Изучение ACTIONSCRIPT® 3.0

Изучение ActionScript® 3.0

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Последнее обновление 11.5.2012.

Содержание

Глава 1. Введение в ActionScript 3.0

О языке ActionScript

ActionScript является языком программирования, используемым в средах выполнения Adobe® Flash® Player и Adobe® AIR™. Он обеспечивает интерактивность, обработку данных и многие другие возможности в содержимом Adobe Flash, Flex и AIR, а также в приложениях.

ActionScript выполняется виртуальной машиной ActionScript, AVM, которая является частью проигрывателя Flash Player и пакета AIR. Код ActionScript обычно преобразуется компилятором в формат байт-кода. (*Байт-код* — это тип языка программирования, написанный и распознаваемый компьютером.) Среди примеров компиляторов — компилятор, встроенный в Adobe® Flash® Professional, компилятор, встроенный в Adobe® Flash® Builder™, а также компилятор, доступный в компоненте Adobe® Flex™ SDK. Байт-код встроен в SWF-файлы, которые исполняет проигрыватель Flash Player и пакет AIR.

ActionScript 3.0 предлагает надежную модель программирования, знакомую разработчикам, имеющим базовые понятия об объектно-ориентированном программировании. Ниже перечислены некоторые из основных функций ActionScript 3.0, добавленные в процессе усовершенствования предыдущих версий ActionScript.

- Новая виртуальная машина ActionScript, называемая AVM2, использует новый набор инструкций в байт-кодах и обеспечивает значительный прирост производительности
- Более современный программный код компилятора выполняет оптимизацию на более высоком уровне, чем предыдущие версии компилятора.
- Расширенный и улучшенный интерфейс программирования приложений (API) с контролем объектов на нижнем уровне и подлинной объектно-ориентированной моделью
- Интерфейс программирования приложений XML создан на основе спецификации ECMAScript для XML (E4X) (ECMA-357 редакция 2). E4X является расширением языка ECMAScript, который добавляет XML в качестве поддерживаемого языком типа данных.
- Модель события на основе спецификации событий DOM уровня 3

Преимущества ActionScript 3.0

ActionScript 3.0 превосходит возможности создания сценариев предыдущих версий ActionScript. Специально разработан, чтобы облегчить создание сложных приложений с большим набором данных и объектно-ориентированным, многократно используемым программным кодом. ActionScript 3.0 не требуется для содержимого, выполняемого в проигрывателе Adobe Flash Player. Однако он обеспечивает возможности повышения производительности, которые доступны только в AVM2 (виртуальная машина ActionScript 3.0). Код ActionScript 3.0 может выполняться до десяти раз быстрее, чем код прежней версии ActionScript.

Предыдущая версия виртуальной машины ActionScript, AVM1, выполняет код ActionScript 1.0 и ActionScript 2.0. Flash Player 9 и 10 поддерживает AVM1 в целях обратной совместимости.

Новые возможности ActionScript 3.0

Хотя ActionScript 3.0 содержит много классов и функций, знакомых программирующим на ActionScript 1.0 и 2.0, архитектурная и концептуальная составляющие ActionScript 3.0 отличаются от предыдущих версий ActionScript. Улучшения в ActionScript 3.0 включают новые возможности языка ядра и усовершенствования в прикладном интерфейсе программирования, которые обеспечивают улучшенное управление объектами нижнего уровня.

Возможности языка ядра

Язык ядра определяет основные конструктивные элементы языка программирования, такие как инструкции, выражения, условия, циклы и типы. ActionScript 3.0 содержит много новых функций, которые ускоряют процесс разработки.

Исключения при выполнении

В ActionScript 3.0 предусмотрены сообщения о большем числе ошибок, чем в предыдущих версиях ActionScript. Исключения при выполнении используются для обработки общих ошибок, они улучшают возможности отладки разрабатываемых приложений, обеспечивая надежную обработку всех возникающих ошибок. Для возникающих при выполнении ошибок можно получить стек-трейсы, снабженные комментариями из исходного файла, а также информацией о номере строки, что позволяет быстро выявить ошибки.

Типы выполнения

В ActionScript 3.0 информация о типе сохраняется в среде выполнения. Эта информация используется для проверки типа выполнения, благодаря чему повышается уровень безопасности типа системы. Информация о типе также используется для создания машинного представления переменных, что повышает производительность и уменьшает использование памяти. Для сравнения, в ActionScript 2.0 аннотации типа являются в основном вспомогательным инструментом разработчика, а все значения динамически вводятся во время выполнения.

Запечатанные классы

В ActionScript 3.0 представлена концепция запечатанных классов. Запечатанный класс обладает фиксированным набором свойств и методов, определенных во время компиляции; дополнительные свойства и методы к нему добавить нельзя. Благодаря отсутствию возможности изменения класса во время выполнения обеспечивается более строгая проверка при компиляции и создание более надежных программ. Также оптимизируется использование памяти, поскольку для каждого экземпляра объекта не требуется отдельная внутренняя хеш-таблица. Возможно использование динамических классов с помощью ключевого слова `dynamic`. Все классы в ActionScript 3.0 по умолчанию «запечатаны», но могут быть объявлены динамическими с помощью ключевого слова `dynamic`.

Закрытия методов

ActionScript 3.0 позволяет при закрытии метода автоматически запомнить исходный экземпляр объекта. Эта возможность особенно полезна при обработке событий. В ActionScript 2.0 при закрытиях методов не запоминалось, из какого экземпляра объекта они извлекались, что приводило к непредсказуемому поведению при вызове закрытия метода.

ECMAScript для XML (E4X)

В ActionScript 3.0 реализован язык сценариев ECMAScript для XML (E4X), недавно стандартизированный как ECMA-357. E4X предлагает органичный, очень удобный набор языковых конструкций для работы с XML. В отличие от стандартных прикладных интерфейсов программирования для разбора XML в языке E4X обеспечивается поддержка XML как встроенного типа данных. E4X упрощает разработку приложений, работающих с XML, значительно сокращая длину программных кодов.

Для просмотра спецификации ECMA E4X перейдите по адресу www.ecma-international.org.

Регулярные выражения

ActionScript 3.0 включает встроенную поддержку регулярных выражений, что позволяет быстро выполнять поиск и обработку соответствующих строк. Поддержка регулярных выражений в ActionScript 3.0 реализована в соответствии с их определением в спецификации языка ECMAScript (ECMA-262) версии 3.

Пространства имен

Пространства имен аналогичны традиционным спецификаторам доступа, используемым для управления видимостью деклараций (`public`, `private`, `protected`). Они работают как пользовательские спецификаторы доступа, имена для которых можно задавать по своему выбору. Во избежание коллизий пространства имен задаются идентификаторами URI, которые также используются для представления пространств имен XML при работе в E4X.

Новые типы примитивов

ActionScript 3.0 содержит три числовые типа: Number, int и uint. Тип Number представляет число с двойной точностью и плавающей запятой. Тип int задает 32-разрядные целочисленные значения со знаком, позволяя коду ActionScript использовать преимущество быстрой обработки целочисленных математических операций центральным процессором. Тип int удобен для создания счетчиков циклов и целочисленных переменных. Тип uint задает 32-разрядные целочисленные значения без знака, его удобно использовать для определения значений цвета RGB, байтовых счетчиков и т. д. В отличие от этого, в ActionScript 2.0 доступен только один числовой тип — Number.

Функции прикладных интерфейсов программирования (API)

Прикладные интерфейсы программирования (API) в ActionScript 3.0 содержат много классов, позволяющих управлять объектами на нижнем уровне. Усовершенствована архитектура языка: она стала более удобной и понятной, чем в предыдущих версиях. И хотя существует слишком много классов, чтобы они могли быть рассмотрены подробно, нелишне отметить некоторые значительные отличия.

Модель событий DOM3

Модель событий Document Object Model Level 3 (DOM3) обеспечивает стандартный способ создания и обработки сообщений о событиях. Эта модель событий предназначена для обеспечения взаимодействия и связи объектов в приложении, поддержания их состояния и реакции на изменение. Модель событий ActionScript 3.0 создана на основе спецификации событий World Wide Web Consortium DOM Level 3. Эта модель обеспечивает более ясный и эффективный механизм, чем системы событий, доступные в предыдущих версиях ActionScript.

События и события обработки ошибок размещаются в пакете flash.events. В компонентах Flash Professional и среде Flex используется аналогичная модель событий, поэтому система событий унифицирована на платформе Flash Platform.

Прикладной программный интерфейс списка отображения

Прикладной программный интерфейс для доступа к списку отображения (дерево, содержащее все визуальные элементы приложения) состоит из классов для работы с визуальными примитивами.

Класс Sprite является облегченным блоком построения, используемым в качестве базового класса для визуальных элементов, например компонентов пользовательского интерфейса. Класс Shape воспроизводит векторные фигуры без сжатия. Экземпляры этих классов легко могут создаваться с помощью оператора `new`, а затем динамически переназначаться в любое время другим родительским объектам.

Управление глубиной осуществляется автоматически. Методы предоставляются для определения порядка наложения объектов и управления им.

Обработка динамических данных и содержимого

ActionScript 3.0 содержит механизмы для загрузки ресурсов и данных в приложение с последующей их обработкой. Эти механизмы интуитивно понятны и согласованы с различными прикладными программными интерфейсами. Класс Loader предлагает единый механизм для загрузки SWF-файлов и графических ресурсов, а также обеспечивает способ доступа к детальной информации о загруженном содержимом. Класс URLLoader предлагает отдельный механизм для загрузки текстовых и двоичных данных в управляемые данными приложения. Класс Socket предлагает средства для считывания и записи двоичных данных в серверные сокеты в любом формате.

Доступ к данным на нижнем уровне

В различных прикладных программных интерфейсах обеспечивается доступ к данным низкого уровня. Для загружаемых данных класс URLStream обеспечивает доступ к несжатым двоичным данным непосредственно в процессе загрузки. Класс ByteArray позволяет оптимизировать чтение, запись и работу с двоичными данными. Прикладной интерфейс программирования звука обеспечивает точное управление звуком с помощью классов SoundChannel и SoundMixer. Прикладные программные интерфейсы взаимодействуют с системами безопасности, предоставляя информацию о соответствующих привилегиях SWF-файла или загруженного содержимого, позволяя обрабатывать ошибки в системе безопасности.

Работа с текстом

В ActionScript 3.0 входит пакет flash.text, содержащий все связанные с обработкой текста прикладные программные интерфейсы. Класс TextLineMetrics обеспечивает подробные метрики для строк текста в текстовом поле; он заменил метод `TextFormat.getTextExtent()`, использовавшийся в ActionScript 2.0. Класс TextField содержит ряд методов низкого уровня, предоставляющих определенную информацию о строке текста или об отдельном символе в текстовом поле. Например, метод `getCharBoundaries()` возвращает прямоугольник в виде ограничивающего прямоугольника символа. Метод `getCharIndexAtPoint()` возвращает указатель символа в заданной точке. Метод `getFirstCharInParagraph()` возвращает указатель первого символа в абзаце. К методам работы на уровне строк относятся метод `getLineLength()`, возвращающий число символов в указанной строке текста, и метод `getLineText()`, который возвращает текст из указанной строки. Класс Font обеспечивает средства для управления встроенными шрифтами в SWF-файлах.

Для управления текстом на более низком уровне классы в пакете flash.text.engine формируют механизм визуализации текста Flash. Этот набор классов обеспечивает управление текстом на низком уровне и предназначен для создания текстовых сред и компонентов.

Глава 2. Начало работы с языком ActionScript

Основы программирования

Поскольку ActionScript является языком программирования, для его изучения необходимо понимание ряда основных концепций, используемых при создании компьютерных программ.

Какие функции выполняют компьютерные программы

Прежде всего необходимо понять, что представляют собой компьютерные программы и какие функции они выполняют. Существует два основных свойства компьютерной программы.

- Программа представляет собой серию инструкций или шагов, которые должен выполнить компьютер.
- Каждый шаг в конечном счете приводит к управлению определенной порцией информации или данных.

В общем смысле, компьютерная программа — это пошаговые инструкции, которые задаются компьютеру оператором и которые компьютер выполняет одну за другой. Каждое отдельное указание называется *инструкцией*. В среде ActionScript после каждого оператора следует точка с запятой.

В сущности, все функции отдельной инструкции программы сводятся к управлению несколькими битами информации, хранящимися в памяти компьютера. Простым примером является сложение двух чисел и сохранение результата в памяти компьютера. Более сложным примером является рисование прямоугольника на экране компьютера и создание программы перемещения этого прямоугольника в другое место экрана. Определенная информация о прямоугольнике сохраняется в памяти компьютера: координаты x, y расположения прямоугольника, ширина и длина сторон, цвет и так далее. Каждый бит этой информации хранится в памяти компьютера. Алгоритм программы перемещения прямоугольника в другое место на экране включает такие действия, как присвоение координате x значения 200, а координате y — значения 150. Другими словами, необходимо присвоить новые значения координатам x и y. Компьютер выполняет некоторые скрытые от пользователя операции с этими данными для фактического преобразования этих чисел в изображение, показанное на экране компьютера. Однако на базовом уровне достаточно знать то, что процесс перемещения прямоугольника на экране включает только изменение битов данных в памяти компьютера.

Переменные и постоянные

В основном процесс программирования включает изменение только части данных в памяти компьютера. Следовательно, важно иметь способ представления части данных в программе. *Переменная* является именем, которое представляет значение в памяти компьютера. При записи операторов для управления значениями имя переменной записывается вместо значения. Когда компьютер обнаруживает имя переменной в программе, он обращается к своей памяти и использует то значение, которое там находится. Например, если есть две переменные с именами `value1` (значение 1) и `value2` (значение 2), каждая из которых содержит число, тогда для сложения этих чисел можно написать следующую инструкцию:

```
value1 + value2
```

При выполнении этих шагов на практике компьютер находит значения для каждой переменной и складывает их.

В ActionScript 3.0 переменная фактически состоит из трех различных частей:

- имени переменной
- типа данных, которые могут быть сохранены в переменной
- фактического значения, сохраненного в памяти компьютера

Было рассмотрено, как компьютер использует имя в качестве местозаполнителя значения. Столь же важен и тип данных. При создании переменной в среде ActionScript задается определенный тип данных, предназначенный для хранения. С этого момента при выполнении программных команд в переменной могут сохраняться значения только этого типа данных. Можно управлять значением с использованием определенных характеристик, связанных с этим типом данных. Для создания переменной в ActionScript (этот процесс называется также объявлением или *заданием* переменной) используется инструкция `var`:

```
var value1:Number;
```

В этом примере определены команды создания переменной `value1`, в которой могут храниться только значения с типом Number. (Number — это определенный тип данных в среде ActionScript.) Можно также сохранить значение прямо в переменной:

```
var value2:Number = 17;
```

Adobe Flash Professional

Во Flash Professional есть другой способ задания переменной. При размещении символа фрагмента ролика, символа кнопки или текстового поля в рабочей области можно задать для них имя экземпляра в инспекторе свойств. В скрытом для пользователя режиме приложение Flash Professional создает переменную с именем, которое совпадает с именем экземпляра. Это имя можно использовать в коде ActionScript для представления этого элемента в рабочей области. Предположим, к примеру, что в рабочей области существует символ фрагмента ролика и вы присваиваете ему имя экземпляра `rocketShip`. При использовании переменной `rocketShip` в коде ActionScript вы фактически управляете фрагментом ролика.

Константа аналогична переменной. Это имя, которое представляет значение с заданным типом данных в компьютерной памяти. Разница состоит в том, что для постоянной значение присваивается только один раз во время выполнения приложения ActionScript. Если постоянной присвоено значение, оно больше не изменяется в приложении. Синтаксис для определения константы почти совпадает с синтаксисом для определения переменной. Единственным отличием является то, что ключевое слово `const` используется вместо ключевого слова `var`:

```
const SALES_TAX_RATE:Number = 0.07;
```

Константа используется для определения значения, которое задействовано на многих стадиях проекта и которое не изменяется при нормальных обстоятельствах. Использование постоянных вместо литеральных значений делает код более удобочитаемым. К примеру, рассмотрим две версии одного кода. Один код используется для умножения цены на значение переменной `SALES_TAX_RATE`. Другой код используется для умножения цены на `0,07`. Версия, в которой используется константа `SALES_TAX_RATE`, более проста в понимании. Кроме того, предположим, что значение, заданное константой, изменяется. Если для представления значения в проекте используется константа, можно изменить значение в одном месте (объявление константы). Напротив, потребуется изменить его в различных местах при использовании запрограммированных значений литералов.

Типы данных

В ActionScript существует много типов данных, которые можно использовать при создании переменных. Некоторые из этих типов данных считаются простыми или фундаментальными типами данных:

- Строки: текстовые значения, такие как имя или текст главы книги
- Числовые: ActionScript 3.0 включает три специфических типа числовых данных:
  - Number: любые целые или дробные числовые значения
  - int: целые числа без дробей
  - uint: беззнаковые (неотрицательные) целые числа
- Логические: значения типа «истинно — ложно», такие как состояния выключателя или равенство или неравенство двух значений

Простые типы данных представляют единичные порции информации: например, одно число или единичный текст. Однако большинство типов данных, определенных в среде ActionScript, являются комплексными типами данных. Они представляют набор значений в одном контейнере. Например, переменная с типом данных Date (Дата) представляет одно значение (момент времени). Тем не менее, значение даты включает несколько значений: день, месяц, год, часы, минуты, секунды и т. д., все из которых являются отдельными числами. Обычно дата считается единичным значением, и с ней можно работать как с единичным значением путем создания переменной Date. Однако в компьютере дата рассматривается как группа из нескольких значений, которые совместно определяют одну дату.

