32
/=, составной оператор присваивания, 39
; (точка с запятой), 32
  и конкатенация строк, 339

> (больше чем), оператор
  threshold(), метод, в качестве параметра, 255
  операторы равенства, 44
>>, побитовая операция, 103
>>>, побитовая операция, 103
< (меньше чем), оператор, 44
<<, побитовая операция, 103
<=, оператор, 44
= (знак равенства)
  выражения присваивания и, 39
    отличия от ==, 42
  переменные, присваивание, 31
*=, составной оператор присваивания, 39
−=, составной оператор присваивания, 39
===, оператор равенства, 42
? : (тернарный условный оператор), 45
*?, выражение, 384
? (знак вопроса)
  как метасимвол регулярного выражения, 370
??, выражение, 384
[], квадратные скобки, 133
  как метасимволы регулярных выражений, 371
\ (обратный слэш)
  сопоставление с шаблоном, 269
  регулярные выражения, 369
^ (знак вставки)
  побитовая операция, 103
  сопоставление с шаблоном, 269, 370
{ } (фигурные скобки), 29, 53
  и ассоциативные массивы, 165
  для операторов, 53
| (вертикальная черта)
  как метасимвол регулярных выражений, 371
  побитовое ИЛИ, 103, 155
||, логическое ИЛИ, 50, 51
J, британский фунт, 114
Ґ, йена, 114
€, знак евро (Евро), 114

Числа

32-разрядные значения цветов, 236

A

\A, метапоследовательность регулярных выражений, 371
<a>, HTML-тег, 271, 295
  гиперссылки, добавление в текст, 295
ab, параметр
  drawTriangle(), метод, 225
About Adobe Flash Player 9, пункт меню, 95
ac, параметр
  drawTriangle(), метод, 225
Action Message Format (AMF), 463
ActionScript
  fontStyle, свойство, 280
  интерпретатор, 32
  основы, 25
  функции, 545
ActionScript Project Wizard, 26
ActiveX, 88
  версия Flash Player для воспроизведения Flash в броузерах, 546
addCallback(), метод, 546
addChild(), метод, 186
  добавление элемента в список отображения, 172, 177
  текстовые поля, как сделать их видимыми, 264
addChildAt(), метод, 186
  добавление элемента в список отображения, 172
addEventListener(), метод, 35, 84, 203
  и повторяющиеся задачи, 56
addTo(), метод, 404
Adobe Flash Player Detection Kit, 86
alert(), функция (JavaScript), 545
allowDomain(), метод, 481
alpha, параметр
  lineStyle(), метод, 216
  MovieClip.beginFill(), метод, 228
ALPHA, свойство (класс BitmapDataChannel), 246
  noise(), метод, 248
  perlinNoise(), метод, 251
alphaMultiplier, свойство, 305
alphaOffset, свойство (класс ColorTransform), 304
alphas, параметр, 229

AMF (Active Messaging Format), протокол двоичной синхронной передачи данных, 463, 538
antiAliasType, свойство, 301
angle, параметр
  drawTriangle(), метод, 225
appendChild(), метод, 511
appendText(), метод
  добавление текста, 270
applyFilter(), метод (класс BitmapData), 256
arc, параметр
  drawArc(), метод, 221
ArgumentError, ошибка, 178, 551
Array, класс, 133
Array.CASEINSENSITIVE, константа, 154
Array.DESCENDING, константа, 153
Array.NUMERIC, константа, 154
Array.pop(),метод, 135
Array.push(), метод, 134
Array.RETURNINDEXEDARRAY, константа, 154
Array.UNIQUESORT, константа, 154
ArrayUtilities.duplicate(), метод, 146
ArrayUtilities.equals(), метод, 163
ArrayUtilities.findLastMatchIndex(), метод, 137
ArrayUtilities.findMatchIndex(), метод, 137
ArrayUtilities.findMatchIndices(), метод, 137
ArrayUtilities.max(), метод, 162
ArrayUtilities.min(), метод, 162
.as, файлы, 26
  определение версии Flash Player, 86
  специальные классы, создание, 69
AS3CBLibrary, библиотека, 215
/ascb, локальный каталог, 466
ascb.drawing.Pen, класс, 215
ascb.filters.ColorMatrixArrays.getSaturationArray(), метод, 316
ascb.play, пакет, 122
ascb.util, пакет, 107
  NumberFormat, класс, использование, 108
  и класс NumberUtiltites, 115
ascb.util.DateFormat, класс, 407
ascb.util.DateFormat.formatMilliseconds(), метод, 398
ascb.util.DateFormat.formatSeconds(), метод, 398
ascb.util.DisplayObjectUtilities, класс, 180
ascb.util.StringUtilities.trim(), метод, 360
ASCII
  и объекты Socket, 568
  преобразование в Unicode, 363
attachMovie(), метод, 168, 186
attachNetStream(), метод, 438
attributes(), метод, 520
autoSize, свойство, 272
AVM (ActionScript Virtual Machine), виртуальная машина ActionScript, 172

B

<b>, HTML-тег, 271
\b, символ возврата, 342
\B, негранцы слова, метапоследовательность, 372
\b, границы слова, метапоследовательность регулярных выражений, 372
backgroundColor, свойство, 266
baseX, параметр (метод perlinNoise()), 250
baseY, параметр (метод perlinNoise()), 250
beginGradientFill(), метод
  непрозрачные/прозрачные цвета, закрашивание форм, 228
  отрисовка формы во время выполнения, 229
  стили линий, настройка, 218
  треугольники, рисование, 225
BevelFilter, класс, 307
beginBitmapFill(), метод, 225
  непрозрачные/прозрачные цвета, закрашивание форм, 228
beginFill(), метод, 225, 228
bin, папка и создание проектов, 26
Bitmap, класс, 171
bitmap, параметр
  Graphics.beginBitmapFill(), метод, 231
BitmapData, класс, 234, 263
bitmapData, свойство, 239
BitmapDataChannel, класс, 246, 248
BitmapData, класс, 231

blockindent, атрибут (тег <textformat>), 271
BLUE, свойство (класс BitmapDataChannel), 246, 248
  perlinNoise(), метод, 251
blueMultiplier, свойство, 305
blueOffset, свойство (класс ColorTransform), 304
BlurFilter, класс, 307
border, свойство, 265
borderColor, свойство, 265
bottomScrollV, свойство (текстовые поля), 274
bounds, параметр (метод startDrag()), 207

, HTML-тег, 271
break, оператор, 48
  поиск соответствующих элементов массива, 137
browse(), метод, 554
  и передача файлов, 557
  фильтрация файлов для отображения, 555
bufferTime, свойство, 444
Button, класс, 215
button, экземпляр, 190
ByteArray, класс, 427
bytesLoaded, свойство
  ProgressEvent, класс, 493
  объект Sound, 414
bytesTotal, свойство (объект Sound), 414

C

call(), метод, 541
  JavaScript-функции, вызов, 544
  ответы Flash Remoting, обработка, 542
cancel, событие, 552, 556
caps, параметр (метод lineStyle), 217
CardHand, объекты, 122
Cards, класс, 122
caretIndex, свойство, 292
case, ключевое слово, 47
catch, блоки, 64, 68
  метод download(), работа с ним, 551
  обработка ошибок сетевых соединений, 574
category, свойство (объекты Unit), 125
CDATA, тег, 535
CGI-сценарии, 489
channel, параметр
  noise(), метод, 248
  perlinNoise(), метод, 251
charAt(), метод, 357
charCodeAt(), метод, 364
chmod, команда, 393
clear(), метод, 217, 451, 462
click, событие (InteractiveObject), 188, 203
close(), метод, 410, 423, 448
  разрыв соединений с серверами, 574
ColdFusion, 113, 489, 530, 537
color, атрибут (тег <font>), 271
color, параметр
  lineStyle, метод, 216
  beginGradientFill()), метод, 229
color, свойство
  ColorTransform, класс, 304
  TextFormat, объект , 296
color, свойство CSS, 279
ColorMatrixFilter, класс, 314
  контрастность, изменение, 317
  насыщенность, изменение, 315
  оттенки серого, применение, 314
  яркость, изменение, 316
ColorTransform, объект, 105, 237, 303, 305
complete, событие (класс LoaderInfo), 196
concat(), метод, 146, 337, 340
  и многомерные массивы, 148
condenseWhite, свойство, 272
connect(), метод, 438, 472, 560, 563
connectToSocketServer(), метод, 575
Console, окно
  использование trace, 33
const, ключевое слово, 83
content, свойство (класс Loader), 201, 238
contentLoaderInfo, свойство, 196
ContextMenu, класс, 85
Converter, класс, 125
  измерение, преобразование единиц измерения, 130
convertWithLabel(), метод, 126, 131
ConvolutionFilter, класс, 309, 316
cookies, 455
copyChannel(), метод, 246
copyPixels(), метод, 244, 246
copySource, параметр (метод threshold()), 255, 256

Courier New, шрифт, 286
Courier, шрифт, 286
createCircle(), метод, 189
createEmptyMovieClip(), метод, 168
CSS (Cascading Style Sheets), каскадные таблицы стилей, 278
  встраивание шрифтов, 287
  гиперссылки, добавление в текст, 296
  задание шрифтов в текстовых полях, 286
currencyFormat(), метод, 113
curveTo(), метод
  дуги, рисование, 221
  кривые, рисование, 220
  линии, рисование, 218

D

\D, метапоследовательность регулярных выражений, 372
\d, метапоследовательность регулярных выражений, 372
dataFormat, свойство (метод URLLoader.load()), 524
Date, класс, 391
date, свойство (объект Date), 394
DateFormat, класс, 395
  получение дат из строк, 407
DateUtilities, класс, 401
DateUtilities.addTo(), метод, 404
day, свойство (объект Date), 394
DAYS, константа (класс DateFormat), 395
days, параметр (метод addTo()), 404
DAYS_ABBREVIATED, константа (класс DateFormat), 395
deal(), метод, 122
Debugger, пункт меню, 95
default, ключевое слово, 48
defaultTextFormat свойство, 284
Delegate, класс, 36
delete, оператор, 523
destBitmap, объект (метод threshold()), 253
destPoint, параметр
  applyFilter(), метод, 257
  threshold(), метод, 254
  метод pixelDissolve(), 260
discard(), метод
  моделирование карточной игры, 123
dispatchEvent(), метод, 84
display, свойство ActionScript, 279
DisplayObject, класс, 171
  взаимодействия с мышью, организация, 202
  добавление элементов в список отображения, 173
  и внешние ролики, 201
  простые кнопки, создание, 189
  специальные визуальные классы, создание, 186, 188
DisplayObject.mask, свойство, 232
DisplayObectContainer, класс, 172
  перемещение объектов на передний/задний план, 182
  удаление элементов из списка отображения, 178
DisplayObjectUtilities.removeAllChildren(), метод, 181
doubleClick, событие (InteractiveObject), 203
download(), метод, 549
  передача файлов, определение выбора, 552
downState, свойство (класс SimpleButton), 188
drag(), метод, 212
DraggableSprite, специальный визуальный класс, 211
draw(), метод
  моделирование карточной игры, 123
draw(),метод (класс BitmapData), 237
drawArc(), метод, 221
Drawing API, прикладной интерфейс рисования, 187
drawCircle(), метод, 223
drawRectangle(), метод, 67
drawRegularPolygon(), метод, 226
drawRoundRect(), метод, 222
drawTriangle(), метод, 225
drop(), метод, 212
DropShadowFilter, класс, 307
dropTarget, свойство (класс Sprite), 207
duplicate(), метод, 146
  и многомерные массивы, 149
duplicateMovieClip(), метод, 168
duration, значение метаданных, 440

E

E4X, синтаксис

поиск XML, 532
Eclipse IDE, 25
trace, использование, 33
параметры, отслеживание, 27
elapsed(), метод, 405
elements(), метод, 514
else, выражение, 46
else…if, выражение, 46
подбрасывание монеты, моделирование, 118
[Embed], тег метаданных, 195
и встраивание шрифтов, 287
endIndex, параметр
setSelection(), метод, 291
substring() и slice(), методы, 348
метод setTextFormat(), 285
endFill(), метод, 228
градиенты, закрашивание форм, 229
enterFrame, событие, 35, 55, 319, 333
и звуковые файлы, 415
ответ на события мыши/клавиатуры, 37
повторение задач в течение некоторого времени, 56
повторяющиеся во времени задачи, 58
растворение двух растровых изображений, 260
equals(), метод, 163
ErrorReportingEnable, переменная (mm.cfg), 34
event, объект, 36
event, протокол, 297
EventDispatcher, класс, 84, 417
обработка событий, 35
exactFit, режим, 92
exec(), метод (класс RegExp), 374, 377
extends, ключевое слово, 78
ExternalInterface, класс, 544
ExternalInterface.addCallback(), метод, 546
ExternalInterface.call(), метод, 544

F

\f, символ перевода страницы, 342
face, атрибут (тег <font>), 271
false, логическое нет, создание растровых изображений, 235
fault, событие
слушатели, 540
FileFilter, массив объектов, 555
FileReference, объект, 553, 556
fillColor, параметр, 235
метод MovieClip.beginFill(), 228
метод pixelDissolve()), 260
fillRect(), метод, 242
fillColor, параметр
filters, свойство, 307
finally, блок, 67
findLastMatchIndex(), метод, 138
findMatchIndex(), метод, 138
findMatchIndices(), метод, 138
Firefox, 88
.fla, файлы, 27, 70
определение версии Flash Player, 86
файлы классов, сохранение, 74
Flash Player
как подсказать пользователю изменить настройки, 98
определение версии, 85, 87
сокрытие пунктов меню, 95
тип, проверка, 88
Flash Player Detection Kit, 86, 87
Flash Player Settings Manager, 468
Flash Remoting, сетевой интерфейс, 541
Flash Video Exporter, утилита кодировки, 446
Flash-cookies, 456
flash.display, пакет, 30, 168, 235
flash.display.CapsStyle, класс, 217
flash.display.Graphics, класс, 215
flash.display.JointStyle, класс, 217
flash.display.LineScaleMode, класс, 216
flash.display.Loader, класс, 195, 238
flash.display.Stage, класс, 85
flash.display.TexfieldType, класс, 266
flash.display.SpreadMethod, константа, 230
flash.errors, пакет, 65
flash.events, 84
flash.events.DataEvent.DATA constant, константа, 568
flash.events.Event, класс, 277
flash.events.FocusEvent, объекты, 292
flash.events.TextEvent, тип, 297
flash.events.TextEvent.TEXT_INPUT, константа, 294
flash.filters, пакет, 307
flash.filters.ColorMatrixFilter, класс, 314

flash.filters.ConvolutionFilter, класс, 309
flash.geom.ColorTransform, класс, 303
flash.geom.Matrix, класс, 237, 306
flash.geom.Matrix, объект, 229
flash.geom.Rectangle, класс, 242, 254
  копирование пикселов, 244
flash.geom.Rectangle, объект, 298
flash.media.ID3Info, класс, 417
flash.media.Video, класс, 438
flash.net.FileReference, класс, 550
flash.net.navigateToURL(), метод, 526
flash.net.NetConnection, класс, 541
flash.net.registerClassAlias(), метод, 463
flash.net.Responder, объект, 543
flash.net.sendToURL(), метод, 526
flash.net.URLLoader, класс, 195, 280, 392, 491
flash.net.URLStream, экземпляр, 494
flash.net.XMLSocket, класс, 559
flash.system.Capabilities, класс, 85
  видеовозможности, определение, 97
flash.system.Capabilities, объект, 91
flash.system.Capabilities.hasAudio, свойство, 96
flash.system.Capabilities.language, свойство, 89
flash.system.Capabilities.os, свойство, 87
flash.system.Capabilities.version, свойство, 86
flash.system.IME, класс, 89, 90
flash.system.IME.enabled, свойство, 90
flash.system.Security, класс, 85
flash.system.Security.allowDomain( ), метод, 100
flash.system.Security.allowInsecureDomain( ), метод, 100
flash.system.Security.showSettings( ), метод, 98
flash.text, пакет
  TextField, класс, 264
flash.text.AntiAliasType.ADVANCED, значение текстового поля, 301
flash.text.GridFitType, класс, 301
flash.text.TextField, класс, 264
flash.text.TextFieldAutoSize, класс, 273
flash.utils, пакет, 57
flash.utils.getTimer(), функция, 402
flash.utils.Timer, класс, 400
FlashVars, параметр, 547
Flex Builder, 25, 70, 538
  и удаление пробелов, 362
  файлы классов, сохранение, 74
floodFill(),метод, 243
flush(), метод, 459, 563, 575
.flv, файлы, 437
FocusEvent.preventDefault(), метод, 293
focusIn, событие, 292
focusOut, событие, 292
<font>, HTML-тег, 271
  встраивание шрифтов, 287, 288
  задание шрифтов в текстовых полях, 285
font, свойство (объект TextFormat), 285
font-family, свойство CSS, 279
fontFamily, свойство ActionScript, 279
font-size, свойство CSS, 279
font-style, свойство CSS, 280
fontSize, свойство ActionScript, 279
font-weight, свойство CSS, 280
fontWeight, свойство ActionScript, 280
for, ключевое слово, 52
for, оператор
  ассоциативные массивы, чтение, 166
  массив, перебор всех членов, 135
  многократное повторение операций, 52
  поиск соответствующих элементов массива, 137
  удаление элементов массивов, 142
format(), метод, 108
  даты и время, 397
  форматирование чисел для отображения без масок, 112
FormatSeconds(), метод, 399
focalPointRatio, параметр
  beginGradientFill(), метод, 230
fps, частота (кадры в секунду), 27
fractal, параметр (метод perlinNoise()), 251
Friedl, Jeffrey (Фридл, Джеффри), 374
fromCharCode(), метод, 78
fscommand(), функция, 470
fullYear, свойство (объект Date), 394
function, ключевое слово, 73