Большинство встроенных типов данных, так же как и большинство типов данных, задаваемых программистами, являются комплексными. Некоторые из комплексных типов данных, возможно, уже вам знакомы:

- MovieClip: символ фрагмента ролика
- TextField: динамическое или вводимое текстовое поле
- SimpleButton: символ кнопки
- Date: информация о единичном моменте времени (дата и время)

Двумя широко распространенными синонимами для обозначения типа данных являются класс и объект. *Класс* является просто определением типа данных. Он равнозначен шаблону всех объектов в типе данных, как в выражении «все переменные типа данных "Пример" имеют характеристики А, Б и В». *Объект*, с другой стороны, является всего лишь фактическим экземпляром класса. Например, переменную с типом данных MovieClip можно описать как объект MovieClip. Ниже одна и та же мысль выражена различными словами.

- Типом данных переменной `myVariable` является Number (число).
- Переменная `myVariable` является экземпляром Number.
- Переменная `myVariable` является объектом Number.
- Переменная `myVariable` является экземпляром класса Number.

Работа с объектами

ActionScript известен как язык программирования, ориентированный на объекты. Объектно-ориентированное программирование является подходом к программированию. Это не более чем способ организации кода в программе с использованием объектов.

Ранее термин «компьютерная программа» обозначал последовательность действий или команд, выполняемых компьютером. Теоретически можно представить компьютерную программу как один длинный список команд. Однако в объектно-ориентированном программировании программные команды распределены между различными объектами. Код группируется в наборы функций, поэтому связанные типы функций и связанные части данных группируются в один контейнер.

Adobe Flash Professional

Те, кому приходилось работать с символами в приложении Flash Professional, фактически уже знают, как работать с объектами. Представим, что вы определили символ фрагмента ролика, например рисунок прямоугольника, и поместили копию в рабочую область. Этот символ фрагмента ролика является также (буквально) объектом ActionScript или экземпляром класса MovieClip.

Существует несколько характеристик фрагмента ролика, которые можно изменить. Если объект выбран, в Инспекторе свойств можно изменять значения, такие как значение координаты x или ширины объекта. Кроме того, можно выполнять различные виды коррекции цветов, например изменение значения альфа (прозрачности) или применение фильтра тени. Другие инструменты Flash Professional позволяют осуществлять дополнительные изменения; например, инструмент «Свободное преобразование» выполняет поворот прямоугольника. Все эти способы изменения символа фрагмента ролика во Flash Professional также доступны в среде ActionScript. В среде ActionScript фрагмент ролика можно изменить путем добавления частей данных, формирующих один набор, называемый объектом MovieClip.

В ориентированном на объекты языке программирования ActionScript существует три типа характеристик, которые могут входить в любой класс:

- Свойства
- Методы
- События

Эти элементы используются для управления частями данных, которые задействованы в программе, и для принятия решения о том, какие действия должны быть выполнены и в каком порядке.

Свойства

Свойство представляет собой одну из порций данных, упакованную в объект. Если рассмотреть в качестве примера музыкальную композицию, ее свойствами могут быть `artist` (имя исполнителя) и `title` (название). Класс MovieClip имеет такие свойства, как `rotation` (поворот), `x`, `width` (ширина) и `alpha` (прозрачность). Работа со свойствами аналогична работе с отдельными переменными. Фактически свойства можно рассматривать как «дочерние» переменные, содержащиеся в объекте.

Ниже приведены примеры кодов ActionScript, в которых используются свойства. Эта строка кода перемещает MovieClip с именем `square` (квадрат) по оси X на 100 пикселов.

```
square.x = 100;
```

Этот код использует свойство rotation для поворота MovieClip `square` в соответствии с поворотом MovieClip `triangle` (треугольник):

```
square.rotation = triangle.rotation;
```

Этот код изменяет горизонтальный масштаб объекта MovieClip `square` таким образом, что его ширина увеличивается в полтора раза:

```
square.scaleX = 1.5;
```

Обратите внимание на общую структуру: используется переменная (`square`, `triangle`) в качестве имени объекта, после нее указывается точка (.), а затем имя свойства (`x`, `rotation`, `scaleX`). Точка, известная как *оператор точки*, используется для указания на доступ к одному из дочерних элементов объекта. Полная структура: «имя переменной — точка — имя свойства» используется в качестве одной переменной, как имя единичного значения в памяти компьютера.

Методы

Метод — это действие, которое может выполнять объект. Например, предположим, что создан символ фрагмента ролика Flash Professional с несколькими ключевыми кадрами и анимацией на временной шкале. Можно создать команды воспроизведения или остановки этого фрагмента ролика, а также перемещения к определенному кадру.

Этот код дает команду фрагменту ролика с именем `shortFilm` начать воспроизведение:

```
shortFilm.play();
```

Эта строка останавливает воспроизведение фрагмента ролика с именем `shortFilm` (воспроизводящая головка останавливается, аналогично режиму паузы при воспроизведении видеофильма):

```
shortFilm.stop();
```

Этот код дает команду для фрагмента ролика с именем `shortFilm` переместить воспроизводящую головку на кадр 1 и остановить воспроизведение (аналогично обратной перемотке видеофильма):

```
shortFilm.gotoAndStop(1);
```

Доступ к методам, так же как и к свойствам, осуществляется путем записи имени объекта (переменной), за которым следуют точка, имя метода и скобки. Скобки указывают на *вызов* метода, то есть дают команду объекту для выполнения действия. Иногда в скобках располагаются значения (или переменные) для передачи дополнительной информации, которая необходима для выполнения действия. Эти значения называются *параметрами* метода. Например, для метода `gotoAndStop()` требуется информация о том, к какому кадру следует перейти, поэтому в скобках необходимо указать один параметр. Другие методы, такие как `play()` и `stop()`, являются очевидными и не требуют дополнительной информации. Тем не менее, они также пишутся со скобками.

В отличие от свойств (и переменных), методы не используются в качестве местозаполнителя значения. Тем не менее, некоторые методы могут выполнять вычисления и возвращать результат, который можно использовать в качестве переменной. Например, метод числового класса `toString()` конвертирует числовое значение в его текстовое представление:

```
var numericData:Number = 9;
var textData:String = numericData.toString();
```

Например, можно использовать метод `toString()` для отображения значения числовой переменной в текстовом поле на экране. Свойство `text` класса TextField определяется в качестве переменной String (строка), поэтому может содержать только текстовые значения. (Свойство `text` представляет фактическое текстовое содержимое, отображаемое на экране.) Эта строка кода преобразует числовое значение переменной `numericData` в текст. Затем значение отображается на экране в объекте TextField с именем `calculatorDisplay`:

```
calculatorDisplay.text = numericData.toString();
```

События

Компьютерная программа — это последовательность команд, которые пошагово выполняет компьютер. Некоторые простые компьютерные программы не содержат ничего, кроме нескольких шагов, после выполнения которых компьютером программа заканчивается. Тем не менее, программы ActionScript разработаны таким образом, что они продолжают работать, ожидая введения данных пользователем или каких-либо других событий. События являются механизмом, определяющим, когда и какие инструкции должен выполнять компьютер.

В сущности, *события* — это все то, что происходит, о чем ActionScript «знает» и на что может ответить. Многие события связаны с взаимодействием пользователя, например нажатие программной кнопки или клавиши клавиатуры. Также существуют другие виды событий. Например, если ActionScript используется для загрузки внешнего изображения, существует событие, которое позволит пользователю узнать о завершении загрузки. Запущенная программа ActionScript фактически ожидает появления событий. При появлении этих событий выполняется определенный код ActionScript, заданный для обработки этих событий.

Основные сведения об обработке событий

Методика указания определенных действий, которые выполняются при возникновении событий, называется *обработкой событий*. При написании кода ActionScript для обработки события необходимо определить три важных элемента.

- Источник события: в каком объекте должно произойти событие? Например, какая кнопка была нажата или какой объект Loader загружает изображение? Источник событий также называется *целью событий*. Он получил такое название, поскольку представляет собой объект, в котором компьютер проверяет наличие событий (то есть место, в котором фактически происходит событие).
- Событие: что именно должно произойти и на что именно требуется ответить? Идентификация определенных событий играет важную роль, поскольку многие объекты инициируют несколько событий.
- Ответ: какие шаги необходимо выполнить, когда событие произойдет?

При написании кода ActionScript для обработки событий необходимы следующие три элемента. Код имеет следующую базовую структуру (элементы, выделенные жирным шрифтом, являются местозаполнителями, которые подставляются в зависимости от конкретного случая).

```
function eventResponse(eventObject:EventType):void
{
    // Actions performed in response to the event go here.
}

eventSource.addEventListener(EventType.EVENT_NAME, eventResponse);
```

Такой код выполняет две задачи. Во-первых, определяет функцию, то есть способ задания действий, которые требуется выполнить в ответ на событие. Затем он вызывает метод `addEventListener()` исходного объекта. Вызов метода `addEventListener()` в сущности «присоединяет» функцию к указанному событию. При возникновении событий выполняются действия функции. Рассмотрим эти части более подробно.

Раздел *function* обеспечивает способ группировки действий под одним именем — аналогично созданию имени быстрого вызова для выполнения действий. Функция равнозначна методу — за исключением того, что ее не обязательно связывать с определенным классом. (Фактически термину «метод» можно дать следующее определение: это функция, которая связана с определенным классом.) При создании функции для обработки события необходимо выбрать для нее имя (в данном случае именем функции будет `eventResponse`). Также задается один параметр (с именем `eventObject` в этом примере). Указание параметра функции аналогично заданию переменной, поэтому требуется указать также и тип данных параметра. (В этом примере тип данных параметра: `EventType`).

Каждый тип событий, которые требуется отслеживать, имеет соответствующий класс в ActionScript. Тип данных, указанный для параметра функции, всегда связан с классом определенного события, на которое требуется ответить. Например, событие `click` (инициируемое, когда пользователь щелкает элемент кнопкой мыши) связано с классом MouseEvent. Для записи функции listener (прослушиватель) для события `click` задается параметр этой функции с типом данных MouseEvent. И, наконец, в фигурных скобках ({ ... }) записываются команды, которые должен выполнить компьютер при возникновении события.

Функция обработки событий записана. Затем объекту-источнику событий (объект, в котором возникает событие, например кнопка) назначаются команды вызова функции при возникновении события. Функция регистрируется в объекте источника событий за счет вызова метода `addEventListener()` этого объекта (все объекты, в которых возникают события, также имеют метод `addEventListener()`). В методе `addEventListener()` используются два параметра.

- Во-первых, имя определенного события, на которое требуется ответить. Каждое событие связано с определенным классом. Каждый класс событий имеет специальное значение, которое аналогично уникальному имени, заданному для каждого из этих событий. Это значение используется для первого параметра.
- Во-вторых, имя функции ответа на событие. Обратите внимание, что имя функции пишется без скобок, когда оно передается как параметр.

Процесс обработки событий

Ниже приводится пошаговое описание процесса, возникающего при создании прослушивателя события (event listener). Это пример создания функции listener, которая вызывается при щелчке мыши по объекту с именем `myButton`.

Фактический код, написанный программистом, таков:

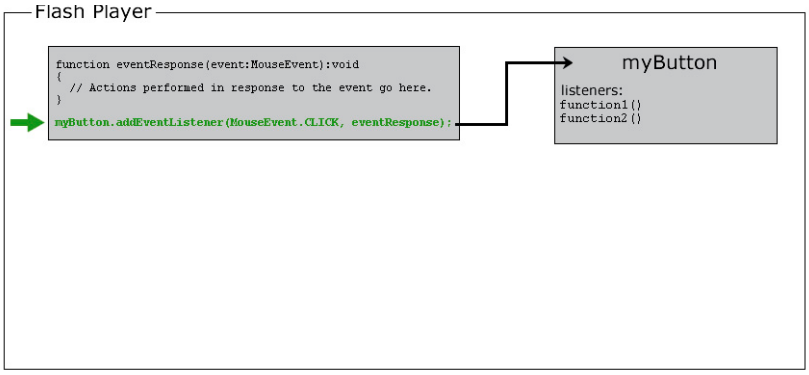
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

myButton.addEventListener(MouseEvent.CLICK, eventResponse);
```

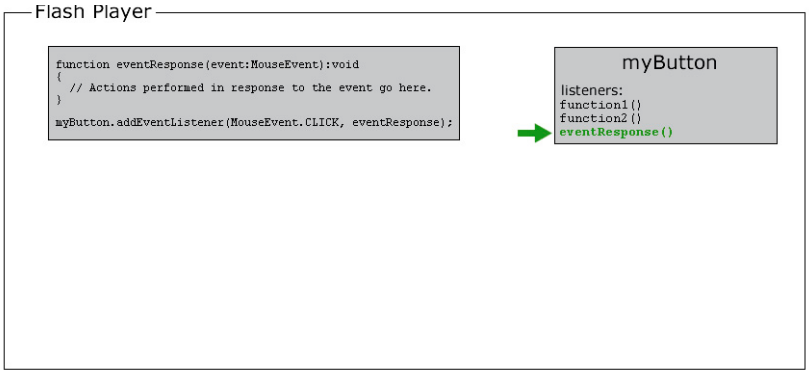
Здесь рассматривается, как этот код будет фактически выполняться при запуске.

1 При загрузке SWF-файла компьютер регистрирует факт наличия функции с именем `eventResponse()`.

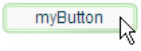
2 Затем компьютер выполняет код (в частности, те строки кода, которых нет в функции). В данном случае это только одна строка кода: вызов метода `addEventListener()` объекта-источника события (с именем `myButton`) и передача функции `eventResponse` как параметра.



Объект `myButton` содержит список функций, перечисленных для каждого возникающего в нем события. При вызове метода `addEventListener()` объект `myButton` сохраняет функцию `eventResponse()` в списке прослушивателей событий.

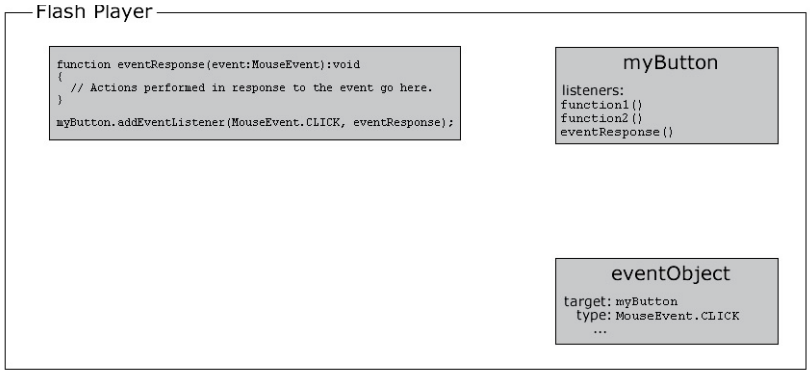


3 В какой-то момент пользователь, щелкая кнопкой мыши по объекту `myButton`, запускает его событие `click` (определенное в коде как `MouseEvent.CLICK`).

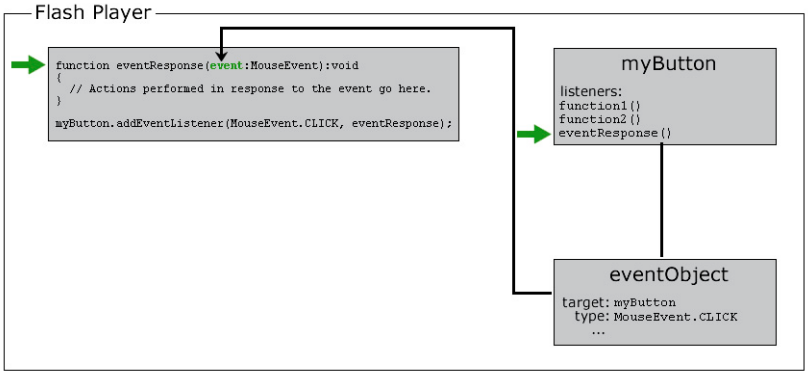


В этот момент происходит следующее:

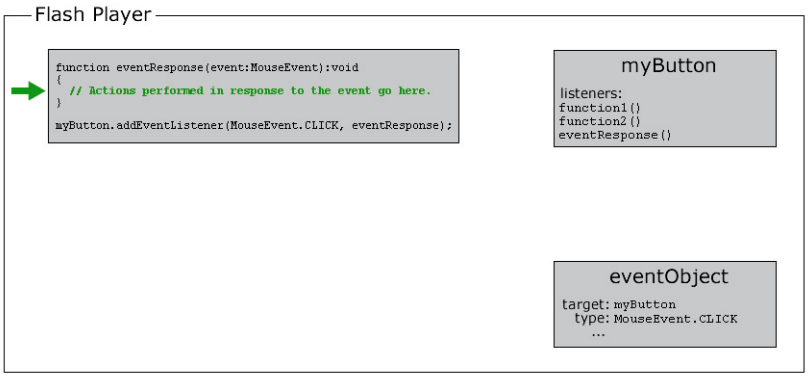
a Создается объект, который является экземпляром класса, связанного с рассматриваемым событием (в данном случае с событием MouseEvent). Для многих событий этот объект является экземпляром класса Event. Для событий мыши это экземпляр MouseEvent. Для остальных событий это экземпляр класса, связанного с этим событием. Этот созданный объект называется *event object*. Он содержит специфическую информацию о происшедшем событии: тип, где оно произошло и другие данные, если они применимы.



b Затем компьютер обращается к списку прослушивателей событий, который хранится в объекте `myButton`. Он перебирает эти функции одну за другой, вызывает их и передает объект события в функцию в качестве параметра. Поскольку функция `eventResponse()` является одним из прослушивателей объекта `myButton`, в рамках этого процесса компьютер вызывает функцию `eventResponse()`.



c При вызове функции `eventResponse()` начинает выполняться ее код, осуществляя тем самым действия, заданные пользователем.



Примеры обработки событий

Ниже приводятся несколько конкретных примеров кода обработки событий. Эти примеры позволяют получить представление о некоторых общих элементах событий и возможных вариантах написания кода обработки событий.

- Нажатие кнопки для запуска воспроизведения текущего фрагмента ролика. В следующем примере именем экземпляра кнопки является `playButton`; `this` — специальное имя, означающее «текущий объект»:

```
this.stop();

function playMovie(event:MouseEvent):void
{
    this.play();
}

playButton.addEventListener(MouseEvent.CLICK, playMovie);
```

- Обнаружение текста в текстовое поле. В этом примере `entryText` является входным текстовым полем, а `outputText` — динамическим текстовым полем:

```
function updateOutput(event:TextEvent):void
{
    var pressedKey:String = event.text;
    outputText.text = "You typed: " + pressedKey;
}

entryText.addEventListener(TextEvent.TEXT_INPUT, updateOutput);
```

- Нажатие кнопки для перехода по адресу URL: В этом случае `linkButton` является именем экземпляра кнопки:

```
function gotoAdobeSite(event:MouseEvent):void
{
    var adobeURL:URLRequest = new URLRequest("http://www.adobe.com/");
    navigateToURL(adobeURL);
}

linkButton.addEventListener(MouseEvent.CLICK, gotoAdobeSite);
```

Создание экземпляров объекта

Перед использованием объекта в ActionScript необходимо, чтобы этот объект существовал. Одним из этапов создания объекта является задание переменной, однако эта операция создает только свободную ячейку в памяти компьютера. Всегда назначайте фактическое значение переменной (то есть создавайте объект и сохраняйте его в переменной), перед тем как использовать ее или управлять ею. Процесс создания объекта называется *созданием экземпляра* объекта. Другими словами, создается экземпляр определенного класса.

Один простой способ создания экземпляра объекта вообще не требует использования ActionScript. Во Flash Professional поместите символ фрагмента ролика, символ кнопки или текстовое поле в рабочую область и назначьте элементу имя экземпляра. В приложении Flash Professional автоматически объявляется переменная с этим именем экземпляра, создается экземпляр объекта и этот объект сохраняется в переменной. Аналогично в приложении Flex пользователь создает компонент в MXML путем создания кода тега MXML или размещения компонента в редакторе в режиме оформления Flash Builder. При назначении идентификатора этому компоненту данный идентификатор становится именем переменной ActionScript, содержащей экземпляр компонента.

Однако не всегда существует необходимость визуального создания объектов, а создание невизуальных объектов невозможно. Существует несколько дополнительных способов создания экземпляров объектов с использованием только ActionScript.

С помощью нескольких типов данных ActionScript можно создавать экземпляры, используя *литеральное выражение*, которое представляет собой значение, записываемое непосредственно в код ActionScript. Примеры:

- Литеральное числовое значение (непосредственный ввод числа):

```
var someNumber:Number = 17.239;
var someNegativeInteger:int = -53;
var someUint:uint = 22;
```

- Литеральное строчное значение (текст, заключенный в двойные кавычки):

```
var firstName:String = "George";
var soliloquy:String = "To be or not to be, that is the question...";
```

- Литеральное логическое значение (используются значения истинно/ложно — `true` или `false`):

```
var niceWeather:Boolean = true;
var playingOutside:Boolean = false;
```

- Литеральный массив значений (список разделенных запятой значений в квадратных скобках):

```
var seasons:Array = ["spring", "summer", "autumn", "winter"];
```

- Литеральное значение XML (непосредственный ввод XML):

```
var employee:XML = <employee>
        <firstName>Harold</firstName>
        <lastName>Webster</lastName>
    </employee>;
```

В ActionScript определяются также литеральные выражения для следующих типов данных: Array, RegExp, Object и Function.

Наиболее распространенным способом создания экземпляра для любого типа данных является использование оператора `new` с именем класса, как показано ниже.

```
var raceCar:MovieClip = new MovieClip();
var birthday:Date = new Date(2006, 7, 9);
```

Создание объекта с помощью оператора `new` часто называется «вызовом конструктора класса». *Конструктор* — это специальный метод, который является частью процесса создания экземпляра класса. Обратите внимание, что при создании экземпляра таким способом после имени класса указываются скобки. Иногда в скобках указываются значения параметров. Эти два действия также выполняются при вызове метода.

Для тех типов данных, которые позволяют создавать экземпляры с помощью литерального выражения, все равно можно создавать экземпляр объекта, используя оператор `new`. Например, эти две строки кода дают одинаковый результат:

```
var someNumber:Number = 6.33;
var someNumber:Number = new Number(6.33);
```

Очень важно приобрести опыт в создании объектов с помощью метода `new` *ClassName*`()`. Многие типы данных ActionScript не имеют визуального представления. Поэтому их нельзя создать, поместив элемент в рабочую область Flash Professional или режим оформления редактора MXML приложения Flash Builder. В среде ActionScript с помощью оператора `new` можно создать только экземпляр любого из этих типов данных.

Adobe Flash Professional
В приложении Flash Professional оператор `new` можно также использовать для создания экземпляра символа фрагмента ролика, который определен в библиотеке, но не помещен в рабочую область.

Дополнительные разделы справки

Работа с массивами

Использование регулярных выражений

«Тип данных Object» на странице 61

Создание объектов MovieClip с помощью ActionScript

Общие элементы программы

Существует несколько дополнительных стандартных блоков, используемых для создания программы ActionScript.

Операторы

Операторы — это специальные символы (иногда слова), которые используются для выполнения вычислений. В основном они используются для математических операций, а также для сравнения значений друг с другом. Как правило, оператор использует одно или несколько значений и возвращает единичный результат. Например:

- Оператор сложения (`+`) складывает два значения, результатом чего является одно число:

  ```
  var sum:Number = 23 + 32;
  ```

- Оператор умножения (`*`) перемножает два значения, результатом чего также является одно число:

  ```
  var energy:Number = mass * speedOfLight * speedOfLight;
  ```

- Оператор равенства (`==`) сравнивает два значения и выдает результат в форме единичного (логического) значения истинно/ложно:

```
if (dayOfWeek == "Wednesday")
{
    takeOutTrash();
}
```

Как здесь показано, оператор равенства и другие операторы сравнения наиболее часто используются с оператором `if` для определения необходимости выполнения определенных команд.

Дополнительные разделы справки

«Операторы» на странице 71

Комментарии

Во время создания программы ActionScript часто требуется оставить замечания для себя. Например, иногда необходимо пояснить принцип выполнения некоторых строк кода или причину выбора определенного элемента. *Комментарии кода* представляют собой инструмент, используемый для записи текста, который игнорируется компьютером при выполнении кода. В ActionScript предусмотрено два вида комментариев.

- Однострочный комментарий: вставляется в любое место строки и отмечается двумя косыми чертам. Компьютер полностью игнорирует строку, в начале которой указана косая черта:

  ```
  // This is a comment; it's ignored by the computer.
  var age:Number = 10; // Set the age to 10 by default.
  ```

- Многострочные комментарии: состоят из начального маркера `/*`, содержания комментария и конечного маркера `*/`. Компьютер игнорирует все содержимое между начальным и конечным маркерами вне зависимости от количества строк, содержащихся в комментарии:

  ```
  /*
  This is a long description explaining what a particular
  function is used for or explaining a section of code.

  In any case, the computer ignores these lines.
  */
  ```

Другим распространенным случаем использования комментариев является временное «отключение» одной или нескольких строк кода. Например, комментарии можно использовать при тестировании различных способов выполнения какого-либо действия. Кроме того, их можно использовать для выявления причины неправильной работы определенного кода ActionScript.

Управление исполнением программы

Довольно часто в программе требуется повторить отдельные действия, выполнить только выбранные действия и не выполнять другие, переключиться на альтернативные действия в зависимости от определенных условий и т. п. *Управление исполнением программы* предназначено для контроля выполняемых действий. В ActionScript предусмотрено несколько элементов управления исполнением программы.

- Функции: своего рода комбинации быстрого вызова. Они обеспечивают способ группировки последовательности действий под одним именем и могут использоваться для выполнения вычислений. Функции необходимы для обработки событий, они также используются как основной инструмент группировки последовательности команд.

- Циклы: циклические структуры, позволяющие задать набор команд, которые компьютер выполняет установленное число раз или до изменения некоторых условий. Часто циклы используются для управления несколькими связанными элементами. При этом используется переменная, значение которой изменяется каждый раз, когда компьютер выполнит один цикл.
- Условные операторы: обеспечивают способ назначения команд, выполняемых только в определенных случаях. Они также используются для предоставления альтернативного набора команд при выполнении различных условий. Наиболее распространенной условной инструкцией является инструкция `if`. Инструкция `if` проверяет значение или выражение в скобках. Если значением является `true`, выполняются строки кода, указанные в фигурных скобках. В противном случае они игнорируются. Например:

```
if (age < 20)
{
    // show special teenager-targeted content
}
```

Сравнение с помощью оператора `if` и оператора `else` позволяет задать альтернативные команды, выполняемые компьютером, если условие не принимает значение `true`:

```
if (username == "admin")
{
    // do some administrator-only things, like showing extra options
}
else
{
    // do some non-administrator things
}
```

Дополнительные разделы справки

«Функции» на странице 82

«Повтор» на странице 79

«Условия» на странице 77

Пример. Фрагмент анимации портфолио (Flash Professional)

Этот пример необходим для того, чтобы дать первую возможность программисту собрать отдельные блоки кода ActionScript в полную программу. Фрагмент анимации портфолио является примером использования существующей линейной анимации и добавления нескольких второстепенных интерактивных элементов. Например, можно включить анимацию, созданную для клиента, в онлайн-портфолио. Интерактивные элементы, добавленные к анимации, включают две кнопки, которые может нажимать зритель: одна для запуска анимации, другая для перехода к отдельному URL (к меню портфолио или на главную страницу автора).

Процесс создания этого фрагмента будет разделен на следующие основные части:

1. Подготовка FLA-файла для добавления ActionScript и интерактивных элементов.
2. Создание и добавления кнопок.
3. Написание кода ActionScript.

4 Проверка приложения.

Подготовка и добавление интерактивных элементов

Перед добавлением интерактивных элементов к анимации следует подготовить FLA-файл, создав место для добавления нового содержимого. Эта операция включает создание пространства в рабочей области, в которое будут добавлены кнопки. Она также включает создание «пространства» во FLA-файле для раздельного хранения различных элементов.

Подготовка FLA-файла к добавлению интерактивных элементов

1 Создайте FLA-файл с простой анимацией, например одной анимацией движения или формы. Если FLA-файл, содержащий анимацию, демонстрируемую в проекте, уже существует, откройте этот файл и сохраните его с новым именем.

2 Выберите место на экране для отображения двух кнопок. Одна кнопка необходима для запуска анимации, а другая кнопка является ссылкой на портфолио или главную страницу автора. При необходимости очистите или добавьте место в рабочей области для нового содержания. Если место уже доступно, можно создать экран запуска в первом кадре. В этом случае, возможно, потребуется изменить анимацию, чтобы она запускалась со второго или последующих кадров.

3 Добавьте новый слой поверх других слоев на временной шкале и присвойте ему имя **кнопки**. На этот слой будут добавлены кнопки.

4 Добавьте новый слой поверх слоя кнопок и присвойте ему имя **действия**. На этот слой будет добавлен код ActionScript для этого приложения.

Создание и добавление кнопок

Далее создаются и размещаются кнопки, формирующие центр интерактивного приложения.

Создание и добавление кнопок к FLA-файлу

1 Используя инструменты рисования, создайте в слое кнопок визуальный образ первой кнопки («воспроизведение»). Например, нарисуйте горизонтальный овал с текстом поверх него.

2 Используя инструмент выделения, выделите все графические части кнопки.

3 В главном меню выберите команды «Модификация» > «Преобразовать в символ».

4 Выберите в диалоговом окне в качестве типа символа «Кнопка», присвойте символу имя и нажмите «ОК».

5 При выделенной кнопке в инспекторе свойств присвойте ей имя экземпляра **playButton**.

6 Повторите шаги с 1 по 5 для создания кнопки, с помощью которой зритель сможет перейти к главной странице автора. Имя этой кнопки: **homeButton**.

Написание кода

Код ActionScript для этого приложения можно разделить на три функциональных блока, несмотря на то что он вводится в одном месте. Код выполняет следующие три функции:

- Остановить воспроизводящую головку сразу после загрузки SWF-файла (в тот момент, когда воспроизводящая головка подойдет к Кадру 1).
- Следить за событием, чтобы запустить воспроизведение SWF-файла при нажатии зрителем кнопки воспроизведения.

- Следить за событием, чтобы отправить обозреватель на соответствующий URL-адрес при нажатии зрителем кнопки главной страницы.

Создание кода остановки воспроизводящей головки в начале Кадра 1

1. Выберите ключевой кадр для Кадра 1 в слое действий.
2. Чтобы открыть панель «Действия», выберите в главном меню команды «Окно» > «Действия».
3. На панели «Сценарий» введите следующий код:

```
stop();
```

Запись кода запуска анимации при нажатии кнопки воспроизведения

1. Добавьте две пустые строки в конце кода предыдущего шага.
2. Введите следующий код в нижней части сценария:

```
function startMovie(event:MouseEvent):void
{
    this.play();
}
```

   Код определяет функцию с именем `startMovie()`. При вызове функция `startMovie()` запускает временную шкалу и начинает воспроизведение.

3. На строке, следующей за кодом, добавленным на предыдущем шаге, введите следующую строку кода:

```
playButton.addEventListener(MouseEvent.CLICK, startMovie);
```

   Эта строка кода регистрирует функцию `startMovie()` как прослушивателя события `click` для кнопки `playButton`. Другими словами, код определяет, что при нажатии кнопки с именем `playButton` вызывается функция `startMovie()`.

Запись кода отправки обозревателя на URL-адрес при нажатии кнопки главной страницы

1. Добавьте две пустые строки в конце кода предыдущего шага.
2. Введите следующий код в нижней части сценария:

```
function gotoAuthorPage(event:MouseEvent):void
{
    var targetURL:URLRequest = new URLRequest("http://example.com/");
    navigateToURL(targetURL);
}
```

   Код определяет функцию с именем `gotoAuthorPage()`. Эта функция сначала создает экземпляр URLRequest, представляющий URL-адрес http://example.com/. Затем она передает этот URL в функцию `navigateToURL()`, благодаря чему в браузере пользователя открывается страница с этим URL-адресом.

3. На строке, следующей за кодом, добавленным на предыдущем шаге, введите следующую строку кода:

```
homeButton.addEventListener(MouseEvent.CLICK, gotoAuthorPage);
```

   Эта строка кода регистрирует функцию `gotoAuthorPage()` как прослушивателя события `click` для кнопки `homeButton`. Другими словами, код определяет, что при нажатии кнопки с именем `homeButton` вызывается функция `gotoAuthorPage()`.

Проверка приложения

Теперь приложение является полностью функциональным. Однако чтобы убедиться в этом, его следует проверить.

Выполнение проверки приложения

1. В главном меню выберите команды «Управление» > «Тестировать ролик». Flash Professional создает SWF-файл и открывает его в окне проигрывателя Flash Player.
2. Нажмите обе кнопки и убедитесь, что они выполняют требуемые действия.
3. Если кнопки не работают, необходимо выяснить следующее:
   - Имеют ли кнопки отдельные имена экземпляров?
   - Совпадают ли имена, используемые ли при вызове метода `addEventListener()`, с именами экземпляров кнопок?
   - Используются ли корректные имена событий при вызове метода `addEventListener()`?
   - Указаны ли корректные параметры для каждой из функций? (Оба метода должны иметь один параметр с типом данных MouseEvent.)

   При возникновении всех этих ошибок, а также большинства других возможных ошибок появляется сообщение об ошибке. Сообщение об ошибке может появиться при выборе команды «Тестировать ролик» или при нажатии кнопки во время тестирования проекта. Ошибки компиляции показаны на панели «Ошибки компиляции» (это ошибки, которые возникают при первом выборе команды «Тестировать ролик»). На панели «Вывод» показаны ошибки выполнения, которые возникают при воспроизведении содержимого, например при нажатии кнопки.

Создание приложений с ActionScript

Процесс создания приложений с помощью языка ActionScript требует намного большего, чем простое знание синтаксиса и названий используемых классов. В большинстве случаев в документации по платформе Flash Platform содержатся эти два раздела (разделы о синтаксисе и об использовании классов ActionScript). Однако для создания приложения ActionScript также потребуется следующая информация.

- Информация о программах, используемых для написания кода ActionScript.
- Информация об организации кода ActionScript.
- Информация о включении кода ActionScript в приложение.
- Информация о действиях, выполняемых при создании приложения ActionScript.

Параметры для организации кода

С помощью ActionScript 3.0 можно усовершенствовать все что угодно, начиная от простых графических анимаций до комплексных систем обработки транзакций клиент-сервер. В зависимости от типа создаваемого приложения используйте один или несколько из нижеприведенных способов включения кода ActionScript в проект.

Сохранение кода в кадрах на временной шкале Flash Professional

В среде Flash Professional можно добавлять код ActionScript в любой кадр на временной шкале. Этот код выполняется при воспроизведении ролика, когда воспроизводящая головка воспроизводит этот кадр.

Размещение кода ActionScript в кадрах обеспечивает простой способ добавления свойств для приложений, созданных в среде Flash Professional. Добавить код можно в любой кадр основной временной шкалы любого символа MovieClip. Тем не менее, эта функциональная гибкость связана с некоторыми неудобствами. При создании объемных приложений становится трудно отслеживать взаимосвязь кадров и содержащихся в них сценариев. Такая усложненная структура со временем затрудняет обслуживание приложения.

Многие разработчики упрощают организацию кода ActionScript в среде Flash Professional, размещая код только в первом кадре временной шкалы или в специальном слое документа Flash. Разделение кода упрощает поиск и обслуживание кода в FLA-файлах Flash. Однако один код нельзя использовать в другом проекте Flash Professional, не выполнив копирование и вставку кода в новый файл.

Чтобы облегчить использование кода ActionScript в других проектах Flash Professional в будущем, храните код во внешних файлах ActionScript (текстовые файлы с расширением .as).

Встраивание кода в MXML-файлы Flex

В среде разработки Flex, такой как Flash Builder, можно добавлять код ActionScript в тег `<fx:Script>` MXML-файла среды Flex. Однако этот способ приводит к усложнению крупных проектов и использованию одного кода в другом проекте Flex. Чтобы облегчить использование кода ActionScript в других проектах Flex в будущем, храните код во внешних файлах ActionScript.

Примечание. *Можно определить исходный параметр для тега `<fx:Script>`. Использование исходного параметра позволяет «импортировать» код ActionScript из внешнего файла, как если бы он был напечатан непосредственно в теге `<fx:Script>`. Тем не менее, используемый исходный файл не может определить свой собственный класс, что ограничивает возможности его повторного использования.*

Сохранение кода в файлах ActionScript

Если проект включает объемный код ActionScript, лучше всего разместить этот код в отдельных исходных файлах ActionScript (то есть в текстовых файлах с расширением .as). Файл ActionScript можно структурировать одним из двух способов в зависимости от характера его использования в приложении.

- Неструктурированный код ActionScript: строки кода ActionScript, включая операторы или определения функций, которые написаны так, как если бы они вводились непосредственно в скрипт временной шкалы или MXML-файл.

  Код ActionScript, написанный таким образом, доступен с помощью оператора `include` в среде ActionScript или теге `<fx:Script>` MXML-файла Flex. Оператор `include` среды ActionScript передает в компилятор команду добавления содержимого внешнего файла ActionScript в определенное положение и указанную область скрипта. Результат аналогичен тому, который получается, когда код вводится напрямую. В языке MXML использование тега `<fx:Script>` с исходным атрибутом определяет внешний код ActionScript, загружаемый компилятором в этой части приложения. Например, следующий тег загружает внешний файл ActionScript с именем Box.as:

  ```
  <fx:Script source="Box.as" />
  ```

- Определение класса ActionScript: определение класса ActionScript, включая определение его метода и свойства.

  При определении класса доступ к коду ActionScript в классе можно получить, создав экземпляр класса и используя его свойства, методы и события. Использование собственных классов равнозначно использованию любых встроенных классов ActionScript и включает две части:

  - Используйте инструкцию `import` для задания полного имени класса, чтобы компилятор ActionScript «знал», где его найти. Например, чтобы использовать класс MovieClip в коде ActionScript, импортируйте класс, указав его полное имя, включая пакет и класс:

```
import flash.display.MovieClip;
```

В качестве альтернативы можно импортировать пакет, содержащий класс MovieClip, что эквивалентно записи отдельных инструкций `import` для каждого класса в пакете:

```
import flash.display.*;
```

Классы верхнего уровня являются единственным исключением из правила в отношении того, что классы должны быть импортированы для их использования в коде. Эти классы не заданы в пакете.

- Напишите код, в котором используется это имя класса. Например, объявите переменную с этим классом в качестве его типа данных и создайте экземпляр класса для хранения в переменной. При использовании класса в коде ActionScript в компилятор передаются команды загрузки определения этого класса. Например, для заданного внешнего класса Box этот оператор создает экземпляр класса Box:

  ```
  var smallBox:Box = new Box(10,20);
  ```

  Когда компилятор обрабатывает ссылку на класс Box в первый раз, он ищет доступный исходный код, в котором существует определение класса Box.

Правильный выбор инструмента

Для написания и изменения кода ActionScript можно использовать один из нескольких инструментов (или одновременно несколько инструментов).

Flash Builder

Adobe Flash Builder — это главный инструмент создания проектов в среде Flex или проектов, которые в основном состоят из кода ActionScript. Flash Builder также включает полнофункциональный редактор ActionScript, а также визуальный макет и возможности редактирования MXML. Его можно использовать для создания проектов Flex или проектов, состоящих только из кода ActionScript. Flex обеспечивает некоторые преимущества, включая обширный набор встроенных средств управления интерфейсом, гибкие средства динамического управления компоновкой, встроенные механизмы для работы с внешними источниками данных и возможности связи внешних данных с элементами пользовательского интерфейса. Однако, поскольку для обеспечения этих функций требуется дополнительный код, проекты Flex могут иметь SWF-файл, размер которого превышает аналоги, отличные от Flex.

Используйте Flash Builder при создании полнофункциональных Интернет-приложений на основе данных в среде Flex. Используйте его при редактировании кода ActionScript, кода MXML и визуальной компоновки приложений — все с помощью одного инструмента.

Многие пользователи Flash Professional, создающие полнофункциональные проекты ActionScript, используют инструмент Flash Professional для создания визуальных ресурсов, а инструмент Flash Builder — в качестве редактора кода ActionScript.

Flash Professional

Кроме возможностей создания графики и анимации, Flash Professional включает инструменты для работы с кодом ActionScript. Код можно присоединить к элементам в FLA-файле или внешних файлах, содержащих только код ActionScript. Flash Professional прекрасно подходит для создания проектов, в основном состоящих из анимации или видео. Он незаменим, если большая часть графических ресурсов создается самим пользователем. Другой причиной использования Flash Professional для разработки проектов ActionScript является создание визуальных ресурсов и написание кода в одном приложении. Flash Professional также включает встроенные компоненты пользовательского интерфейса. Можно использовать эти компоненты для уменьшения размера SWF-файла, а визуальные инструменты — для оформления проекта.

Flash Professional включает два инструмента для написания кода ActionScript:

- Панель «Действия»: эта панель, доступная при работе с FLA-файлом, позволяет записывать код ActionScript, присоединенный к кадрам на временной шкале.
- Окно «Сценарий»: окно сценария — это специализированный текстовый редактор для работы файлами кода ActionScript (.as).

Сторонний редактор ActionScript

Поскольку файлы ActionScript (.as) сохраняются как простые текстовые файлы, для создания этих файлов можно использовать любую программу, поддерживающую редактирование простых текстовых файлов. Кроме продуктов ActionScript компании Adobe существуют различные программы с текстовыми редакторами и специальными функциями для ActionScript, созданные сторонними разработчиками. Создать файл MXML или классы ActionScript можно в любой программе с текстовым редактором. После этого можно создать приложение на основе этих файлов с помощью инструмента Flex SDK. Проект может быть создан на основе Flex или быть приложением, состоящим только из кода ActionScript. В противном случае некоторые разработчики используют Flash Builder или редактор ActionScript стороннего производителя для написания классов ActionScript, а также инструмент Flash Professional для создания графического содержимого.

Среди причин выбора редактора ActionScript стороннего производителя можно назвать следующие.

- Вы предпочитаете писать код ActionScript в отдельной программе и создавать визуальные элементы в приложении Flash Professional.
- Вы используете приложение в среде программирования, отличной от ActionScript (например, создание HTML-страниц или приложений на другом языке программирования), если требуется использовать то же самое приложение для создания кода ActionScript.
- Если требуется создать проекты в Flex или только в ActionScript с использованием набора Flex SDK без компонента Flash Professional или Flash Builder.

Ниже перечислены наиболее известные редакторы кодов со специальной поддержкой ActionScript:

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate (с пакетами ActionScript и Flex)

Процесс разработки в ActionScript

Вне зависимости от размеров проекта ActionScript использование процесса создания и разработки приложения позволяет сделать работу более эффективной. Процесс разработки приложения в ActionScript 3.0 включает в себя следующие этапы.

1 Проектирование приложения.

   Перед созданием приложения опишите его в общих чертах.

2 Составление кода в ActionScript 3.0.

Код ActionScript можно создавать с помощью инструментов Flash Professional, Flash Builder, Dreamweaver или текстового редактора.

3 Создайте проект Flash или Flex для выполнения кода.

В среде Flash Professional создайте FLA-файл, настройте параметры публикации, добавьте компоненты пользовательского интерфейса в приложение и ссылку на код ActionScript. В среде Flex определите приложение, добавьте компоненты пользовательского интерфейса с помощью MXML и создайте ссылку на код ActionScript.

4 Публикация и проверка приложения ActionScript.

Тестирование приложения включает выполнение приложения в среде разработки, чтобы убедиться в правильности его работы.

Все действия этих этапов не обязательно выполнять в указанном порядке, не обязательно также полностью завершать один этап, прежде чем приступать к следующему. Например, можно создать один экран приложения (этап 1), а затем создать графику, кнопки и т. д. (этап 3) перед написанием кода ActionScript (этап 2) и тестированием (этап 4). Можно также разработать часть приложения, а затем добавлять отдельные кнопки или элементы интерфейса, создавая для каждого элемента код ActionScript и проверяя его по мере готовности. Важно запомнить эти четыре этапа процесса разработки. Однако при разработке в реальных условиях эффективность повышается, если осуществляется переход между этапами.

Создание пользовательских классов

Процесс создания классов для использования в пользовательском проекте кажется довольно сложным. Тем не менее, самой сложной частью в создании класса является разработка методов, свойств и событий класса.

Стратегия разработки класса

Тема проектирования с ориентацией на объект довольно сложна. Академическим изучением этой дисциплины и ее профессиональным применением можно заниматься всю жизнь. Тем не менее, здесь приводится несколько рекомендаций и подходов, которые помогут сделать первые шаги в этом направлении.

1 Продумайте роль, которую играют экземпляры этого класса в приложении. Обычно объекты выполняют одну из следующих трех функций.

- Объект значений: эти объекты в основном являются контейнерами данных. Вероятнее всего, они имеют несколько свойств и небольшое число методов (а иногда используются без методов). Как правило, они являются закодированным представлением четко заданных элементов. Например, приложение музыкального проигрывателя может включать класс Song (песня), представляющий одну реальную песню, а также класс Playlist (список воспроизведения), представляющий абстрактную группу песен.
- Экранные объекты: эти объекты реально отображаются на экране. В качестве примеров можно назвать выпадающие списки, индикаторные панели состояния, графические элементы, такие как различные существа в видеоиграх, и т. п.

- Структурные объекты приложения: эти объекты выполняют широкие функции, поддерживая обработку данных и логические операции, выполняемые приложением. Например, можно создать объект для выполнения определенных вычислений при имитации биологической среды. Можно создать один объект, отвечающий за синхронизацию значений элемента управления набором номера и показаний громкости в приложении музыкального проигрывателя. Другой объект может управлять правилами в видеоигре. Или можно создать класс для загрузки сохраненного рисунка в приложение рисования.

2 Определите специальные функции для данного класса. Различные функциональные типы часто становятся методами класса.

3 Если класс должен использоваться как объект значений, решите, какие данные будут содержать его экземпляры. Эти элементы являются хорошими кандидатами в свойства.

4 Поскольку класс разрабатывается специально для пользовательского проекта, очень важно снабдить его теми функциями, которые необходимы для конкретного приложения. Попытайтесь дать ответы на следующие вопросы.

- Какие части информации приложение сохраняет, отслеживает и обрабатывает? Ответ на этот вопрос поможет определить все необходимые объекты значений и свойства.
- Какие наборы действий выполняет приложение? Например, что происходит при первой загрузке приложения, при нажатии определенной кнопки, по завершении воспроизведения фильма и так далее? Эти действия отлично подходят на роль методов. Они также могут быть свойствами, если включают изменение отдельных значений.
- Какая информация необходима для выполнения любого из указанных действий? Эти порции информации становятся параметрами метода.
- По мере выполнения приложения что будет изменяться в классе, о чем должна будет передаваться информация в другие компоненты приложения? Эти элементы являются хорошими кандидатами в события.

5 Существует ли объект, аналогичный объекту, который необходимо исключить из-за недостатка функциональных возможностей, которые необходимо добавить? Рассмотрите возможность создания подклассов. (*Подкласс* — это класс, который создается на основе функций существующего класса без определения всех собственных функций.) Например, чтобы создать класс, представляющий визуальный объект на экране, используйте свойства существующего объекта отображения в качестве основы класса. В этом случае объект отображения (такой как MovieClip или Sprite) будет *базовым классом*, а созданный класс будет расширением этого класса.

Дополнительные разделы справки

«Наследование» на странице 113

Написание кода для класса

После представления структуры класса или по крайней мере планирования информации, хранимой в этом классе, и действий, выполняемых этим классом, фактический синтаксис написания этого класса достаточно понятен.

Ниже перечислены основные шаги создания собственного класса ActionScript.

1 Откройте новый текстовый документ в программе текстового редактора ActionScript.

2 Введите инструкцию `class` для задания имени класса. Чтобы добавить оператор `class`, введите выражение `public class` и имя класса. Добавьте открывающиеся и закрывающиеся фигурные скобки, в которых будет указываться содержимое класса (определения методов и свойств). Например:

```
public class MyClass
{
}
```

Слово `public` означает, что этот класс доступен для любого другого кода. Другие возможности описаны в разделе «Атрибуты пространства имен для управления доступом» на странице 97.

3 Введите оператор `package`, чтобы обозначить имя пакета, содержащего класс. Используется следующий синтаксис: слово `package`, после которого указываются имя пакета, открывающиеся и закрывающиеся фигурные скобки, в которых содержится блок оператора `class`). Например, измените код, содержащийся в предыдущем шаге, на следующий:

```
package mypackage
{
    public class MyClass
    {
    }
}
```

4 Определите каждое свойство в классе с помощью оператора `var` в определении класса. Синтаксис совпадает с синтаксисом, используемым для определения любой переменной (с добавлением модификатора `public`). Например, добавление следующих строк между открывающейся и закрывающейся фигурными скобками определения класса приведет к созданию свойств с именами `textProperty`, `numericProperty` и `dateProperty`.