G

g (global), флаг регулярного выражения, 373, 376
  и метод replace(), 380
  и метод search(), 379

gamescore.cfm, 530
getCategories(), метод, 131
getCharBoundaries(), метод, 298
getCharIndexAtPoint(), метод, 298
getChildAt(), метод, 182, 186
getChildIndex(), метод, 182, 186
getColor(), метод, 201
getConverterFrom(), метод, 131
getConverterTo(), метод, 125, 131
getFirstCharInParagraph(), метод, 298
getLineIndexAtPoint(), метод, 298
getLineIndexOfChar(),метод, 298
getLineLength(), метод, 298
getLineMetrics(), метод, 298
getLineOffset(), метод, 298
getLineText(), метод, 298
getLocal(), метод, 456, 465
getNextHighestDepth(), метод, 170
getObjectsUnderPoint(), метод, 212
getParagraphLength(), метод, 298
getPixel(), метод, 240
getPixel32(), метод, 240
getTimer(), метод, 401
getTimer(), функция, 57
.gif, файлы, 194
global (g), см. g (global)
GlowFilter, класс, 307
GMT (Greenwich Mean Time), среднее время по гринвичскому меридиану), 391
GradientBevelFilter, класс, 307
GradientGlowFilter, класс, 307
Graphics, класс, 215
Graphics, объект
  стили линий, настройка, 216
Graphics.lineGradientStyle(), метод, 217
Graphics.lineTo(), метод, 218
Graphics, класс, 233
Graphics.beginBitmapFill(), метод, 231
Graphics.beginFill(), метод, 227
Graphics.curveTo(), метод, 220
Graphics.drawRect(), метод, 221
Graphics.drawRoundRect(), метод, 221
Graphics.drawRoundRectComplex(), метод, 221
Graphics.endFill(), метод, 227
grayscale, параметр
  perlinNoise(), метод, 251
  метод noise(), 248
gradientType, параметр
  beginGradientFill(), метод, 229
GREEN, свойство (класс BitmapDataChannel), 246
  noise(), метод, 248
  perlinNoise(), метод, 251
greenMultiplier, свойство, 305
greenOffset, свойство (класс ColorTransform), 304
gridTypeFit, свойство, 301

H

handleClick(), метод, 191
handleComplete(), метод, 197
handleOpen(), метод, 197
handleProgress(), метод, 197
hasAudio, свойство
  определение аудиовозможностей, 96
hasEmbeddedVideo, свойство, 97
hasMP3, свойство
  определение аудиовозможностей, 96
hasStreamingVideo, свойство, 97
hasVideoEncoder, свойство, 97
high, параметр (метод noise()), 247
hitTestPoint(), метод, 211
hitTestState, свойство (класс SimpleButton), 188, 191
host, параметр
  (метод connect()), 561
hours, параметр (метод addTo()), 404
hours, свойство (объект Date), 394
HTML (HyperText Markup Language), язык разметки гипертекста
  <textformat> тег, 271
HTML (HyperText Markup Language), язык разметки гипертекста
  <a>, тег, 271
  <b>, тег, 271

, тег, 271
  <font>, тег, 271, 285, 288, 295
  <i>, тег, 271
  <img>, тег, 271
  , тег, 271
  <p>, тег, 271
  <u>, тег, 271
  вывод на экран, 271
  загрузка блока текста, 491
  передача параметров, 547
  форматирование текста, 278
  шрифты, задание в текстовых полях, удаление тегов с помощью регулярных выражений, 383

htmlText, свойство, 271
гиперссылки, добавление в текст, 295
http, протокол, 296
https, протокол, 296
httpStatus, событие сбоя, 486, 196

I

<i>, HTML-тег, 271
i (ignoreCase), флаг регулярного выражения, 373, 378
id3, свойство (объект Sound), 417
ID3, тег звукового файла, 417
чтение, 418
if, выражение, 45, 50
ignoreCase (i), флаг, 378
IllegalOperationError, ошибка, 65, 551
IMAP, протокол, 560
IME (Input Method Editor), редактор методов ввода), 90
<img>, HTML-тег, 271
import, директива, 29
#include, директива, 29
indent, атрибут (тег <textformat>), 271
index, параметр (метод setTextFormat()), 285
indexOf(), метод, 343, 347
init, событие (класс LoaderInfo), 196
и внешние ролики, 201
insertChildAfter(), метод, 508, 511
insertChildBefore(), метод, 508, 511
int, числовой тип, 55, 102
InteractiveObject, класс, 202
InteractiveObject, отображаемый объект, 202
internal, ключевое слово, 31, 72
и многократное использование кода, 59
Internet Explorer, подключаемый модуль для, 88
innerRadius, параметр
Pen.drawStar(),метод, 227
interpolationMethod, параметр
beginGradientFill(), метод, 230
ioError, событие (класс LoaderInfo), 196
ioError, событие сбоя, 486
IOErrorEvent, событие, слушатель, 552
isNaN(), функция, 41
ISO-639-1, коды языков в, 89

J

Java, 537
JavaScript
Flash Player, определение версии, 86
передача параметров из HTML, 548
функции ActionScript, вызов, 546
функции, вызов, 544
join(), метод, 355
изменение порядка в строках, 362
преобразование массивов в строки, 145
удаление свободного места, 361
joints, параметр (метод lineStyle), 217
.jpg, файлы, 194

K

kerning, свойство ActionScript, 280
kerning, свойство CSS, 280
KeyboardEvent, класс, 38
keyDown, событие, 38
keyFocusChange, событие, 293

L

label, свойство (объекты Unit), 125
labelPlural, свойство (объекты Unit), 125
lastIndexOf(), метод, 343, 347
leading, атрибут (тег <textformat>), 271
leftmargin, атрибут (тег <textformat>), 271
length, параметр
substr(), метод, 348
length, параметр
drawRegularPolygon(), метод, 226
letterSpacing, свойство ActionScript, 280
letter-spacing, свойство CSS, 280
LIFO (Last In First Out), последним пришел, первым вышел), 135
, HTML-тег, 271
limit (метод split()), 144
lineGradientStyle(), метод, 217
lineStyle(), метод, 216
lineTo(), метод, 218
lineTo(), метод
прямоугольники, рисование, 222
load(), метод, 195, 199, 423, 485, 524
Loader, класс, 172, 194
внешние изображения, загрузка как растровых изображений, 238

внешние ролики, загрузка/взаимодействие с ними, 198, 202
и взаимодействия с мышью, 203
LoadVars, класс, 488
LoadVars.load(), метод, 488, 491
загрузка переменных из текстового файла, 485
LoadVars.sendAndLoad(), метод, 498
loadWSDL(), метод, 539
LocalConnection, класс, 202, 471
LocalConnection.connect(), метод, 471
LocalConnection.send(), метод, 475
Locale, класс, 110
Locale.slanguage, свойство, 111
Locale.svariant, свойство, 111
localToGlobal(), метод, 205
localX, свойство (класс MouseEvent), 205
localY, свойство (класс MouseEvent), 205
lockCenter, параметр (метод startDrag()), 207
low, параметр (метод noise()), 247
LSO, 455
LSO(Local Shared Objects), локальные совместно используемые объекты
запись данных, 457

M

m (multiline), флаг регулярного выражения, 373
Macromedia XML (MXML), 26, 70
файлы классов, сохранение, 74
mailto, ссылки, 296
marginLeft, свойство ActionScript, 280
margin-left, свойство CSS, 280
marginRight, свойство ActionScript, 280
margin-right, свойство CSS, 280
mask, параметр (метод threshold()), 254
Math, класс, 60, 78
Math.atan2(), функция, 329, 332
Math.ceil(), метод, 106, 440
Math.cos(), 332
Math.cos(), метод, 127
Math.cos(), функция, 329
Math.floor(), метод, 106, 440
Math.PI, 83
Math.pow(), метод, 126
Math.random(), метод, 115
расположение элементов массива в случайном порядке, 161
Math.round(), метод, 106, 440
Math.sin(), метод, 127
Math.sin(), функция, 329, 332
Math.sqrt(), метод, 126
Matrix, класс
отрисовка объектов в растровых изображениях, 237
сдвиг отображаемых объектов, 306
matrix, параметр
beginGradientFill(), метод, 229
Graphics.beginBitmapFill(), метод, 231
maxChars, свойство, 270
maximum, параметр (метод NumberUtilities.random()), 115
maxScrollH, свойство (текстовые поля), 274
maxScrollV, свойство (текстовые поля), 274
MaxWarnings, переменная (mm.cfg), 34
MemoryError, ошибка, 65, 551
Microsoft Windows, 456
milliseconds, параметр (метод addTo()), 404
milliseconds, свойство (объект Date), 394
minimum, параметр (метод NumberUtilities.random()), 115
minutes, параметр (метод addTo()), 404
minutes, свойство (объект Date), 394
miterLimit, параметр (метод lineStyle), 217
mm.cfg, файл, 34
mm.cfg, файл, размещение в Mac OS X, 34
month, свойство (объект Date), 394
MONTHS, константа (класс DateFormat), 395
months, параметр (метод addTo()), 404
MONTHS_ABBREVIATED, константа (класс DateFormat), 395
mouseDown, событие (InteractiveObject), 203
MouseEvent, класс, 37, 202, 206
перенос объектов методом drag-and-drop, 211
MouseEvent.CLICK, 191
MouseEvent.MOUSE_UP, 83
mouseFocusChange, событие, 293
mouseMove, событие (InteractiveObject), 203
mouseOut, событие (InteractiveObject), 203

mouseUp, событие (InteractiveObject), 203
mouseWheel, событие (InteractiveObject), 203
mouseWheelEnabled, свойство, 277
mouseX, свойство (класс DisplayObject), 202, 204
mouseY, свойство (класс DisplayObject), 202, 204
moveTo(), метод, 218
MovieClip, класс, 168, 172, 215
  добавление элементов в список отображения, 173
  и взаимодействия с мышью, 203
  и специальные визуальные классы, 186
Mozilla, 88
MXML (Macromedia XML), 26, 70
  файлы классов, сохранение, 74
mxmlc, 75
  аргументы командной строки, 28
mx.rpc.events.FaultEvent, 540
mx.rpc.soap.WebService, объект, 538
mx.utils.StringUtil.trim(), метод, 360

N

\n, символ перевода строки, 342, 372
$n, код подстановки, 381
{n}, метапоследовательность регулярных выражений, 371
{n,}, метапоследовательность регулярных выражений, 371
{n,m}, метапоследовательность регулярных выражений, 371, 383
name, свойство (объекты Unit), 125
NaN, константа (нечисло), 43
navigateToURL(), метод, 545
Navigator, окно (Flex Builder 2), 27
.NET, 537
NetConnection, класс, 541
NetStream, класс, 438
New ActionScript Project Wizard, 26
new, оператор, 173
$nn, код подстановки, 381
NNTP, 560
noBorder, режим, 92
noise(), метод, 247
  BitmapData, класс, 241
  и метод perlinNoise, 249
noScale, режим, 92
Number, ключевое слово, 55
number, параметр (метод NumberUtilities.round()), 107
number, числовой тип, 102
NumberFormat, класс, 108
  денежные суммы, форматирование, 113
  форматирование чисел для отображения без масок, 112
NumberFormat.currencyFormat(), метод, 113
NumberUtilities.getUnique(), метод, 124
NumberUtilities.random(), метод, 115
  игра в кости, моделирование, 120
  подбрасывание монеты, моделирование, 117
NumberUtilities.round(), метод, 107
numChildren, свойство, 180
numLines, свойство, 298
numPixels, параметр (метод pixelDissolve()), 260
  задание, 262

O

Object, класс, 164
Object.registerClass(), метод, 463
octaves, параметр (метод perlinNoise()), 250
offsets, параметр (метод perlinNoise()), 252
On2 Flix, утилита кодировки, 446
onCuePoint(), метод обратного вызова, 446
onLoad(), метод, 491
onMetaData(), метод, 440, 446
onMouseDown, метод, 37
onMouseUp, метод, 37
onTextScroll(), метод, 277
open, событие (класс LoaderInfo), 196
operation, параметр (метод threshold()), 254
outerRadius, параметр Pen.drawStar(), метод, 227
Outline, окно (Flex Builder 2), 27
override, атрибут, 81
overState, свойство (класс SimpleButton), 188

P

<p>, HTML-тег, 271

package, ключевое слово, 29
parse(), метод, 407
parseInt(), функция, 103
  представление в разных системах счисления, 104
password, свойство, 267
pattern, параметр
  replace(), метод, 380
  String.replace(), метод, 355
pause(), метод, 447
Pen, класс, 215
  многоугольники, рисование, 226
  треугольники, рисование, 225
  эллипсы, рисование, 224
Pen.drawArc(), метод, 221
Pen.drawEllipse(), метод, 224
Pen.drawLine(), метод, 219
Pen.drawRegularPolygon(), метод, 226
Pen.drawStar(), метод, 227
Pen.drawTriangle(), метод, 224
Perl, 489, 537
Perlin, Ken (Перлин, Кен), 249
perlinNoise(), метод, 253
PHP, получение текущих даты и времени, 392
pixelDissolve(), метод (класс BitmapData), 260, 263
pixelHinting, параметр (метод lineStyle), 216
place(), метод, 211
play(), метод, 410, 423
  и тег ID3, 417
  определение момента завершения воспроизведения звука, 419
  смещение начала воспроизведения звука, 412
  циклическое повторение звуков, 414
.png, файлы, 194
points, параметр
  Pen.drawStar(), метод, 227
pop(), метод, 141
POP3, протокол, 560
port, параметр
  (метод connect()), 561
position, свойство (класс SoundChannel), 423
prependChild(), метод, 511
Print, пункт меню, 95
private, ключевое слово, 31, 72
  многократное использование кода, 59
progress, событие, 553, 558
  URLLoader, класс, 493
  класс LoaderInfo, 196
ProgressEvent, 553
Project Wizard, 26
protected, ключевое слово, 31, 72
  и методы надкласса, 81
  многократное использование кода, 59
  подклассы, создание, 79
public, ключевое слово, 30, 72
  и методы надкласса, 81
  и многократное использование кода, 59
  подклассы, создание, 79
push(), метод, 134

Q

Quality, пункт меню, 95
QuickTime, проигрыватели, 415

R

\r, символ возврата каретки, 342, 372
radix, параметр (функция parseInt()), 104
random(), метод
  игра в кости, моделирование, 120
  подбрасывание монеты, моделирование, 117
RangeError, ошибка, 175, 179, 185
radius, параметр
  drawCircle(), метод, 223
  метод drawArc(), 221
radialLines, параметр
  drawArc(), метод, 221
ratios, параметр
  beginGradientFill(), метод, 229
readBoolean(), метод, 566
readByte(), метод, 565, 566
readChallenge(), метод, 571
readDouble(), метод, 566
readFloat(), метод, 566
readInt(), метод, 565, 566
readNormalProtocol(), метод, 571
readObject(), метод, 566
readShort(), метод, 566
readUnsignedByte(), метод, 566
readUnsignedInt(), метод, 566
readUnsignedShort(), метод, 566
readUTF(), метод, 566

readVersion(), метод, 571
RED, свойство (класс BitmapDataChannel), 246
  noise(), метод, 248
  perlinNoise(), метод, 251
redMultiplier, свойство, 305
redOffset, свойство (класс ColorTransform), 304
regexes (см. регулярные выражения), 367
RegExp, класс, 369
regexps (см. регулярные выражения), 367
RegExp.test(), метод, 374
registerClassAlias(), метод, 463
relatedObject, свойство, 292
removeChild(), метод, 178
removeChildAt(), метод, 178
replace(), метод, 355
replace, параметр
  replace(), метод, 380
  String.replace(), метод, 355
replaceSelectedText(), метод, 302
replaceText(), метод, 302
Responder, класс, 542
restrict, свойство
  фильтрация текстового ввода, 268
result, событие
  слушатель, 539
ResultEvent, параметр, 540
resume(), метод, 447
return, оператор, 62
reverse(), метод, 155
  изменение порядка в строках, 362
RexExp.exec(), метод, 374
repeat, параметр (метод Graphics.beginBitmapFill()), 231
RGB-значения цвета, 102
rightmargin, атрибут (тег <textformat>), 271
root.loaderInfo.parameters, свойство, 548
_rotation, свойство, 125
round(), метод, 78
  случайные числа, генерирование, 115
roundToInterval, параметр
  NumberUtilities.round(), метод, 107
  метод NumberUtilities.random(), 115
rotation, параметр
  createGradientBox(), метод, 230
  drawRegularPolygon(), метод, 226
  drawTriangle(), метод, 225
  Pen.drawStar(), метод, 227
RPC (Remote Procedure Calls), удаленные вызовы процедур, 526, 537
RSO (Remote Shared Objects), удаленные совместно используемые объекты, 455

S

s (dotall), флаг регулярного выражения, 373
S, метапоследовательность (один непробел), 372
s, метапоследовательность (один пробел), 372
_sans, группа шрифтов (группа шрифтов sans), 286
scaleMode, параметр (метод lineStyle), 216
scaleX, параметр (метод createGradientBox()), 230
scaleY, параметр
  createGradientBox(), метод, 230
screenResolutionX, свойство, 91
screenResolutionY, свойство, 91
ScriptTimeoutError, ошибка, формируемая автоматически, 65
scroll(), метод (класс BitmapData), 263
scrollH, свойство (текстовые поля), 274
scrollV, свойство (текстовые поля), 274
search(), метод, 378
seconds, параметр (метод addTo()), 404
seconds, свойство (объект Date), 394
securityError, событие (класс LoaderInfo), 196
securityError, событие сбоя, 486
SecurityErrorEvent, событие, слушатель, 552
SecurityPanel.CAMERA, значение (класс flash.system.SecurityPanel), 98
SecurityPanel.DEFAULT, значение (класс flash.system.SecurityPanel), 99
SecurityPanel.LOCAL_STORAGE, значение (класс flash.system.SecurityPanel), 99
SecurityPanel.MICROPHONE, значение (класс flash.system.SecurityPanel), 99
SecurityPanel.PRIVACY, значение (класс flash.system.SecurityPanel), 99