```
public var textProperty:String = "some default value";
public var numericProperty:Number = 17;
public var dateProperty:Date;
```

5 Определите каждый метод в классе с помощью синтаксиса, который использовался для задания функции. Например:

- Для создания метода `myMethod()` введите:

```
public function myMethod(param1:String, param2:Number):void
{
    // do something with parameters
}
```

- Для создания конструктора (специального метода, который является частью процесса создания экземпляра класса) создайте метод с именем, которое в точности совпадает с именем класса:

```
public function MyClass()
{
    // do stuff to set initial values for properties
    // and otherwise set up the object
    textVariable = "Hello there!";
    dateVariable = new Date(2001, 5, 11);
}
```

Если конструктор метода не включен в класс, компилятор автоматически создает пустой конструктор в классе. (Другими словами, конструктор без параметров и операторов.)

Существует несколько дополнительных элементов класса, которые можно определить. Эти элементы являются более сложными.

- *Средства доступа* — это специальные узлы связи между методом и свойством. Когда записывается код для определения класса, средство доступа записывается как метод. При определении свойства можно выполнять несколько действий (а не только чтение и назначение значения). Тем не менее, при создании экземпляра класса средство доступа рассматривается как свойство: его имя используется для чтения и задания значения.
- События в ActionScript не задаются с помощью специального синтаксиса. Вместо этого события в классе определяются с использованием функций класса EventDispatcher.

Дополнительные разделы справки

«Переменные» на странице 99

«Методы» на странице 101

«Методы доступа get и set» на странице 104

Обработка событий

Пример: создание базового приложения

Язык ActionScript 3.0 можно использовать в нескольких средах разработки приложений, включая инструменты Flash Professional и Flash Builder, а также любой текстовый редактор.

В этом примере рассмотрены этапы создания и модернизации простого приложения ActionScript 3.0 с использованием инструмента Flash Professional или Flash Builder. Создаваемое приложение является простой моделью использования внешних файлов классов ActionScript 3.0 в инструментах Flash Professional и Flex.

Проектирование приложения ActionScript

В этом примере приводится стандартное приложение ActionScript, которое называется «Hello World» и имеет простую структуру.

- Приложение называется HelloWorld.
- В нем отображается одно текстовое поле со словами «Hello World!»
- В приложении используется один объектно-ориентированный класс Greeter. Такая структура позволяет использовать класс в проекте Flash Professional или Flex.
- В этом примере сначала создается базовая версия приложения. Затем добавляются функции, чтобы пользователь мог ввести свое имя, а приложение проверило наличие имени в списке известных пользователей.

После этого краткого определения можно приступать к созданию приложения.

Создание проекта HelloWorld и класса Greeter

В проектном задании для приложения Hello World говорится, что его код должен допускать простое повторное использование. Чтобы достичь этой цели, в приложении используется один объектно-ориентированный класс Greeter. Этот класс используется в приложении, создаваемом в инструменте Flash Builder или Flash Professional.

Создание проекта HelloWorld и класса Greeter в Flex

1 В инструменте Flash Builder выберите «Файл» > «Создать» > «Проект Flex».

2 Введите HelloWorld в поле «Имя проекта». Убедитесь, что в качестве типа приложения выбрано значение «Web (выполняется в Adobe Flash Player)», а затем нажмите «Готово».

Flash Builder создает проект и показывает его в окне проводника пакетов. По умолчанию проект содержит файл с именем HelloWorld.mxml, который открывается в редакторе.

3 Теперь, чтобы создать пользовательский файл класса ActionScript в среде Flash Builder, выберите «Файл» > «Создать» > «Класс ActionScript».

4 В диалоговом окне «Новый класс ActionScript» в поле «Имя» введите **Greeter** в качестве имени класса и нажмите кнопку «Готово».

Теперь появится окно редактирования нового файла ActionScript.

В продолжение «Добавление кода к классу Greeter» на странице 29.

Чтобы создать класс Greeter в инструменте Flash Professional, выполните следующие действия:

1 В инструменте Flash Professional выберите «Файл» > «Создать».

2 В диалоговом окне «Создать документ» выберите файл ActionScript и нажмите кнопку «ОК».

Теперь появится окно редактирования нового файла ActionScript.

3 Выберите команды «Файл» > «Сохранить». Выберите папку для размещения приложения, присвойте файлу ActionScript имя **Greeter.as** и нажмите кнопку «ОК».

После этого можно перейти к «Добавление кода к классу Greeter» на странице 29.

Добавление кода к классу Greeter

Класс Greeter определяет объект `Greeter`, который используется в приложении HelloWorld.

Добавление кода к классу Greeter

1 Введите следующий код в новом файле (часть кода может быть уже добавлена):

```
package
{
    public class Greeter
    {
        public function sayHello():String
        {
            var greeting:String;
            greeting = "Hello World!";
            return greeting;
        }
    }
}
```

Класс Greeter включает один метод `sayHello()`, который возвращает строку с фразой «Hello World!».

2 Для сохранения этого файла ActionScript выберите команды «Файл» > «Сохранить».

Класс Greeter теперь готов для использования в приложении.

Создание приложения с использованием кода ActionScript

Класс Greeter, который был только что создан, определяет самодостаточный набор программных функций, однако он не представляет собой законченное приложение. Для использования класса необходимо создать документ Flash Professional или проект Flex.

При создании кода необходимо использовать экземпляр класса Greeter. Ниже описана процедура использования класса Greeter в приложении.

Чтобы создать приложение ActionScript с использованием инструмента Flash Professional, выполните следующие действия.

1. Выберите команды «Файл» > «Создать».
2. В диалоговом окне «Новый документ» выберите «Файл Flash (ActionScript 3.0)» и нажмите кнопку «ОК».

   Открывается окно создания документа.
3. Выберите команды «Файл» > «Сохранить». Выберите папку, в которой находится файл класса Greeter.as, присвойте документу Flash имя **HelloWorld.fla** и нажмите кнопку «ОК».
4. В палитре инструментов Flash Professional выберите инструмент «Текст». Перетащите его в рабочей области, чтобы определить новое текстовое поле шириной приблизительно 300 пикселов и высотой 100 пикселов.
5. На панели «Свойства» при все еще выделенном в рабочей области текстовом поле задайте тип текста как «Динамический текст» и введите **mainText** в качестве имени экземпляра текстового поля.
6. Щелкните кнопкой мыши первый кадр временной шкалы. Откройте панель «Действия», выбрав меню «Окно» > «Действия».
7. На панели «Действия» введите следующий сценарий:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello();
```

8. Сохраните файл.

После этого можно перейти к «Публикация и проверка приложения ActionScript» на странице 31.

Чтобы создать приложение ActionScript с использованием инструмента Flash Builder, выполните следующие действия.

1. Откройте файл HelloWorld.mxml и добавьте код в соответствии со следующей распечаткой:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400"/>

</s:Application>
```

Этот проект Flex включает четыре тега MXML:

- Тег `<s:Application>` определяет контейнер приложения.
- Тег `<s:layout>` определяет стиль компоновки (вертикальная компоновка) для тега Application.
- Тег `<fx:Script>` включает некоторую часть кода ActionScript.
- Тег `<s:TextArea>` определяет поле, в котором отображаются текстовые сообщения для пользователя.

Код в теге `<fx:Script>` определяет метод `initApp()`, который вызывается при загрузке приложения. Метод `initApp()` задает текстовое значение текстовой области `mainTxt` для строки «Hello World!», возвращенной методом `sayHello()` только что созданного класса Greeter.

2 Для сохранения приложения выберите команды «Файл» > «Сохранить».

После этого можно перейти к «Публикация и проверка приложения ActionScript» на странице 31.

Публикация и проверка приложения ActionScript

Разработка программного обеспечения — это интерактивный процесс. После написания кода его необходимо компилировать и редактировать до тех пор, пока он не будет полностью соответствовать поставленным задачам. Необходимо запустить скомпилированное приложение и проверить, что оно выполняет задачи, описанные в задании. Если нет, необходимо редактировать код, пока не будет получен нужный результат. В средах разработки Flash Professional и Flash Builder предусмотрено несколько способов для публикации, проверки и отладки приложений.

Ниже приводятся основные этапы проверки приложения HelloWorld в каждой из упомянутых сред.

Чтобы опубликовать и проверить приложение ActionScript с использованием инструмента Flash Professional, выполните следующие действия.

1 Опубликуйте приложение и проверьте его на наличие ошибок компиляции. В инструменте Flash Professional выберите «Управление» > «Тестировать ролик», чтобы скомпилировать код ActionScript и выполнить приложение HelloWorld.

2 Если при тестировании приложения в окне «Вывод» отображаются ошибки и предупреждения, исправьте эти ошибки в файле HelloWorld.fla или HelloWorld.as. Затем повторно проверьте приложение.

3 При отсутствии ошибок компиляции приложение Hello World появится в окне Flash Player.

После успешного создания простого, но законченного объектно-ориентированного приложения ActionScript 3.0, можно приступить к «Модернизация приложения HelloWorld» на странице 32.

Чтобы опубликовать и проверить приложение ActionScript с использованием инструмента Flash Builder, выполните следующие действия.

1 Выберите «Выполнить» > «Выполнить HelloWorld».

2 Приложение HelloWorld будет запущено.

- Если при тестировании приложения в окне «Вывод» отображаются ошибки и предупреждения, исправьте эти ошибки в файле HelloWorld.mxml или Greeter.as. Затем повторно проверьте приложение.
- При отсутствии ошибок компиляции приложение Hello World появится в открывшемся окне обозревателя. На экране должен отображаться текст «Hello World!»

После успешного создания простого, но законченного объектно-ориентированного приложения ActionScript 3.0 можно приступить к «Модернизация приложения HelloWorld» на странице 32.

Модернизация приложения HelloWorld

Чтобы сделать приложение более интересным, можно ввести в него подтверждение имени пользователя после сверки с заданным списком имен.

Прежде всего нужно обновить класс Greeter, расширив его функциональные возможности. Затем следует обновить приложение, чтобы оно могло использовать новые функции.

Обновление файла Greeter.as

1 Откройте файл Greeter.as.

2 Измените содержимое файла следующим образом (новые и измененные строки выделены жирным шрифтом):

```
package
{
    public class Greeter
    {
        /**
         * Defines the names that receive a proper greeting.
         */
        public static var validNames:Array = ["Sammy", "Frank", "Dean"];

        /**
         * Builds a greeting string using the given name.
         */
        public function sayHello(userName:String = ""):String
        {
            var greeting:String;
            if (userName == "")
            {
                greeting = "Hello. Please type your user name, and then press "
                           + "the Enter key.";
            }
            else if (validName(userName))
            {
                greeting = "Hello, " + userName + ".";
            }
            else
            {
                greeting = "Sorry " + userName + ", you are not on the list.";
            }
            return greeting;
        }

        /**
         * Checks whether a name is in the validNames list.
         */
        public static function validName(inputName:String = ""):Boolean
        {
            if (validNames.indexOf(inputName) > -1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

Класс Greeter имеет теперь новые свойства:

- В массиве `validNames` содержится список разрешенных имен пользователей. При загрузке класса Greeter в массиве устанавливается список из трех имен.

- Метод `sayHello()` принимает имя пользователя и изменяет приветствие в зависимости от определенных условий. Если имя пользователя `userName` задано пустой строкой (`""`), свойство `greeting` устанавливается на запрос имени пользователя. Если имя пользователя принято, приветствие выглядит так: `"Hello, `*userName*`."` И, наконец, при невыполнении предыдущих двух условий переменная `greeting` устанавливается таким образом: `"Sorry `*userName*`, you are not on the list."` («Извините, [имя пользователя], Вас нет в списке»).
- Метод `validName()` возвращает истинное значение `true`, если введенное имя `inputName` найдено в массиве `validNames`, и возвращает ложное значение `false`, если имя не найдено. Инструкция `validNames.indexOf(inputName)` сверяет каждую строку массива `validNames` со строкой введенного имени `inputName`. Метод `Array.indexOf()` возвращает положение указателя первого экземпляра объекта в массиве. Он возвращает значение -1, если объект не найден в массиве.

Затем необходимо изменить файл приложения, в котором существует ссылка на этот класс ActionScript.

Чтобы изменить приложение с использованием инструмента Flash Professional, выполните следующие действия.

1. Откройте файл HelloWorld.fla.
2. Измените сценарий в Кадре 1 так, чтобы пустая строка (`""`) перешла в метод `sayHello()` класса Greeter:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");
```

3. Выберите инструмент «Текст» в палитре инструментов. Создайте два новых текстовых поля в рабочей области. Расположите их рядом друг с другом под существующим текстовым полем `mainText`.
4. В первом новом текстовом поле, которое является меткой, введите текст **Имя пользователя:**.
5. Выберите другое текстовое поле и установите его тип в инспекторе свойств как «Вводимый текст». В качестве типа строки выберите Single line (отдельная строка). Введите **textIn** в качестве имени экземпляра.
6. Щелкните кнопкой мыши первый кадр временной шкалы.
7. На панели «Действия» добавьте следующие строки в конце имеющегося сценария:

```
mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

Новый код добавляет следующие функциональные возможности:

- Первые две строки просто задают границы для двух текстовых полей.
- Входное текстовое поле, такое как `textIn`, имеет набор событий, которые оно может распределять. Метод `addEventListener()` позволяет задать функцию, которая запускается при возникновении события определенного типа. В данном случае этим событием будет нажатие клавиши на клавиатуре.

- Настраиваемая функция `keyPressed()` проверяет, будет ли нажата именно клавиша Enter. Если требуемая клавиша нажата, метод `sayHello()` объекта `myGreeter` передает текст из текстового поля `textIn` в качестве параметра. Этот метод возвращает строку приветствия на основе переданного в него значения. Возвращенная строка затем назначается свойству `text` текстового поля `mainText`.

Полный сценарий для Кадра 1 выглядит следующим образом:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");

mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

8 Сохраните файл.

9 Для запуска приложения выберите команды «Управление» > «Тестировать ролик».

При выполнении приложения появляется приглашение к вводу имени пользователя. Если имя принимается программой, появится подтверждающее сообщение «hello».

Чтобы изменить приложение с использованием инструмента Flash Builder, выполните следующие действия.

1 Откройте файл HelloWorld.mxml.

2 Затем измените тег `<mx:TextArea>`, чтобы показать пользователю, что текст используется только для отображения. Измените цвет фона на светло-серый и задайте для атрибута `editable` значение `false`:

```
    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false" />
```

3 Теперь добавьте следующие строки непосредственно после закрывающего тега `<s:TextArea>`. Эти строки создают компонент TextInput, который позволяет пользователю вводить значение имени пользователя:

```
    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>
```

Атрибут `enter` определяет действия, которые выполняются при нажатии пользователем клавиши Enter в поле `userNameTxt`. В этом примере код передает текст, введенный в поле, в метод `Greeter.sayHello()`. Приветствие в поле `mainTxt` изменяется соответственно.

Файл HelloWorld.mxml имеет следующий вид:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false"/>

    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>

</s:Application>
```

4 Сохраните отредактированный файл HelloWorld.mxml. Выберите «Выполнить» > «Выполнить HelloWorld» для выполнения приложения.

При выполнении приложения появляется приглашение к вводу имени пользователя. Если имя (Sammy, Frank или Dean) принимается программой, появится подтверждающее сообщение «`Hello, `*userName*».

Глава 3. Язык ActionScript и его синтаксис

ActionScript 3.0 включает основной язык ActionScript, а также прикладной программный интерфейс Adobe Flash Platform. Базовый язык — это часть кода ActionScript, которая определяет синтаксис языка, а также типы данных верхнего уровня. ActionScript 3.0 обеспечивает программные ресурсы для сред выполнения Adobe Flash Platform: Adobe Flash Player и Adobe AIR.

Обзор языка

В основу языка ActionScript 3.0 положены объекты, они являются его главным конструктивным материалом. Каждая объявленная переменная, каждая написанная функция, каждый созданный экземпляр класса являются объектами. Программу на ActionScript 3.0 можно рассматривать как группу объектов, выполняющих задачи, реагирующих на события и взаимодействующих друг с другом.

Программисты, знакомые с объектно-ориентированным программированием в Java или C++, могут воспринимать объекты как модули, включающие два типа: данные, хранимые в соответствующих переменных или свойствах, а также поведение, реализуемое с помощью методов. ActionScript 3.0 определяет объекты подобным образом, но немного по-другому. В ActionScript 3.0 объекты представляют собой просто наборы свойств. Эти свойства выступают в качестве контейнеров, которые могут содержать не только данные, но также функции или другие объекты. Если функция связана с объектом таким образом, она называется методом.

Хотя определение ActionScript 3.0 и может показаться немного странным для имеющего опыт программирования на Java или C++, на практике определение типов объектов с помощью классов ActionScript 3.0 сходно со способом, которым классы определяются в Java или C++. Различие между двумя определениями объектов важно, когда обсуждается объектная модель ActionScript и другие расширенные возможности, но в большинстве других ситуаций термин *свойства* означает переменные, являющиеся членами класса, как альтернативу методам. В справочнике Adobe ActionScript 3.0 для Adobe Flash Platform, к примеру, термин *свойства* обозначает переменные или свойства установки и получения атрибутов. Термин *методы* здесь используется для обозначения функций, являющихся частью класса.

Единственным небольшим различием между классами в ActionScript и классами в Java или C++ является то, что в ActionScript классы - это не просто абстрактные сущности. В ActionScript классы представляются *объектами классов*, хранящими свойства и методы этого класса. Это позволяет реализовывать методики, которые могут показаться незнакомыми программистам на Java и C++, например, включение инструкций или исполняемого программного кода в верхний уровень класса или пакета.

Другим различием между классами ActionScript и классами в Java или C++ является то, что каждый класс ActionScript обладает так называемым *объектомпрототипа*. В предыдущих версиях ActionScript объекты прототипов, связанные вместе в *цепочки прототипов,* совместно служили в качестве основания всей иерархии наследования класса. Однако в ActionScript 3.0 объекты прототипов играют лишь малую роль в системе наследования. Объект прототипа также может быть полезен, но как альтернатива статическим свойствам и методам, если требуется обеспечить совместное использование свойства и его значений всеми экземплярами класса.

В прошлом опытные программисты ActionScript могли напрямую управлять цепочкой прототипов с помощью специальных встроенных элементов языка. Теперь, когда в языке предлагается более зрелая реализация интерфейса программирования, основанного на классах, многие из этих специальных элементов языка, таких как `__proto__` и `__resolve`, больше не являются частью языка. Кроме того, оптимизация внутреннего механизма наследования, обеспечивающая значительный прирост производительности, предотвращает прямой доступ к механизму наследования.

Объекты и классы

В ActionScript 3.0 каждый объект определяется классом. Класс можно рассматривать как шаблон или проект типа объекта. Определения класса могут включать переменные и константы, которые содержат значения данных и методов, являющихся функциями, содержащими поведение, связанное с этим классом. Значения, хранимые в свойствах, могут быть *примитивными значениями* или другими объектами. Примитивные значения — это числа, строки или логические значения.

ActionScript содержит ряд встроенных классов, являющихся частью языка ядра. Некоторые из этих встроенных классов, например Number, Boolean и String, отражают примитивные значения, доступные в ActionScript. Другие же, такие как классы Array, Math и XML, определяют более сложные объекты.

Все классы, независимо от того, являются ли они встроенными или определяемыми пользователем, выводятся из класса Object. Программистам, имеющим опыт работы с предыдущей версией ActionScript, важно обратить внимание на то, что тип данных Object больше не является типом данных по умолчанию, даже несмотря на то, что все другие классы являются его производными. В ActionScript 2.0 следующие две строки программного кода были эквивалентны, поскольку отсутствие аннотации типа означало, что переменная будет иметь тип Object.

```
var someObj:Object;
var someObj;
```

В ActionScript 3.0 вводится концепция переменных без объявления типа, которые можно назначать следующими двумя способами.

```
var someObj:*;
var someObj;
```

Переменная без объявления типа отличается от переменной с типом Object. Основным отличием является то, что переменные без объявления типа могут содержать специальное значение `undefined`, в то время как переменные с типом Object не могут содержать такое значение.

Можно определить свои собственные классы, используя ключевое слово `class`. Свойства класса можно объявить тремя способами: константы могут быть определены с помощью ключевого слова `const`, переменные определяются с помощью ключевого слова `var`, а свойства getter и setter определяются с использованием атрибутов `get` и `set` в объявлении метода. Методы можно объявлять с помощью ключевого слова `function`.

Экземпляр класса создается с помощью оператора `new`. В следующем примере создается экземпляр класса Date с названием `myBirthday`.

```
var myBirthday:Date = new Date();
```

Пакеты и пространства имен

Пакеты и пространства имен являются взаимосвязанными концепциями. Пакеты позволяют связывать определения классов вместе таким способом, что облегчается совместное использование кода и снижается вероятность конфликтов имен. Пространства имен позволяют управлять видимостью идентификаторов, например именами свойств и методов, их можно применять к программному коду независимо от того, внутри или вне пакета он находится. Пакеты позволяют организовать файлы классов, а пространства имен позволяют управлять видимостью отдельных свойств и методов.

Пакеты

Пакеты в ActionScript 3.0 реализуются с помощью пространств имен, но эти два понятия не являются синонимами. При объявлении пакета явно создается специальный тип пространства имен, который гарантированно будет известен во время компиляции. Пространства имен, когда создаются явным образом, не обязательно известны во время компиляции.

В следующем примере для создания простого пакета, содержащего один класс, используется директива `package`.

```
package samples
{
    public class SampleCode
    {
        public var sampleGreeting:String;
        public function sampleFunction()
        {
            trace(sampleGreeting + " from sampleFunction()");
        }
    }
}
```

Именем класса в данном случае является SampleCode. Поскольку класс находится внутри пакета примеров, компилятор автоматически уточняет имя класса во время компиляции, воссоздавая полное имя: samples.SampleCode. Компилятор также уточняет имена любых свойств или методов, так что имена `sampleGreeting` и `sampleFunction()` становятся соответственно `samples.SampleCode.sampleGreeting` и `samples.SampleCode.sampleFunction()`.

Многие разработчики, особенно те, кто программирует на Java, могут захотеть размещать классы только на верхнем уровне пакетов. В тоже время ActionScript 3.0 поддерживает на верхнем уровне пакета не только классы, но и переменные, функции и даже инструкции. Одним из более сложных способов использования этой функции является определение пространства имен на верхнем уровне пакета, чтобы оно было доступно для всех классов этого пакета. Однако следует обратить внимание, что на верхнем уровне пакета разрешены только два спецификатора доступа — `public` и `internal`. В отличии от Java, где можно объявлять вложенные классы частными, в ActionScript 3.0 не поддерживаются ни вложенные, ни частные классы.

Во многом другом, однако, пакеты ActionScript 3.0 сходны с пакетами в языке программирования Java. Как видно из предыдущего примера, полностью уточненные ссылки на пакет передаются с помощью оператора точки (.), как это делается в Java. Можно использовать пакеты для организации собственного программного кода в интуитивно понятную иерархическую структуру, с которой смогут работать другие программисты. Это упрощает совместную работу с программными кодами, позволяя создавать собственные пакеты для использования другими разработчиками, а также применять пакеты, созданные другими, в собственном программном коде.

Использование пакетов также помогает соблюсти уникальность имен применяемых идентификаторов и предотвратить конфликты с именами других идентификаторов. Действительно, кто-то наверное скажет, что это основное преимущество использования пакетов. Например, два программиста, которым необходимо совместно использовать программные коды, могут по отдельности создать класс с названием SampleCode. Без пакетов при этом возникнет конфликт имен, единственным решением которого будет переименование одного из классов. Однако благодаря использованию пакетов конфликт имен легко предотвращается, поскольку один, а лучше оба класса можно поместить в пакеты с уникальными именами.

Также в имена пакетов можно включить встроенные точки, чтобы создать вложенные пакеты. Это позволяет создавать иерархическую структуру пакетов. Хорошим примером служит пакет flash.display, предоставляемый в рамках ActionScript 3.0. Пакет flash.display вложен в пакет flash.

Язык ActionScript 3.0 главным образом организован внутри пакета flash. Например, пакет flash.display содержит прикладной программный интерфейс списка отображения, а пакет flash.events содержит новую модель событий.

Создание пакетов

ActionScript 3.0 обеспечивает значительную гибкость в способах организации пакетов, классов и исходных файлов. В предыдущих версиях ActionScript разрешалось размещать только один класс в одном исходном файле, а также требовалось совпадение имен файла и класса. ActionScript 3.0 позволяет включать несколько классов в один исходный файл, но только один класс в каждом файле может быть сделан доступным для программного кода, внешнего по отношению к файлу. Говоря другими словами, только один класс в каждом файле может быть объявлен в декларации пакета. Необходимо объявлять любые дополнительные классы вне определения пакета, что сделает такие классы невидимыми для программного кода вне этого исходного файла. Имя класса, объявляемого в определении пакета, должно соответствовать имени исходного файла.

ActionScript 3.0 обеспечивает более гибкие возможности при объявлении пакетов. В предыдущих версиях ActionScript пакеты всего лишь представляли собой каталоги, в которые помещались исходные файлы, и эти пакеты объявлялись не с помощью инструкции `package`, а просто имя пакета включалось как часть полного уточненного имени класса в определение класса. Хотя пакеты по-прежнему представляют собой каталоги в ActionScript 3.0, они могут содержать не только классы. В ActionScript 3.0 для объявления пакета используется инструкция `package`, что означает, что также можно объявлять переменные, функции и пространства имен на верхнем уровне пакета. На верхнем уровне пакета также можно добавлять выполняемые инструкции. Если на верхнем уровне пакета объявляются переменные, функции или пространства имен, то единственными атрибутами, доступными на этом уровне, являются `public` и `internal`. Только одно объявление пакета в файле может использовать атрибут `public` независимо от того, объявляется ли при этом класс, переменная, функция или пространство имен.

Пакеты особенно полезны при упорядочивании программных кодов, а также для предотвращения конфликтов имен. Но не нужно путать концепцию пакетов с несвязанной с ней концепцией наследования классов. Два класса, находящиеся в одном и том же пакете, имеют общее пространство имен, но совсем не обязательно, что они связаны между собой каким-то иным образом. Сходным образом вложенный пакет может не иметь семантических связей со своим родительским пакетом.

Импорт пакетов

Если требуется использовать класс, находящийся внутри пакета, необходимо выполнить импорт либо этого пакета, либо определенного класса. Это отличается от технологии, используемой в ActionScript 2.0, где импорт классов был необязателен.

Например, рассмотрим пример класса SampleCode, представленного ранее. Если класс находится в пакете с именем samples, необходимо использовать одну из следующих инструкций import, прежде чем использовать класс SampleCode.

```
import samples.*;
```

или

```
import samples.SampleCode;
```

В целом, инструкции `import` должны быть максимально конкретизированы. Если планируется использовать только класс SampleCode из пакета samples , необходимо импортировать только класс SampleCode, но не весь пакет, к которому он принадлежит. Импорт всего пакета может привести к неожиданным конфликтам имен.

Необходимо также размещать исходный код, определяющий пакет или класс, внутри *подкаталога классов*. Путь к классам — это задаваемый пользователем список путей к локальным каталогам, определяющий, где компилятор ищет импортированные пакеты и классы. Подкаталог классов иногда называют *путем построения* или *исходным путем*.

После того как класс и пакет правильно импортированы можно использовать либо полностью уточненное имя класса (samples.SampleCode), либо просто само имя класса (SampleCode).

Полностью уточненные имена очень полезны в тех случаях, когда одинаковые имена классов, методов или свойств приводят к неоднозначностям в программных кодах, но могут усложнить управление, если использовать такие имена для всех идентификаторов. Например, использование полностью уточненных имен приводит к слишком подробному коду при создании экземпляра класса SampleCode.

```
var mySample:samples.SampleCode = new samples.SampleCode();
```

По мере увеличения уровней вложенности пакетов читаемость такого программного кода снижается. Поэтому в тех случаях, когда есть уверенность, что неоднозначность идентификаторов может вызвать проблемы, можно упрощать программный код, используя простые идентификаторы. Например, для создания нового экземпляра класса SampleCode требуется гораздо меньше информации, если использовать только идентификатор класса.

```
var mySample:SampleCode = new SampleCode();
```

Если попытаться использовать имена идентификаторов без предварительного импорта соответствующего пакета или класса, компилятор не сможет найти определения классов. С другой стороны, если импортировать пакет или класс, то любая попытка определить имя, конфликтующее с именем импортированного объекта, приведет к возникновению ошибки.

Если создается пакет, спецификатор доступа по умолчанию для всех членов этого пакета установлен как `internal`, и это означает, что по умолчанию члены пакета видны только другим членам этого пакета. Если требуется, чтобы класс был доступен для программного кода вне пакета, необходимо объявить этот класс как `public`. Например, в следующем пакете содержатся два класса, SampleCode и CodeFormatter.

```
// SampleCode.as file
package samples
{
    public class SampleCode {}
}

// CodeFormatter.as file
package samples
{
    class CodeFormatter {}
}
```

Класс SampleCode виден вне пакета, потому что он объявлен как класс `public`. Класс CodeFormatter, в то же время, видим только внутри самого пакета samples. При попытке обращения к классу CodeFormatter за пределами пакета samples появится сообщение об ошибке, как показано в следующем примере.

```
import samples.SampleCode;
import samples.CodeFormatter;
var mySample:SampleCode = new SampleCode(); // okay, public class
var myFormatter:CodeFormatter = new CodeFormatter(); // error
```

Если требуется, чтобы оба класса были доступны за пределами пакета, необходимо объявить оба этих класса как `public`. Нельзя применить атрибут `public` при объявлении пакета.

Полностью уточненные имена полезны при разрешении конфликтов имен, которые могут произойти при использовании пакетов. Такие сценарии могут возникать, если импортируются два пакета, определяющие классы с одинаковым идентификатором. Например, рассмотрим следующий пакет, в котором также имеется класс с названием SampleCode.

```
package langref.samples
{
    public class SampleCode {}
}
```

Если импортировать оба класса, как показано далее, возникнет конфликт имен при использовании класса SampleCode.

```
import samples.SampleCode;
import langref.samples.SampleCode;
var mySample:SampleCode = new SampleCode(); // name conflict
```

Компилятор не сможет распознать, какой класс SampleCode использовать. Чтобы разрешить этот конфликт, необходимо использовать полностью уточненное имя.

```
var sample1:samples.SampleCode = new samples.SampleCode();
var sample2:langref.samples.SampleCode = new langref.samples.SampleCode();
```

Примечание. *Программисты со стажем работы на C++ часто путают инструкцию `import` с `#include`. Директива `#include` необходима в C++, поскольку компиляторы этого языка одновременно обрабатывают только один файл и не будут открывать другие файлы в поисках определений классов, пока явно не включен файл заголовка. В ActionScript 3.0 также есть директива `include`, но она не предназначена для импорта классов и пакетов. Для импорта классов и пакетов в ActionScript 3.0 необходимо использовать инструкцию `import` и помещать исходный файл, содержащий пакет, в подкаталог классов.*

Пространства имен

Пространства имен позволяют управлять доступностью создаваемых свойств и методов. Можно рассматривать спецификаторы управления доступом `public`, `private`, `protected` и `internal` как встроенные пространства имен. Если эти предварительно определенные спецификаторы управления доступом не соответствуют имеющимся требованиям, можно создать собственные пространства имен.

Тем, кто имеет представление о пространствах имен XML, большая часть обсуждаемой здесь информации покажется знакомой, хотя синтаксис и подробные сведения реализации пространств имен в ActionScript слегка отличаются от используемых в XML. Но даже если раньше работать с пространствами имен не приходилось, эта концепция достаточно понятна, хотя в ее реализации используется специфическая терминология, которую нужно освоить.

Чтобы разобраться, как работают пространства имен, полезно знать, что имя свойства или метода всегда состоит из двух частей: идентификатора и пространства имен. Идентификатор — это та часть, которая обычно воспринимается как имя. Например, идентификаторами в следующем классе являются `sampleGreeting` и `sampleFunction()`.

```
class SampleCode
{
    var sampleGreeting:String;
    function sampleFunction () {
        trace(sampleGreeting + " from sampleFunction()");
    }
}
```

В тех случаях, когда определения не предваряются атрибутом пространства имен, их имена по умолчанию уточняются как относящиеся к пространству имен `internal`, и это означает, что они видимы для вызова только в рамках данного пакета. Если для компилятора задан строгий режим работы, он создает предупреждение о применении пространства имен `internal` к любому идентификатору без атрибута пространства имен. Для обеспечения широкого доступа к идентификатору необходимо специально предварять его имя атрибутом `public`. В предыдущем примере программного кода для обоих классов `sampleGreeting` и `sampleFunction()` в качестве значения пространства имен указано `internal`.

При использовании пространств имен необходимо выполнить три основных шага. Прежде всего необходимо определить пространство имен, используя ключевое слово `namespace`. Например, в следующем программном коде определяется пространство имен `version1`.

```
namespace version1;
```

Во-вторых, необходимо применить пространство имен, используя его вместо спецификатора контроля доступа при объявлении свойства или метода. В следующем примере функция с именем `myFunction()` размещается в пространстве имен `version1`.

```
version1 function myFunction() {}
```

В-третьих, как только пространство имен будет применено, к нему можно адресоваться с помощью директивы `use` или уточняя имя идентификатора указанием пространства имен. В следующем примере функция `myFunction()` вызывается с использованием директивы `use`.

```
use namespace version1;
myFunction();
```

Для вызова функции `myFunction()` можно также использовать уточненное имя, как показано в следующем примере.

```
version1::myFunction();
```

Определение пространств имен

Пространства имен содержат одно значение, универсальный идентификатор ресурса (URI), который иногда называется *именем пространства имен*. Идентификатор URI позволяет обеспечить уникальность определения пространства имен.

Пространство имен создается путем объявления определения пространства имен одним из следующих способов. Можно либо определить пространство имен путем явного указания URI, как это делалось бы для пространства имен XML, либо можно пропустить URI. В следующем примере показано, как можно определить пространство имен с помощью URI.

```
namespace flash_proxy = "http://www.adobe.com/flash/proxy";
```

Идентификатор URI выступает в качестве уникальной строки идентификации пространства имен. Если идентификатор URI пропущен, как показано в следующем примере, компилятор создаст уникальную внутреннюю строку идентификации вместо этого URI. Доступ пользователя к этой внутренней строке идентификации невозможен.

```
namespace flash_proxy;
```

Как только пространство имен определено, с использованием URI или без его использования, это пространство имен не может быть переопределено в той же области действия. Попытка определить пространство имен, которое уже было определено ранее в этой области действия, приведет к ошибке компилятора.

Если пространство имен определяется в пакете или классе, это пространство имен может быть невидимо для программного кода за пределами этого пакета или класса до тех пор, пока не будет использован соответствующий спецификатор контроля доступа. Например, в следующем программном коде показано пространство имен `flash_proxy`, определенное в пакете flash.utils. В следующем примере отсутствие спецификатора контроля доступа означает, что пространство имен `flash_proxy` будет видимо только программному коду в самом пакете flash.utils и будет недоступно любому программному коду вне пакета.

```
package flash.utils
{
    namespace flash_proxy;
}
```

В следующем программном коде атрибут `public` используется для открытия доступа к пространству имени `flash_proxy` извне этого пакета.

```
package flash.utils
{
    public namespace flash_proxy;
}
```

Применение пространств имен

Применение пространства имен означает размещение определения в пространстве имен. В число определений, которые можно помещать в пространства имен, входят функции, переменные и постоянные (нельзя поместить класс в произвольное пространство имен).

Например, рассмотрим функцию, объявленную с использованием пространства имен для контроля доступа `public`. Использование атрибута `public` в определении функции вызывает ее размещение в пространстве имен public, что делает эту функцию доступной для всех программных кодов. Как только пространство имен задано, можно использовать его тем же способом, что и атрибут `public`, и определение доступно программному коду, из которого можно обращаться к произвольно заданным пространствам имен. Например, если определяется пространство имен `example1`, можно добавить метод с названием `myFunction()`, используя в качестве атрибута `example1`, как показано в следующем примере.