SecurityPanel.SETTINGS_MANAGER, значение (класс flash.system.SecurityPanel), 99
Security.showSettings(), метод, 468
seed, параметр
  noise(), метод, 247
  perlinNoise(), метод, 250
  метод pixelDissolve(), 260
seek(), метод, 449
  NetStream, класс, 442
select, событие, 552, 556
send(), метод, 575
_serif, группа шрифтов, 286
setChildIndex(), метод, 182, 186
  перемещение объектов методом drag-and-drop, 210
setColor(), метод, 201
setInterval(), функция, 56
setPixel(), метод, 240
  шаблоны шума, создание, 247
setPixel32(),метод, 240
setStyle(), метод, 280
setTextFormat(), метод, 278
  форматирование фрагментов текста, 284
setTimeout(), функция, 56
Settings, диалоговое окно, 98
Settings, пункт меню, 95
Shape, класс, 169, 172, 215
  специальные визуальные классы, 186
SharedObject, класс, 455
SharedObject, классы
  и хранящаяся информация, 456
SharedObject.clear(), метод, 458
SharedObject.flush(), метод, 458
sharpness, свойство, 301
shift(), метод, 141
Show All, пункт меню, 95
Show Redraw Regions, пункт меню, 95
showAll, режим, 92
showSettings( ), метод, 99
sides, параметр
  drawRegularPolygon(), метод, 226
SimpleButton, класс, 188
  и взаимодействия с мышью, 202
size, атрибут (тег <font>), 271
slice(), метод, 146, 348, 351
  и многомерные массивы, 148
smooth, параметр
  Graphics.beginBitmapFill(), метод, 231
SMTP (Simple Mail Transfer Protocol), простой протокол электронной почты, 560
SOAP (Simple Object Access Protocol), простой протокол доступа к объектам), 537
Socket, класс, отправка данных, 563
Socket, объекты, 561
  обработка ошибок сетевых соединений, 575
  получение данных, 565
Socket.close(), метод, 573
Socket.connect(), метод, 560
socketData, обработчик события, 570
sort(), метод, 153
  расположение элементов массива в случайном порядке, 160
  специальные сортировки, реализация, 158
sortOn(), метод, 153
Sound, объект, 408, 436
SoundChannel, класс, 419, 425
  отслеживание процесса воспроизведения звуков, 421
  приостановка/возобновление воспроизведения звуков, 423
SoundChannel.leftPeak, свойство, 425
SoundChannel.position, свойство, 420
SoundChannel.rightPeak, свойство, 425
soundComplete, событие, 419
Sound.length, свойство, 420
SoundLoaderContext, класс, 411
SoundMixer, класс, 426, 427
SoundMixer.computeSpectrum(), метод, 427
SoundTransform, класс, 425, 429
soundTransform, свойство, 439
sourceBitmap, объект (метод threshold()), 253
-source-path (mxmlc), параметр, 75
sourceRect, параметр (метод pixelDissolve()), 260
splice(), метод, 141
split(), метод, 351, 354
  извлечение символов по одному, 357
  изменение порядка в строках, 362
  расположение элементов массива в случайном порядке, 160

удаление и замена символов/слов, 355, 356
spreadMethod, параметр
beginGradientFill(), метод, 230
Sprite, класс, 30, 169, 172, 215
добавление элементов в список отображения, 173
и взаимодействия с мышью, 203
перемещение объектов с помощью мыши, 207
специальные визуальные классы, 186
Sprite.rotation, свойство, 103
SSL (Secure Socket Layer), протокол защищенных сокетов, 268
Stage.focus, свойство, 290
stage.scaleMode, свойство, 92
stage.showDefaultContextMenu, свойство, 95
start(), метод (addEventListener), 56
startDrag(), метод, 211, 450
startDrop(), метод, 207
startIndex, параметр
indexOf(), метод, 343
lastIndexOf(), метод, 345
setSelection(), метод, 291
substr(), substring() и slice(), методы, 348
метод setTextFormat(), 285
startingAngle, параметр
drawArc(), метод, 221
stage.align, свойство, 94
static, атрибут, 77
stitch, параметр (метод perlinNoise()), 251
stop(), метод (addEventListener), 56
stopAll(), метод, 426
stopDrag(), метод, 207
strict, флаг, 41
String(), функция, 338
String, класс, 336, 366
String, тип данных, 336
String.charAt(), метод, 357
String.charCodeAt(), метод, 363
String.concat(), метод, 337, 340
String.fromCharCode(), метод, 363
String.match(), метод, 378
String.replace(), метод, 355, 380
String.search(), метод, 378
String.split(), метод, 357, 501
StringUtils, класс, 30
Stubblebine,Tony (Стаббелбайн, Тони), 374
StyleSheet, объект, 278
StyleSheet, свойство, 278
substr(), метод, 348, 351
substring(), метод, 348, 351
substring, параметр
indexOf(), метод, 343
lastIndexOf(), метод, 345
super, ключевое слово, 82
swapDepths(), метод, 170
.swf, файлы, 26
определение версии Flash Player, 86
switch, выражение, 45, 50
systemFont, атрибут, 287

T

\t, символ табуляции, 342, 372
tabstops, атрибут (тег <textformat>), 271
test(), метод (класс RegExp), 374, 377
TextField, класс, 169, 172, 264
добавление элементов в список отображения, 173
и взаимодействия с мышью, 203
и загрузка внешнего содержимого, 197
удаление элементов из списка отображения, 179
TextField.password, свойство, 267
TextField.restrict, свойство, 268
TextField.setSelection(), метод, 291, 292
TexField.setTextFormat(), метод, 278
TextFormat, объект, 278
<textformat>, HTML-тег, 271
textAlign, свойство ActionScript, 280
text-align, свойство CSS, 280
textDecoration, свойство ActionScript, 280
text-decoration, свойство CSS, 280
textFormat, параметр (метод setTextFormat()), 285
TextFormat.align, свойство, 114
textIndent, свойство ActionScript, 280
text-indent, свойство CSS, 280
textInput, событие, 294
thickness, параметр (метод lineStyle), 216
threshold(), метод (класс BitmapData), 253

threshold, параметр (метод threshold()), 254
time, свойство (класс NetStream), 440
Timer(), метод, 401
Timer, класс, 56, 58
TimerEvent.TIMER, константа, 401
Times New Roman, шрифт, 286
Times, шрифт, 286
togglePause(), метод, 447
toLowerCase(), метод, 347, 359
  изменение регистра, 359
toString(), метод, 82, 104, 146, 338, 518
toUpperCase(), метод, 347
  изменение регистра, 359
trace(), функция, 33
  и загрузка внешних изображений, 198
TraceOutputFileEnable, переменная (mm.cfg), 34
TraceOutputFileName, переменная (mm.cfg), 34
transform.colorTransform, свойство, 303, 304, 305
try, блоки, 64, 68
  метод download(), работа с ним, 551
  обработка ошибок сетевых соединений, 574
tx, параметр
  createGradientBox(), метод, 230
ty, параметр
  createGradientBox(), метод, 230
type, параметр метода addEventListener(), 35
type, свойство, 266
_typewriter, группа шрифтов, 286

U

<u>, HTML-тег, 271
uint, значения типа
  задание значений свойства color, 304
uint, числовой тип, 102
underline, свойство (объект TextFormat), 296
Unicode, 289
  и преобразование строк, 363, 366
  коды для символов Latin 1, 576
Unit, класс, 125
  измерение, преобразование единиц измерения, 130
Unit.DEGREE, константа (класс Unit), 125
Unit.getUnits(), метод, 131
Unit.GRADIAN, константа (класс Unit), 125
Unit.RADIAN, константа (класс Unit), 125
unload, событие (класс LoaderInfo), 196
unnnn, метапоследовательность регулярных выражений, 372
updateAfterEvent(), метод, 211
upload(), метод, 557
UpperCase(), метод, 359
upState, свойство (класс SimpleButton), 188
URL (Universal Resource Locator), универсальный указатель ресурса
  переменные, загрузка из текстовых файлов, 485
URLLoader, класс, 485, 500
URLLoader, объект
  внешнее содержимое, загрузка, 195
URLLoader.load(), метод, 524, 526
URLRequest, класс, 409
  загрузка файлов, 551
URLRequest, экземпляры, 526
UTC (Universal Coordinated Time), всемирное координированное время), 391