```
namespace example1;
class someClass
{
    example1 myFunction() {}
}
```

Объявление метода `myFunction()` с использованием пространства имен `example1` в качестве атрибута означает, что этот метод принадлежит пространству имен `example1`.

При применении пространств имен необходимо учитывать следующие правила.

- Для каждого объявления можно применять только одно пространство имен.

- Невозможно применить атрибут пространства имен более чем к одному определению одновременно. Говоря другими словами, если потребуется применить пространство имен к десяти различным функциям, необходимо будет добавить это пространство имен в качестве атрибута к определению каждой из десяти функций.
- Если применяется пространство имен, нельзя также указывать спецификатор контроля доступа, поскольку пространства имен и спецификаторы контроля доступа взаимоисключают друг друга. То есть нельзя объявить функцию или свойство с атрибутом `public`, `private`, `protected` или `internal` в дополнение к применению пространства имен.

Привязка пространств имен

Нет необходимости явно выполнять привязку к пространствам имен, если используется метод или свойство, объявленные с каким-либо пространством имен для контроля доступа, например атрибутами `public`, `private`, `protected` и `internal`. Доступ к этим специальным пространствам имен управляется контекстом. Например, определения, помещенные в пространство имен `private`, автоматически доступны для программных кодов в рамках данного класса. Однако для самостоятельно определенных пространств имен такой контекстной чувствительности не существует. Чтобы использовать метод или свойство, помещенное в произвольное пространство имен, необходимо создать ссылку на это пространство имен.

Привязку пространства имен можно выполнить с помощью директивы `use namespace` или можно уточнить имя с помощью пространства имен, используя в качестве уточняющего имя знака пунктуации (`::`). Привязка пространства имен с помощью директивы `use namespace` открывает пространство имен, так что его можно применять к любому неуточненному идентификатору. Например, если определено пространство имен `example1`, можно обращаться к именам в этом пространстве, используя директиву `use namespace example1`.

```
use namespace example1;
myFunction();
```

Одновременно может быть открыто сразу несколько пространств имен. Как только пространство имен открыто с помощью директивы `use namespace`, оно становится открытым для всего блока программного кода, в котором оно открыто. Не существует способов явным образом закрыть пространство имен.

Открытие нескольких пространств имен увеличивает вероятность возникновения конфликтов имен. Если предпочтительнее не открывать пространство имен, можно избежать директивы `use namespace`, просто уточняя имя метода или свойства с помощью пространства имен и уточняющего имя знака пунктуации. Например, в следующем программном коде показано, как можно уточнить имя `myFunction()`, используя пространство имен `example1`.

```
example1::myFunction();
```

Использование пространств имен

Можно найти конкретный пример пространства имен, используемого для предотвращения конфликтов имен в классе flash.utils.Proxy, который входит в состав ActionScript 3.0. Класс Proxy, заменивший класс `Object.__resolve`, который использовался в ActionScript 2.0, позволяет перехватывать обращения к неопределенным свойствам или методам перед тем, как произойдет ошибка. Все методы класса Proxy находятся в пространстве имен `flash_proxy`, чтобы предотвратить конфликты имен.

Чтобы лучше понять, как используется пространство имен `flash_proxy`, необходимо разобраться с порядком использования класса Proxy. Функциональные возможности класса Proxy доступны только для классов, наследуемых из него. То есть если потребуется использовать методы класса Proxy к объекту, определение класса объекта должно расширять класс Proxy. Например, если требуется перехватить попытки вызова неопределенных методов, следует расширить класс Proxy, а затем переписать метод `callProperty()` в классе Proxy.

Необходимо помнить, что применение пространств имен обычно включает три этапа: определение, применение и привязку. Поскольку методы класса Proxy никогда не вызываются явным образом, то пространство имен `flash_proxy` только определяется и применяется, но никогда не привязывается. ActionScript 3.0 определяет пространство имен `flash_proxy` и применяет его в классе Proxy. Затем программному коду необходимо только применить пространство имен `flash_proxy` к классам, расширяющим класс Proxy.

Пространство имен `flash_proxy` определяется в пакете flash.utils способом, аналогичным следующему.

```
package flash.utils
{
    public namespace flash_proxy;
}
```

Пространство имен применяется к методам класса Proxy, как показано в следующей выборке из класса Proxy.

```
public class Proxy
{
    flash_proxy function callProperty(name:*, ... rest):*
    flash_proxy function deleteProperty(name:*):Boolean
    ...
}
```

Как показано в следующем программном коде, необходимо вначале импортировать как класс Proxy, так и пространство имен `flash_proxy`. Затем следует объявить свой класс, расширяющий класс Proxy (необходимо также добавить атрибут `dynamic`, если предполагается компиляция в строгом режиме). Когда метод `callProperty()` будет перезаписан, необходимо использовать пространство имен `flash_proxy`.

```
package
{
    import flash.utils.Proxy;
    import flash.utils.flash_proxy;

    dynamic class MyProxy extends Proxy
    {
        flash_proxy override function callProperty(name:*, ...rest):*
        {
            trace("method call intercepted: " + name);
        }
    }
}
```

Если создается экземпляр класса MyProxy и вызывается неопределенный метод, например метод `testing()`, см. следующий пример, то объект Proxy перехватывает вызов метода и выполняет инструкции, находящиеся в перезаписанном методе `callProperty()` (в данном случае простую инструкцию `trace()`).

```
var mySample:MyProxy = new MyProxy();
mySample.testing(); // method call intercepted: testing
```

Расположение методов класса Proxy в пространстве имен `flash_proxy` обеспечивает два преимущества. Во-первых, отдельное пространство имен уменьшает беспорядок интерфейса общего доступа для любого класса, расширяющего класс Proxy. (В классе Proxy имеется около двенадцати методов, которые можно перезаписать, все они не предназначены для прямого вызова. Размещение всех их в пространстве имен public может привести к сбоям.) Во-вторых, использование пространства имен `flash_proxy` предотвращает конфликты имен, в случае если подкласс Proxy содержит методы экземпляров с именами, совпадающими с какими-либо именами методов в классе Proxy. Например, может потребоваться назвать один из создаваемых методов `callProperty()`. Показанный далее программный код допустим, поскольку созданная версия метода `callProperty()` находится в другом пространстве имен.

```
dynamic class MyProxy extends Proxy
{
    public function callProperty() {}
    flash_proxy override function callProperty(name:*, ...rest):*
    {
        trace("method call intercepted: " + name);
    }
}
```

Пространства имен также могут быть полезны, если требуется обеспечить доступ к методам или свойствам таким способом, который нельзя организовать четырьмя спецификаторами контроля доступа (`public`, `private`, `internal` и `protected`). Например, может получиться, что несколько служебных методов распределены между несколькими пакетами. Требуется, чтобы эти методы были доступны для всех этих пакетов, но в то же время методы должны быть закрыты для общего доступа. Для решения этой задачи можно создать несколько пространств имен и использовать их в качестве собственных специальных спецификаторов контроля доступа.

В следующем примере определенное пользователем пространство имен используется для группировки двух функций, находящихся в разных пакетах. Группируя их в одном пространстве имен, можно сделать обе функции видимыми для класса и пакета, используя одну инструкцию `use namespace`.

В этом примере, чтобы продемонстрировать данную технологию, используется четыре файла. Все эти файлы должны находиться в подкаталоге классов. Первый файл, myInternal.as, используется для определения пространства имен `myInternal`. Поскольку этот файл находится в пакете с именем example, необходимо поместить файл в папку с именем example. Пространство имен помечено как `public`, поэтому его можно импортировать в другие пакеты.

```
// myInternal.as in folder example
package example
{
    public namespace myInternal = "http://www.adobe.com/2006/actionscript/examples";
}
```

Второй и третий файлы, Utility.as и Helper.as, определяют классы, которые содержат методы, доступные для других пакетов. Класс Utility находится в пакете example.alpha, что означает, что файл должен быть помещен в папку alpha, вложенную в папку example. Класс Helper находится в пакете example.beta, что означает, что файл должен быть помещен в папку beta, вложенную в папку example. Оба этих пакета, example.alpha и example.beta, должны импортировать пространство имен, прежде чем использовать его.

```
// Utility.as in the example/alpha folder
package example.alpha
{
    import example.myInternal;

    public class Utility
    {
        private static var _taskCounter:int = 0;

        public static function someTask()
        {
            _taskCounter++;
        }

        myInternal static function get taskCounter():int
        {
            return _taskCounter;
        }
    }
}

// Helper.as in the example/beta folder
package example.beta
{
    import example.myInternal;

    public class Helper
    {
        private static var _timeStamp:Date;

        public static function someTask()
        {
            _timeStamp = new Date();
        }

        myInternal static function get lastCalled():Date
        {
            return _timeStamp;
        }
    }
}
```

Четвертый файл, NamespaceUseCase.as, является основным классом приложения и должен располагаться на одном уровне с папкой example. Во Flash Professional этот класс будет использоваться в качестве класса документа для FLA. Класс NamespaceUseCase также импортирует пространство имен `myInternal` и использует его для вызова двух статических методов, которые находятся в других пакетах. Статические методы в этом примере использованы исключительно для упрощения программного кода. Оба метода, статический и создающий экземпляр, могут размещаться в пространстве имен `myInternal`.

```
// NamespaceUseCase.as
package
{
    import flash.display.MovieClip;
    import example.myInternal; // import namespace
    import example.alpha.Utility;// import Utility class
    import example.beta.Helper;// import Helper class

    public class NamespaceUseCase extends MovieClip
    {
        public function NamespaceUseCase()
        {
            use namespace myInternal;

            Utility.someTask();
            Utility.someTask();
            trace(Utility.taskCounter); // 2

            Helper.someTask();
            trace(Helper.lastCalled); // [time someTask() was last called]
        }
    }
}
```

Переменные

Переменные позволяют сохранять значения, используемые в программе. Чтобы объявить переменную, необходимо использовать инструкцию `var` с именем переменной. В ActionScript 3.0 использование инструкции `var` требуется всегда. Например, в следующей строке ActionScript объявляется переменная с именем `i`:

```
var i;
```

Если пропустить оператор `var` при объявлении переменной, появляется ошибка компилятора в строгом режиме и ошибка при выполнении в стандартном режиме. Например, в следующей строке программного кода возникает ошибка, если переменная `i` ранее не была определена:

```
i; // error if i was not previously defined
```

Для связи переменной с типом данных это необходимо при объявлении переменной. Объявление переменной без указания ее типа разрешено, но при этом создается предупреждение в строгом режиме компилятора. Тип переменной назначается добавлением к имени переменной двоеточия (:) с последующим указанием типа этой переменной. Например, в следующем программном коде объявляется переменная `i`, которая имеет тип int.

```
var i:int;
```

Значение переменной можно задать, используя оператор присваивания (=). Например, в следующем коде объявляется переменная `i` и ей назначается значение 20.

```
var i:int;
i = 20;
```

Возможно, покажется более удобным назначать значение переменной одновременно с ее объявлением, как показано в следующем примере.

```
var i:int = 20;
```

Технология назначения значения переменной во время ее объявления широко используется не только при назначении примитивных значений, таких как целочисленные значения и строки, но также при создании массивов или создании экземпляров классов. В следующем примере показано, как в одной строчке программного кода создается массив и ему присваивается значение.

```
var numArray:Array = ["zero", "one", "two"];
```

Можно создать экземпляр класса с помощью оператора `new`. В следующем примере создается экземпляр с именем `CustomClass` и выполняется привязка вновь созданного экземпляра класса переменной с именем `customItem`.

```
var customItem:CustomClass = new CustomClass();
```

Если объявляется больше одной переменной, можно объявить их все в одной строке кода, используя оператор запятой (`,`) для разделения переменных. Например, в следующем программном коде объявляется три переменных в одной строке кода.

```
var a:int, b:int, c:int;
```

Также можно назначить значения каждой из переменных в одной строке кода. Например, в следующем программном коде объявляются три переменные (`a`, `b` и `c`), и каждой из них назначается значение.

```
var a:int = 10, b:int = 20, c:int = 30;
```

Хотя можно использовать оператор запятой для группирования объявления переменных в одной инструкции, это может привести к снижению наглядности программного кода.

Знакомство с областью действия переменной

Областью действия переменной является область программного кода, где данная переменная доступна для лексических ссылок. Переменная *global* объявляется для всех областей программного кода, а переменная *local* — только для какой-то его части. В ActionScript 3.0 переменным всегда назначается область действия функции или класса, в которой они объявлены. Глобальная переменная — это переменная, определяемая вне определения какой-либо функции или класса. Например, в следующем программном коде глобальная переменная `strGlobal` создается путем обновления ее вне какой-либо функции. В этом примере показано, что глобальная переменная доступна как вне, так и внутри определения функции.

```
var strGlobal:String = "Global";
function scopeTest()
{
    trace(strGlobal); // Global
}
scopeTest();
trace(strGlobal); // Global
```

Локальная переменная объявляется путем объявления переменной внутри функции. Наименьшей областью кода, для которой можно определить локальную переменную, является определение функции. Локальная переменная, объявленная внутри функции, существует только в этой функции. Например, если объявляется переменная с именем `str2` в функции `localScope()`, то эта переменная не будет доступна вне данной функции.

```
function localScope()
{
    var strLocal:String = "local";
}
localScope();
trace(strLocal); // error because strLocal is not defined globally
```

Если используемое для локальной переменной имя уже объявлено для глобальной переменной, локальное определение скрывает (или затеняет) глобальное определение, пока локальная переменная находится в области действия. При этом глобальная переменная по-прежнему существует вне этой функции. Например, в следующем программном коде создается глобальная строковая переменная с именем `str1`, а затем создается локальная переменная с тем же именем в функции `scopeTest()`. Инструкция `trace` в функции выводит локальное значение этой переменной, а инструкция `trace` вне функции выводит глобальное значение переменной.

```
var str1:String = "Global";
function scopeTest ()
{
    var str1:String = "Local";
    trace(str1); // Local
}
scopeTest();
trace(str1); // Global
```

Переменные ActionScript в отличие от переменных в C++ и Java не имеют области действия уровня блока. Блоком программного кода является группа операторов между открывающейся фигурной скобкой ({) и закрывающейся фигурной скобкой (}). В некоторых языках программирования, таких как C++ и Java, переменные, объявленные в блоке программного кода, недоступны вне этого блока кода. Это ограничение области действия называется областью действия уровня блока и не существует в ActionScript. Если переменная объявляется внутри блока программного кода, эта переменная доступна не только в этом блоке кода, но также во всех других частях функции, которой принадлежит этот блок кода. Например, следующая функция содержит переменные, которые определены в различных областях действия уровня блока. Все переменные доступны во всех частях функции.

```
function blockTest (testArray:Array)
{
    var numElements:int = testArray.length;
    if (numElements > 0)
    {
        var elemStr:String = "Element #";
        for (var i:int = 0; i < numElements; i++)
        {
            var valueStr:String = i + ": " + testArray[i];
            trace(elemStr + valueStr);
        }
        trace(elemStr, valueStr, i); // all still defined
    }
    trace(elemStr, valueStr, i); // all defined if numElements > 0
}

blockTest(["Earth", "Moon", "Sun"]);
```

Интересным следствием отсутствия области действия уровня блока является возможность считывать и записывать переменные, которые еще не были объявлены, при условии, что они будут объявлены до конца функции. Это возможно благодаря технологии, называемой *подъем*, которая подразумевает перемещение всех объявлений переменных в верхнюю часть функции. Например, следующий программный код компилируется несмотря на то, что исходная функция `trace()` для переменной `num` появляется до того, как объявлена переменная `num`.

```
trace(num); // NaN
var num:Number = 10;
trace(num); // 10
```

Однако компилятор не будет выполнять подъем каких-либо инструкций присваивания. Это объясняет, почему выполнение исходной функции `trace()` для переменной `num` в результате дает `NaN` (нечисловое значение), которое является значением по умолчанию для переменных с типом данных Number. Это означает, что можно назначать значения для переменных даже до того, как они будут объявлены, как показано в следующем примере.