V

var, ключевое слово, 31
  константы, создание, 83
VBScript, определение версии Flash Player, 86
Video, класс, 172, 444
void, объявление методов, 63

W

\w, метапоследовательность (буква, цифра или символ подчеркивания), 372
\W, метапоследовательность (не буква, цифра или символ подчеркивания), 372
WebService, объект, 538
while, выражение
  и метод indexOf(), 344
Windows 2000
  mm.cfg file, файл, размещение, 34

Windows XP
mm.cfg, файл, размещение, 34
Wizard (Project), 26
writeBoolean(), метод, 564
writeByte(), метод, 563
writeDouble(), метод, 564
writeFloat(), метод, 564
writeInt(), метод, 564
writeMultiByte(), метод, 564
writeObject(), метод, 564
writeShort(), метод, 564
writeUnsignedInt(), метод, 564
writeUTF(), метод, 564
writeUTFBytes(), метод, 563, 564
WSDL (Web Service Description Language), язык описания веб-сервисов), 538

X

x (extended), флаг регулярного выражения, 373
X, координаты
заливки области, создание, 243
и шум Перлина, 252
прокрутка растровых изображений, 263
X, координаты
прямоугольники, рисование, 222
x, ось
перемещение объектов, 319
x, параметр
drawEllipse(), метод, 224
drawCircle(), метод, 223
drawRegularPolygon(), метод, 226
drawTriangle(), метод, 225
Pen.drawStar(), метод, 227
метод drawArc(), 221
XML (eXtensible Markup Language), расширяемый язык разметки, 501, 536
HTML, использование в, 535
поиск, 535
текстовые узлы, 503
файлы политики, 101
атрибуты, 504
и вложенные узлы, 505
загрузка, 524
блока текста, 491
из других доменов, 525
корневой узел, 503
обход деревьев, 514
объекты
заполненные, 506
создание, 506
объекты сетевых соединений, 569, 573
отправка данных, 563
отправка, 526, 531
пакет, 502
поиск, 532
преимущества, 502
теги, 504
текстовые узлы, 511
узлы, 504
вложенные, 504
добавление в объекты, 512
элементы
добавление в объекты, 508
XMLList, объект, 517
чтение атрибутов, 520
XML-RPC, 526
XMLSocket, класс, 561
XMLSocket, объекты, 559
получение данных, 565
обработка ошибок сетевых соединений, 575
XMLSocket.close(), метод, 573
XMLSocket.connect(), метод, 560
XPath, 532
xRadius, параметр (метод drawEllipse()), 224

Y

Y, координаты
заливки области, создание, 243
и шум Перлина, 252
прокрутка растровых изображений, 263
прямоугольники, рисование, 222
y, ось
перемещение объектов, 319
y, параметр
drawEllipse(), метод, 224
drawCircle(), метод, 223
drawRegularPolygon(), метод, 226
drawTriangle(), метод, 225
Pen.drawStar(), метод, 227
метод drawArc(), 221
years, параметр (метод addTo()), 404
yRadius, параметр

drawEllipse(), метод, 224

Z

\Z, метапоследовательность регулярных выражений, 372
\z, метапоследовательность регулярных выражений, 372
Zoom In, пункт меню, 95
Zoom Out, пункт меню, 95

A

автономные проигрыватели, 88
альфа-канал, создание растровых изображений, 235
амперсанд (&)
  и конкатенация строк, 340
амплитуда, 425
анимация (программная), 318, 335
  перемещение объектов, 319
    заданные направления, 321, 323, 325
  применение техник к другим свойствам, 332
аргументы
  командная строка, 27
  методы, 61
ассоциативные массивы, 132
  создание, 164
атрибуты (XML), 503
  добавление в элементы XML, 512
  и вложенные узлы, 505
  удаление, 522
  чтение, 520, 522
аудиовозможности, определение, 96

Б

безопасность системы, 99, 101
больше чем (>)
  операторы сравнения, 44
британский фунт, см. J, 114
броузеры, фильтрация файлов для отображения, 555
Булевы значения
  возвращаемое операторами равенства, 41
  и перенос объектов методом drag-and-drop, 211
  и условные операторы, 51
  массивы, сравнение, 163
  подбрасывание монеты, и моделирование, 116
буферы
  видео, контроль/управление, 444
  задание для звука, 411

B

валюта, форматирование сумм, 113
варианты (коды стран), 111
веб-сервисы, 537, 543
  методы, вызов, 538
  обработка ответов, 539
  ошибки, обработка, 540
вектор скорости отображаемых объектов, 319
векторная графика, 234
версии проигрывателя, определение, 85, 87
взаимосвязь классов, 71
видео, 437
видеовозможности, определение, 97
видеоданные, 454
  время воспроизведения, чтение, 440
  длительность, чтение, 440, 442
  загрузка/воспроизведение, 437, 439
  звук, управление, 439
  контроль/управление буферизацией, 444
  масштабирование, 443
  область видеоизображения, 451
  остановка, 448
  перемотка, 449, 451
  приостановка/возобновление, 447
  пропускная способность, определение, 451, 454
  сигнальные точки, обнаружение, 446
  управление временем воспроизведения, 442
вложенные операторы for, 54
вложенные узлы, XML, 504
  и атрибуты, 505
внешние изображения
  время выполнения, загрузка, 194, 198
  растровые, загрузка, 238
внешние ролики, загрузка/взаимодействие с ними, 198, 202
внутренние переменные, 31
возобновление воспроизведения звука, 423, 425

восьмеричные литералы, 103
восьмеричные основания, 102
время, 391, 407
  DMYHMSM и миллисекунды Эпохи, преобразования, 399
  воспроизведения
    управление, 442
    чтение, 440
  всемирное координированное (UTC), 391
  вычисление интервалов между датами, 401, 406
  представление секунд/миллисекунд в виде минут и секунд, 398
  таймеры, использование, 400
  текущее, получение, 391, 394
  форматирование, 396, 398
выбор текста, реакция, 292
выдавливание, 311
выравнивание роликов, изменение, 93
выражение +?, 384
выражение {n,}?, 385
выражение {n,m}?, 385

Г

гиперссылки, 271
  вызов ActionScript из них, 297
  добавление в текст, 295, 297
гипотенуза, 126
  окружности, определение координат точек, 128
громкость звука, 429
групповые символы, 367
группы шрифтов, 286
Гука, закон, 327

Д

данные, отправка/загрузка, 483, 500
даты, 391, 407
  вычисление прошедшего времени/интервалов между, 401, 406
  значения даты, получение, 394
  названия дня/месяца, получение, 395
  получение из строки, 407
  текущие, получение, 391, 394
  форматирование, 396, 398
двоичная система, 102
двумерные массивы, 149
действия (операторы), 32
дерево иерархии (списки отображения), 169
деревья (XML), 503
диалоговые окна для сохранения, 550
динамические классы, 473
динамические текстовые поля, 266
документы (XML), 502
домены, загрузка файлов .swf из другого домена, 100
дуги, рисование, 221

Е

Евро см. €, 114
единицы измерения углов, преобразование, 124

Ж

жадные шаблоны, 383

З

загрузка файлов, 549, 552
заливка области, создание, 243
замедление, перемещение отображаемых объектов, 323
зарезервированные слова, 31
защищенные переменные, 31
звезды, рисование, 227
звук, 408, 436
  буферы, настройка, 411
  видео, 439
  громкость и стереобаланс, 429
  начало/прекращение, 410
  определение момента завершения воспроизведения, 418, 420
  отслеживание процесса воспроизведения, 420, 423
  получение размера, 414, 416
  прекращение воспроизведения, 426
  приложения, создание, 430
  приостановка/возобновление, 423, 425
  смещение начала, 412
  создание/загрузка объектов, 408, 410
  спектральный состав, чтение, 427, 429
  циклическое повторение, 413
  чтение уровня, 425
звуки ADCP, 96

знак вставки (^)
побитовая операция, 103
знак равенства (=)
и выражения присваивания, 39
операторы присваивания и отличия от ==, 42
переменные, присваивание, 31
значения, 164

И

И логическое (&&), 50, 51
игра в кости, моделирование, 120, 122
измерения, преобразование единиц измерения, 130
изображения
время выполнения, загрузка, 194, 198
растровые изображения, загрузка в них, 238
ИЛИ побитовое (|), 103, 155
ИЛИ логическое (||), 50, 51
имена пакетов, 71
индексные переменные (счетчики циклов), 52
индексы (массивов)
и ассоциативные массивы, 164
и многомерные данные, 150
удаление элементов, 142
индексы (массивы), 132
перебор всех членов, 134
инициализирующие выражения, 52
интерпретатор (ActionScript), 32
йена, см. ₹, 114

К

кавычки, 340
каналы, копирование, 246
карточная игра, моделирование, 122, 124
карты, моделирование игры, 122, 124
каталоги
размещение классов, 29
совместное использование данных приложениями Flash, 466
сохранение файлов классов, 74
килограммы, преобразование в фунты, 130
классы, 31
переменные, 31
присваивание имен, 72
размещение кода, 29
свойства, настройка, 27
специальные, 69, 84
создание, 69, 74
хранение, 74
ключевые кадры, 318
ключи (ассоциативные массивы), 132, 164
разделяемые запятыми, 165
строки, 144
кнопки, создание, 188, 194
код
многократно используемый, 58, 60, 62
размещение, 28, 33
коды URL-кодирования, 484
коды подстановки, 381
коды символов (Unicode), 363
коды стран (варианты), 111
коды языков, 110
команды, 32
компиляторы, 27
сохранение файлов классов, 74
компиляторы командной строки, 27
пути к классам, настройка, 75
композиция классов, 79
конкатенация (строк), 337, 340
константа нечисло (NaN), 43
константы, 83
конструкторы, 30
размещение кода, 29
контейнеры отображаемых объектов, 169
контрастность, изменение для отображаемых объектов, 317
контуры, создание вокруг текстовых полей, 265
контрольные точки, рисование кривых, 220
координаты по оси X, 127
координаты по оси Y, 127
координаты, образующие векторные изображения, 234
копирование значения (элементарные типы данных), 147
корневые узлы (XML), 503
края (отображаемого объекта), выделение, 312
кривые, рисование, 220
кривые Безье, 220, 223
круги

рисование, 223

Л

ленивые выражения, 384
линии, рисование, 218
литералы (регулярные выражения), 370
литеральная нотация (массивы), 133
логический оператор неравенства (!=), 51
логическое И (&&), 50
логическое ИЛИ (||), 50
логическое НЕ (!), 50
локальные совместно используемые объекты (LSO), 455
  совместное использование данных приложениями Flash, 465, 468
  создание/открытие, 456
  сохранение, 458, 461
  удаление данных, 462
  управление размерами, 468
  чтение данных, 461
локальные соединения
  и обмен информацией, 470
  обмен информацией между роликами
    и отправка данных, 476

М

маски, 215
  форматирование чисел для отображения без них, 112
массив объектов, 150
массивы, 132, 167
  ассоциативные
    создание, 164
    чтение элементов, 166
  вставка элементов в середину, 143
  даты и время, получение, 395
  копии, создание, 146, 150
  многомерные, хранение, 150, 153
  наименьший/наибольший элементы, получение, 161
  обращение/сортировка, 153, 158
  перебор всех членов, 135
  расположение в случайном порядке, 160
  с нулевым начальным индексом, 134
  с целочисленными индексами, 132
  соответствующие элементы, поиск, 136, 141
  специальные сортировки, реализация, 158, 160
  сравнение, 162
  строки, преобразование, 144
  удаление элементов, 141, 143
  элементы, добавление в начало/конец, 133, 135
массивы-дубликаты, 146
маски, 233
  создание, 232
математические операторы, 38, 41
математические операции, 102, 131
матрицы выдавливания, 311
матрицы выделения краев, 312
медиапроигрыватели, 415
меньше чем (<)
  операторы равенства, 44
местоположение пера
  линии, рисование, 218, 220
метаданные, определение свойств, 27
метапоследовательности, 369
  регулярных выражений, 371–372
метасимволы, 369
методы, 31
  convert(), 131
  fromCharCode(), 363
  parseCSS(), 281
  perlinNoise(), 249
  removeChild(), 181
  removeChildAt(), 181
  sort(), 158
  sortOn(), 158
  splice()
    вставка элементов в середину массивов, 143
  split()
    преобразование строк в массивы, 144
  String.split(), 144
  unshift(), 134
  веб-сервисы, вызов, 538
  выход, 62
  многократное использование кода, 58
  получение результатов, 63
  размещение кода, 29
  формирование для обеспечения многократного использования, 60, 62
миллисекунды Эпохи, 399
миллисекунды, форматирование, 398

минуты, форматирование, 398
многократно используемый код, 58, 60, 62
многомерные массивы, 148, 153
многоугольники, рисование, 226
модификаторы доступа, 59

Н

надклассы, 79
  реализация в подклассе версий, 81
наклон, 306
направление (ускорения), 325
наследование, определение, 79
настройки экрана, определение, 91
насыщенность, ее изменение для отображаемых объектов, 315
НЕ логическое (!), 50, 51
нежадные шаблоны, 383, 385
неопределенные элементы (массивов), 141
неравенство, проверка, 41, 44
непрозрачные цвета, закрашивание форм, 227
нули, начальные/конечные, вставка, 108, 111

О

область видимости, 31
обмен информацией между роликами
  локальные соединения, 470
    и отправка данных, 476
    подтверждение получения, 479
  на одном компьютере, 471
  отправка данных, 476, 478
обновляющие выражения, 52
обработка ошибок, 64, 68
обработчики, 31
обратный слэш
  сопоставление с шаблоном, 269
  экранирование кавычек и апострофов, 341
объект URLLoader, 145
объекты, 31
  XML
    заполненные, 506
    создание, 506
  замедление, 323
  звук, создание/загрузка, 408, 410
  изменение цвета, 334
  перемещение, 319
    перемещение на передний и на задний план, 181, 186
    перемещение с помощью мыши, 206, 214
    пружины, 327, 329
    рисование в растровых изображениях, 237
    ускорение, 325, 327
объявление методов, 73
объявления (XML), 503
округление чисел, 106, 107
окружности
  определение координат точек, 127, 130
оператор >=, 44
операторы, 32
операторы присваивания, 31, 39
  и операторы равенства, различия, 42
операторы равенства, 42
операторы цикла, 52
  массивы, 135
операционные системы, определение, 87
основания, представление, 102
основы, 68
отмена выбора текста, реакция, 292
отправка данных в серверный сценарий, 496
оттенки (цветов), 304
оттенки серого, 314

П

пакет XML, 502
пакеты, 70, 74
параллельные массивы, 150
параметр deleteCount (метод splice()), 143
параметр delimiter (метод split()), 144
параметры (метода), обобщение для многократного использования, 60
переменные, 31
  загрузка из серверного сценария, 488
перемещение объектов методом drag-and-drop, 206, 214
перемотка видеозаписи, 449
переход к элементам посредством оператора точка (.), 515
переход к элементам посредством оператора точка (dot down)
  текстовые узлы, чтение, 518
Перлин, Кен, см. Perlin, Ken

пикселы (растровые изображения), 234
  копирование, 244, 246
  работа с ними, 240, 242
Пифагора, теорема, 126
побитовое ИЛИ (|), 103, 155
побитовые операции, 103
повышение резкости, 313
подбрасывание монеты, моделирование, 116, 119
подкаталоги, сохранение файлов классов, 70
подклассы
  методы надкласса, реализация, 81
  создание, 78, 81
подстроки
  извлечение, 348, 351
  поиск, 343, 347
подтверждение установления связи, 570
поля ввода, 266
  максимальная длина, задание, 269
  пароль, 267
  пользовательский ввод
    форматирование, 283
  фильтрация ввода, 268
пользовательский ввод
  поля, 266
  реакция, 294
  регулярные выражения, проверка допустимости, 386, 390
  форматирование, 283
пороговое значение, 253, 256
портативные устройства, поддержка Flash Player, 91
последовательности переключения кода, 484
  Unicode для символов Latin 1, 576
постфиксные операторы, 40
преобразования, 303, 317
префиксные операторы, 40
прилежащие катеты, 128
приложения
  свойства, настройка, 27
приостановка воспроизведения звука, 423, 425
пробелы (текст)
  вставка специальных символов, 341
  начальные/конечные, вставка, 108, 111
  сокращение, 272
  удаление, 360, 362
проверяемые выражения, 52
прозрачные цвета, закрашивание форм, 227
программная анимация, 318, 335
прогрессивный .jpg, 194
проекты (Flex), 25
  создание, 26
промежуточные кадры, 318
пропускная способность, 451, 454
противолежащие катеты, 128
пружины, 327, 329
прямой слэш (/), и хранение данных, 466
прямоугольная область, 207
прямоугольники, рисование, 221, 223
прямоугольные заливки, создание, 242
прямоугольные треугольники, вычисление расстояния, 127

Р

работа с файлами, 549, 558
  загрузка, 549, 552
  определение момента выбора пользователем файлов для передачи, 556
  отслеживание процесса загрузки, 553
  отслеживание процесса передачи файла, 558
  передача файлов, 552, 557
  поиск файлов путем просмотра, 554
  фильтрация файлов для отображения, 555
равенство, проверка, 41, 44
размытие, 307
разработка сетевых приложений, 559, 575
  обработка ошибок сетевых соединений, 574
  отправка данных, 563, 565
  подтверждение установления связи, 570
  получение данных, 565
  разрыв соединений с серверами, 573
  соединение, 560, 563
растровые изображения, 234, 263
  внешние изображения,загрузка, 238
  закрашивание форм, 231
  заливка области, создание, 243
  каналы, копирование, 246
  копирование пикселов, 244, 246

объекты, рисование, 237
пикселы, работа с ними, 240, 242
прокрутка, 263
растворение одного в другом, 260, 263
фильтры, применение, 256, 259
шаблоны шума, создание, 247
шум Перлина, создание, 249, 253
расширенное форматирование текста, 297, 300
радиус эллипса
большой, 224
малый (вертикальный), 224
регистр, изменение, 359
регулярные выражения, 269, 367, 390
литералы, 369
нежадные шаблоны, 383, 385
пользовательский ввод, проверка допустимости, 386, 390
понимание шаблонов, 368, 374
разделение строк, 353
соответствия шаблону
поиск, 378, 379
удаление и замена символов/слов, 380, 383
тестирование, 374, 377
режимы масштабирования (роликов), 92
результаты работы сценария
переменные, обработка на стороне сервера, 498
рекурсия, 149
рисование, 215, 233
дуги, 221
звезды, 227
кривые, 220
круги, 223
линии, 218
многоугольники, 226
прямоугольники, 221
треугольники, 224
эллипсы, 224
ролики
внешние, загрузка/взаимодействие, 198, 202
выравнивание, изменение, 93
масштабирование, 92
обмен информацией, 470, 481, 482
и локальные соединения, 476
на одном компьютере, 471

C

свечение, 307
свойства, 31
ассоциативные массивы, 164
приложения, настройка, 27
свойства-множители, 304
сглаживание, 300
сдвиг, 306
секунды, форматирование, 398
серверы, загрузка файлов .swf с другого сервера, 99
сигнальные точки, 446
символы Latin 1, 576
символы переключения кода, 269
синтаксис E4X, 508
поиск элементов по имени, 515
текстовые узлы, добавление, 511
текстовые узлы, чтение, 517
системы счисления, представления, 104, 106
скорость отображаемых объектов, 319
скосы, 307
сложные данные, хранение, 150, 153
случайные числа, 115
смена родителя, 176
события, 31
обработка, 35
передача, 84
события клавиатуры, ответ, 36
события мыши
взаимодействия, организация, 202, 206
ответ, 36
перемещение объектов, 206, 214
события прокрутки, ответ на них, 277
совместно используемые объекты, 455
удаление данных, 462
чтение данных, 461
создание простых кнопок, создание интерактивных кнопок, 188, 194
сообщения, трассировка, 33, 34
сортировка массивов, 153, 158
специальная, реализация, 158, 160
составные операторы присваивания, 39
составные типы данных, 146
спектральный состав (звука), 427, 429
специальные классы, 69, 84
сериализация, 463, 465
создание, 69, 74
специальные пробельные символы, 341

специальный визуальный класс, создание, 186, 188
списки отображения, 168, 214
  добавление элементов, 172, 177
  объекты, перемещение на передний/задний план, 181, 186
  удаление элементов, 178, 181
среда выполнения, 85, 101
среднее время по гринвичскому меридиану (GMT), 391
Стаббелбайн, Тони см. Stubbelbine,Tony
статические методы, 60
  создание, 77
статические свойства, 77
стили линий
  градиент, 217
  настройка, 216
строгое равенство/неравенство, 41
строки, 336, 366
  Unicode, преобразование, 363, 366
  даты, синтаксический разбор, 407
  извлечение символов по одному, 357, 359
  изменение порядка слов/символов, 362
  кавычки и апострофы, 340
  массивы, преобразование, 144
  объединение, 337, 340
  подстроки
    извлечение, 348, 351
    поиск, 343, 347
  пробелы, удаление, 360, 362
  разбор по словам, 351, 354
  удаление и замена символов/слов, 354, 356
структура (XML), 503, 506
супер-cookies, 456
сцена, 169
сценарии CGI
  выполняющиеся на стороне сервера
    результаты обработки переменных, 498
  получение текущих даты и времени, 392
  права доступа, 393
счетчики циклов, 52

T

таймеры, 400
теги, XML, 504
текст, 264, 302
  HTML, вывод на экран, 271
  Unicode, вывод на экран, 289
  выделение средствами ActionScript, 291
  вывод на экран, 270
  гиперссылки, 295, 297
    вызов ActionScript из них, 297
  замена, 302
  изменение размеров полей, 272
  контуры вокруг, 265
  поворот, 288
  поля ввода, 266
    максимальная длина, задание, 269
    пароль, 267
    фильтрация, 268
  пользовательский ввод, форматирование, 283
  пробел, сокращение, 272
  прокрутка программная, 274, 277
  расширенное форматирование, использование, 297, 300
  реакция на пользовательский ввод, 294
  сглаживание, применение расширенного, 300
  события прокрутки, реакция, 277
  точки вставки, задание, 292
  установка/снятие фокуса, реакция, 292
  фон, создание, 265
  форматирование, 278, 283
    части, 284
  шрифты, 285
    встраивание, 287, 288
текстовые узлы, 503
  деревья, 514
  добавление, 511, 512
  удаление, 522
  чтение, 517, 520
текстовые файлы, загрузка переменных из них, 485
тело цикла, 53
тени, 307
тернарный условный оператор (? :), 45
типы данных, 44
  составные, 44
  элементарные, 44
точка с запятой (), 32
  и конкатенация строк, 339

точки вставки, задание, 292
треугольники
вычисление расстояния, 127
рисование, 224
тригонометрия, 329, 332

У

удаленное взаимодействие во Flash, 537, 543
методы, вызов, 541
ответы, обработка, 542
удаленные совместно используемые объекты (RSO), 455
узлы (XML), 503, 504
узоры шума, создание
Перлин, 249
уникальные числа, 124
ускорение отображаемых объектов, 325, 327
величина, 325
пружины, 327, 329
условные выражения, 45, 50
условные операторы, 45
устройство рендеринга, 172

Ф

файлы .as
размещение кода, 29
файлы .cfm
gamescores.cfm, 530
файлы политики, 100
Фаренгейт, преобразование в градусы Цельсия, 130
фигурные скобки ({ }), 29
и ассоциативные массивы, 165
фильтрация по условию, 532
фильтры, 247, 303, 317
выделение краев, 312
дополнительные эффекты, применение, 309, 311
повышение резкости отображаемых объектов, 313
применение базовых фильтров, 307, 309
применение к растровым изображениям, 256, 259
свертки, выдавливание отображаемых объектов, 311
цифровые негативы, создание, 314
формат DMYHMSM (дни, месяцы, годы, часы, минуты, секунды, миллисекунды), 399
формы
градиент, закрашивание, 228, 231
непрозрачные/прозрачные цвета, закрашивание, 227
растровые изображения, заливка, 231
Фридл, Джеффри см. Friedl, Jeffry
функции, 31
функции сравнения, 158
функция сортировки, 158
фунты, преобразование в килограммы, 130

Х

хранящиеся данные, 455, 469
хеш-таблицы, 164

Ц

цвет
внесение изменений, 303
восстановление, 305
закрашивание форм, 227
оттенки серого, применение, 315
оттенки, применение, 304
Цельсия градусы, преобразование в градусы Фаренгейта, 130
циклическое воспроизведение звука, 413
циклы for...each
чтение элементов XML, 521
цифровые негативы, 314

Ч

частота (fps, кадры в секунду), 27
числа, 102, 131
валюта, форматирование сумм, 113
вычисление расстояния между двумя точками, 126
единицы измерения углов, преобразование, 124
измерения, преобразование единиц, 130
маски, форматирование для отображения без них, 112
моделирование
карточная игра, 122, 124

подбрасывание монеты, 116, 119
округление, 106, 107
окружности, определение координат точек, 127, 130
основания, представление, 102
представления в разных системах счисления, 106
случайные, генерирование, 115
уникальные, генерирование, 124
числа с плавающей точкой
округление, 106
случайные, генерирование, 115
чувствительность к регистру
и методы replace() и split(), 356
и сортировка массивов, 153
изменение, 359
подстроки и поиск, 347
фильтрация текстового ввода, 268

Ш

шаблоны (см. регулярные выражения), 369
шаблоны шума, создание, 247
Перлин, 253
шифрование паролей, 267
шрифты
встраивание, 287, 288
поворот, 288
текстовые поля, задание, 285
шум Перлина, 249, 253, 263

Э

экземпляры (объект), 32
экраны мобильных телефонов, 91
экспресс-установка (Flash Player Detection Kit), 87
элементарные данные, 146
элементы (XML), 503
извлечение, 514
поиск по имени, 515, 517
удаление, 522
элементы (массивов)
вставка в середину, 143
добавление в начало/конец, 133, 135
их копии, 146, 150
многомерные/сложные данные, хранение, 150, 153
наименьший/наибольший, получение, 161
обращение/сортировка, 153, 158
перебор всех членов, 135
поиск соответствия, 136
расположение в случайном порядке, 160
создание, 164
удаление, 141, 143
чтение, 166
эллипсы, рисование, 224

Я

язык разметки гипертекста (см. HTML), 383
языки операционной системы, проверка, 89, 91
языки сценариев, 489
яркость, изменение для отображаемых объектов, 316

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-091-X, название «ActionScript 3.0. Сборник рецептов» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.