```
num = 5;
trace(num); // 5
var num:Number = 10;
trace(num); // 10
```

Значения по умолчанию

Значением по умолчанию является значение, которое содержит переменную перед тем, как ей будет установлено пользовательское значение. *Инициализация* переменной происходит при первой установке ее значения. Если объявляется переменная, но не устанавливается ее значение, эта переменная *не инициализирована*. Значение неинициализированной переменной зависит от ее типа данных. В следующей таблице описаны значения переменных по умолчанию, упорядоченные по типам данных.

Тип данных	**Значение по умолчанию**
Boolean	`false`
int	0
Число	`NaN`
Объект	`null`
String	`null`
uint	0
Не объявлено (эквивалентно аннотации типа `*`)	`undefined`
Все другие классы, включая определяемые пользователем классы.	`null`

Для переменных с типом Number значением по умолчанию является `NaN` (нечисловое значение), являющееся специальным значением, определенным стандартом IEEE-754 для передачи значений, не являющихся числами.

Если определяется переменная, но не указывается ее тип данных, типом данных, применяемым по умолчанию, является `*`, который фактически означает, что тип переменной не задан. Если также не инициализировать переменную без типа с помощью значения, значением по умолчанию для нее будет `undefined`.

Для типов данных, отличных от Boolean, Number, int и uint, значением по умолчанию для любой неинициализированной переменной является `null`. Это применимо ко всем классам, определяемым языком ActionScript 3.0, а также к любым создаваемым классам.

Неопределенное значение `null` не является действительным значением для переменных с типом Boolean, Number, int или uint. Если выполняется попытка назначить значение `null` такой переменной, это значение преобразуется в значение по умолчанию для этого типа данных. Переменным с типом Object можно назначать значение `null`. При попытке назначить значение `undefined` переменной с типом Object, значение преобразуется в `null`.

Для переменных с типом Number существует специальная функция с названием `isNaN()`, которая возвращает логическое значение `true`, если переменная не число, и `false` в ином случае.

Типы данных

Тип данных определяет набор значений. Например, тип данных Boolean является набором всего из двух значений: `true` и `false`. Помимо типа данных Boolean в ActionScript 3.0 определяется несколько более часто используемых типов данных, таких как строки (String), числовые значения (Number) и массивы (Array). Можно определить собственный тип данных, используя классы или интерфейсы для определения нестандартного набора значений. Все значения в ActionScript 3.0, независимо от того, примитивные они или сложные, являются объектами.

Примитивное значение — это значение, которое относится к одному из следующих типов данных: Boolean, int, Number, String и uint. Работать с примитивными значениями обычно получается быстрее, чем со сложными, поскольку ActionScript хранит примитивные значения специальным способом, делая возможным оптимизацию быстродействия и использования памяти.

Примечание. *Читателям, заинтересованным в технических подробностях, любопытно будет узнать, что внутренним форматом хранения примитивных значений в ActionScript являются постоянные объекты. Тот факт, что они хранятся как постоянные объекты, означает, что вместо передачи самих значений можно передавать ссылки на них. Это сократит потребление памяти и повысит скорость выполнения, поскольку ссылки обычно значительно меньше, чем сами значения.*

Сложное значение — это альтернатива примитивным значениям. К типам данных, определяющих наборы сложных значений, относятся Array, Date, Error, Function, RegExp, XML и XMLList.

Во многих языках программирования существуют различия между примитивными значениями и интерфейсными объектами. Например, в Java имеется примитивное значение int и класс java.lang.Integer, который служит интерфейсом к нему. Примитивы Java не являются объектами, но таковы их интерфейсные объекты, с помощью которых примитивы становятся пригодны для некоторых операций, а интерфейсные объекты лучше подходят для других операций. В ActionScript 3.0 в практических целях простые значения и их интерфейсные объекты неразличимы. Все значения, даже простые значения, являются объектами. Среда выполнения обрабатывает эти простые типы как особые случаи, поскольку они ведут себя как объекты, но при этом не требуют обычной нагрузки, связанной с созданием объектов. Это значит, что следующие две строки программного кода эквивалентны.

```
var someInt:int = 3;
var someInt:int = new int(3);
```

Все примитивные и сложные типы данных, перечисленные выше, определяются классами ядра ActionScript 3.0. Классы ядра позволяют создавать объекты с помощью литеральных значений вместо использования оператора `new`. Например, можно создать массив с помощью литерального значения или конструктора класса Array, как показано в следующем примере.

```
var someArray:Array = [1, 2, 3]; // literal value
var someArray:Array = new Array(1,2,3); // Array constructor
```

Проверка типа

Проверка типа может производиться либо во время компиляции, либо во время выполнения. В языках статичного типа, таких как C++ и Java, проверка типа выполняется при компиляции. В языках динамического типа, таких как Smalltalk и Python, проверка типа возможна во время выполнения. Являясь языком динамического типа, ActionScript 3.0 позволяет проверку типа во время выполнения, но также поддерживает проверку типа при компиляции благодаря специальному режиму компиляции, называемому *строгим режимом*. В строгом режиме проверка типа происходит как во время компиляции, так и во время выполнения, а в стандартном режиме проверка возможна только во время выполнения.

Языки динамического типа предлагают значительную гибкость при структурировании программного кода, но иногда из-за этого происходят ошибки типов, которые обнаруживаются во время выполнения. В языках статического типа ошибки типов обнаруживаются во время компиляции, но для этого необходимо, чтобы информация о типе была известна во время компиляции.

Проверка типа во время компиляции

Проверка типа во время компиляции часто более предпочтительна в крупных проектах, поскольку по мере роста проекта гибкость типов данных обычно становится менее важной, чем необходимость отслеживания ошибок на максимально ранних стадиях. Вот почему по умолчанию компилятор ActionScript в инструментах Flash Professional и Flash Builder установлен для работы в строгом режиме.

Adobe Flash Builder

Строгий режим во Flash Builder можно отключить с помощью настроек компилятора ActionScript в диалоговом окне «Свойства проекта».

Чтобы обеспечить возможность проверки во время компиляции, компилятору необходимо располагать сведениями о типах данных для переменных и выражений в программном коде. Чтобы явным образом объявить тип данных для переменной, добавьте оператор двоеточия (`:`) с последующим указанием типа данных в качестве суффикса к имени переменной. Чтобы связать тип данных с параметром, используется оператор двоеточия с последующим указанием типа данных. Например, в следующем программном коде к параметру `xParam` добавляются сведения о типе данных, а затем объявляется переменная `myParam` с явно указанным типом данных.

```
function runtimeTest(xParam:String)
{
    trace(xParam);
}
var myParam:String = "hello";
runtimeTest(myParam);
```

В строгом режиме работы компилятор ActionScript сообщает о несоответствиях типов как об ошибках компиляции. Например, в следующем программном коде объявляется параметр функции `xParam` с типом Object, но затем предпринимается попытка назначить этому параметру значения типа String и Number. При этом в строгом режиме работы компилятора выдаются ошибки.

```
function dynamicTest(xParam:Object)
{
    if (xParam is String)
    {
        var myStr:String = xParam; // compiler error in strict mode
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam; // compiler error in strict mode
        trace("Number: " + myNum);
    }
}
```

Однако даже в строгом режиме можно селективно отключать проверку типа во время компиляции, оставляя незаполненной правую часть инструкции присваивания. Можно пометить переменную или выражение как не имеющие типа, либо пропустив аннотацию типа, либо используя специальную аннотацию типа с помощью звездочки (*). Например, если параметр xParam в предыдущем примере изменяется так, что больше не имеет аннотации типа, программный код компилируется в строгом режиме:

```
function dynamicTest(xParam)
{
    if (xParam is String)
    {
        var myStr:String = xParam;
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam;
        trace("Number: " + myNum);
    }
}
dynamicTest(100)
dynamicTest("one hundred");
```

Проверка типа при выполнении

Проверки типа при выполнении в ActionScript 3.0 происходит независимо от того, идет работа в строгом или в стандартном режиме. Рассмотрите ситуацию, в которой значение 3 передается в качестве аргумента функции, ожидающей массив. В строгом режиме компилятор создаст ошибку, поскольку значение 3 не совместимо с типом данных Array. Если отключить строгий режим и выполнять программу в стандартном режиме, компилятор пропустит несоответствие типов, но при проверке типа во время выполнения возникнет ошибка выполнения.

В следующем примере показана функция с именем typeTest(), которая ожидает аргумент Array, но ей передается значение 3. Это вызывает ошибку выполнения в стандартном режиме, поскольку значение 3 не является членом объявленного для параметра типа данных (Array).

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum:Number = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0 standard mode
```

Также возможны ситуации, когда ошибки выполнения возникают при работе в строгом режиме. Это возможно в тех случаях, если используется строгий режим, но выключена проверка типа при компиляции при использовании переменных без указания типа. Если используются переменные без указания типа, проверка типа не отключается полностью, а просто откладывается до выполнения программного кода. Например, если для переменной myNum в предыдущем примере не объявлен тип данных, компилятор не сможет определить несоответствие типов, но при выполнении кода возникнет ошибка выполнения, поскольку сравнивается значение при выполнении переменной myNum, установленное с помощью команды присваивания и равное 3, с типом xParam, для которого задан тип данных Array.

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0
```

Проверка типа при выполнении позволяет более гибко использовать наследование, чем при проверке во время компиляции. Откладывая проверку типа до времени выполнения, стандартный режим позволяет ссылаться на свойства подкласса даже в том случае, если выполняется *восходящее преобразование*. Восходящее преобразование происходит, если для объявления типа экземпляра класса используется базовый класс, а для его инициации используется подкласс. Например, можно создать класс с именем ClassBase, который может быть расширен (классы с атрибутом `final` не могут быть расширены).

```
class ClassBase
{
}
```

Затем можно создать подкласс ClassBase с именем ClassExtender, у которого есть одно свойство с именем `someString`, как показано в следующем примере.

```
class ClassExtender extends ClassBase
{
    var someString:String;
}
```

Используя оба класса, можно создать экземпляр класса, объявляемый с типом данных ClassBase, но инициируемый с помощью конструктора ClassExtender. Восходящее преобразование считается безопасной операцией, поскольку базовый класс не содержит каких-либо свойств или методов, которых не было бы в подклассе.

```
var myClass:ClassBase = new ClassExtender();
```

В то же время подкласс содержит свойства или методы, которые отсутствуют в базовом классе. Например, класс ClassExtender содержит свойство `someString`, не существующее в классе ClassBase. В стандартном режиме работы ActionScript 3.0 можно ссылаться на это свойство с помощью экземпляра `myClass`, не создавая ошибки во время компиляции, как показано в следующем примере.

```
var myClass:ClassBase = new ClassExtender();
myClass.someString = "hello";
// no error in ActionScript 3.0 standard mode
```

Оператор is

Оператор `is` позволяет проверить, является ли переменная или выражение членом определенного типа данных. В предыдущих версиях ActionScript оператор `instanceof` обеспечивал данную функциональную возможность, но в ActionScript 3.0 оператор `instanceof` не следует использовать для проверки принадлежности к типу данных. Оператор `is` необходимо использовать вместо оператора `instanceof` для проверки типа вручную, поскольку выражение `x instanceof y` проверяет только цепочку прототипа `x` на существование `y` (а в ActionScript 3.0 цепочка прототипа не передает полной картины иерархии наследования).

Оператор `is` проверяет правильность иерархии наследования и может применяться для проверки не только того, является ли объект экземпляром определенного класса, но также является ли объект экземпляром класса, реализующего определенный интерфейс. В следующем примере создается экземпляр класса Sprite с именем `mySprite`, который использует оператор `is` для проверки того, является ли `mySprite` экземпляром классов Sprite и DisplayObject, а также реализует ли он интерфейс IEventDispatcher.

```
var mySprite:Sprite = new Sprite();
trace(mySprite is Sprite); // true
trace(mySprite is DisplayObject);// true
trace(mySprite is IEventDispatcher); // true
```

Оператор `is` проверяет иерархию наследования и правильно диагностирует, что `mySprite` совместим с классами Sprite и DisplayObject (класс Sprite является подклассом класса DisplayObject). Оператор `is` также проверяет, действительно ли `mySprite` наследует от какого-либо класса, в котором реализован интерфейс IEventDispatcher. Поскольку класс Sprite наследует от класса EventDispatcher, в котором реализован интерфейс IEventDispatcher, оператор `is` правильно сообщает, что в классе `mySprite` также реализован этот интерфейс.

В следующем примере показаны те же тесты из предыдущего примера, но с оператором `instanceof` вместо оператора `is`. Оператор `instanceof` правильно определяет, что класс `mySprite` является экземпляром классов Sprite или DisplayObject, но он возвращает результат `false` при попытке проверить, реализует ли класс `mySprite` интерфейс IEventDispatcher.

```
trace(mySprite instanceof Sprite); // true
trace(mySprite instanceof DisplayObject);// true
trace(mySprite instanceof IEventDispatcher); // false
```

Оператор as

Оператор `as` также позволяет проверить, является ли переменная или выражение членом определенного типа данных. В отличие от оператора `is` оператор `as` не возвращает логические значения. При этом оператор `as` возвращает значение выражения вместо `true` или значение `null` вместо `false`. В следующем примере показан результат использования оператора `as` вместо оператора `is` в простом случае проверки. Определяется, действительно ли экземпляр Sprite является членом типов данных DisplayObject, IEventDispatcher и Number.

```
var mySprite:Sprite = new Sprite();
trace(mySprite as Sprite); // [object Sprite]
trace(mySprite as DisplayObject); // [object Sprite]
trace(mySprite as IEventDispatcher); // [object Sprite]
trace(mySprite as Number);          // null
```

При использовании оператора `as` операнд справа должен быть типом данных. Попытка использовать в качестве операнда справа выражение вместо типа данных приведет к ошибке.

Динамические классы

Динамический класс определяет объект, который может быть изменен во время выполнения путем добавления или изменения свойств и методов. Нединамические классы, такие как класс String, являются *запечатанными* классами. К запечатанным классам нельзя добавлять свойства или методы во время выполнения.

Динамический класс создается с помощью атрибута `dynamic` при объявлении класса. Например, в следующем программном коде создается динамический класс с именем `Protean`.

```
dynamic class Protean
{
    private var privateGreeting:String = "hi";
    public var publicGreeting:String = "hello";
    function Protean()
    {
        trace("Protean instance created");
    }
}
```

Если впоследствии создается экземпляр класса `Protean`, можно добавлять свойства и методы к нему за пределами определения класса. Например, в следующем программном коде создается экземпляр класса `Protean` и к нему добавляется свойство с именем `aString`, а также свойство с именем `aNumber`.

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
trace(myProtean.aString, myProtean.aNumber); // testing 3
```

Свойства, добавленные к экземпляру динамического класса, являются объектами выполнения, поэтому любые проверки типов выполняются во время выполнения. Нельзя добавить аннотацию типа к свойству, добавленному таким образом.

Также к экземпляру `myProtean` можно добавить метод, определив функцию и прикрепив ее к свойству экземпляра `myProtean`. В следующем программном коде инструкция трассировки перемещается в метод с именем `traceProtean()`.

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
myProtean.traceProtean = function ()
{
    trace(this.aString, this.aNumber);
};
myProtean.traceProtean(); // testing 3
```

Методы, созданные таким способом, не имеют доступа к любым частным свойствам или методам класса Protean. Более того, даже ссылки на общие свойства или методы класса `Protean` должны уточняться с помощью ключевого слова `this` или имени класса. В следующем примере показан метод `traceProtean()`, пытающийся обращаться к частным или общим переменным класса `Protean`.

```
myProtean.traceProtean = function ()
{
    trace(myProtean.privateGreeting); // undefined
    trace(myProtean.publicGreeting); // hello
};
myProtean.traceProtean();
```

Описания типов данных

К примитивным типам данных относятся Boolean, int, Null, Number, String, uint и void. Классы ядра ActionScript также определяют следующие сложные типы данных: Object, Array, Date, Error, Function, RegExp, XML и XMLList.

Логический тип данных (Boolean)

Логический тип данных (Boolean) включает два значения: `true` и `false`. Никакие другие значения для переменных с типом Boolean не действительны. По умолчанию для переменной с типом Boolean, которая объявлена, но не инициализирована, устанавливается значение `false`.

Тип данных int

Тип данных int предназначен для внутреннего хранения информации в виде 32-разрядных целых чисел и включает подмножество целых чисел от

-2147483648 (-2^{31}) до 2147483647 ($2^{31} - 1$) включительно. В предыдущих версиях ActionScript предлагался только тип данных Number, в котором использовались как целые числа, так и числа с плавающей точкой. В ActionScript 3.0 теперь есть доступ к двоичным типам нижнего уровня для работы с 32-разрядными целыми числами со знаком и без знака. Если переменная не использует числа с плавающей точкой, то применение типа данных int вместо типа данных Number позволит ускорить обработку данных и сделать ее более эффективной.

Для целых значений, выходящих за пределы диапазона минимального и максимального значений, используйте тип данных Number, с помощью которого можно обрабатывать значения в промежутке между положительным и отрицательным значением 9007199254740992 (53-разрядные целые значения). По умолчанию значение переменной с типом данных int равно 0.

Тип данных Null

Тип данных Null содержит только одно значение `null`. Это значение по умолчанию для типа данных String и всех классов, которые определяют сложные типы данных, включая класс Object. Никакой другой из примитивных типов данных, таких как Boolean, Number, int и uint, не может содержать значение `null`. Во время выполнения значение `null` преобразуется в соответствующее значение по умолчанию при попытке назначить значение `null` переменным типа Boolean, Number, int или uint. Нельзя использовать этот тип данных в качестве аннотации типа.

Тип данных Number

В ActionScript 3.0 тип данных Number может быть представлен целыми числами, целыми числами без знака, а также числами с плавающей точкой. В то же время, чтобы повысить производительность, необходимо использовать тип данных Number только для целых значений, которые не помещаются в 32-разрядных типах `int` и `uint`, или для значений с плавающей точкой. Для сохранения числа с плавающей точкой включите в него десятичную точку. Если десятичная точка пропущена, число сохраняется как целое.

В типе данных Number используется 64-разрядный формат с двойной точностью в соответствии со стандартом IEEE для двоичных арифметических операций с плавающей точкой (IEEE-754). Этот стандарт определяет, как числа с плавающей точкой сохраняются с помощью 64 доступных разрядов. Один разряд используется для обозначения того, положительным или отрицательным является число. Одиннадцать разрядов используются для экспоненциальной части, которая сохраняется с основанием 2. Оставшиеся 52 разряда используются для хранения *значащей части числа* (также называемой *мантиссой*), которая является числом, возводимым в степень, указанную в экспоненциальной части.

Используя некоторые из этих разрядов для хранения экспоненциальной части, тип данных Number может сохранять числа с плавающей точкой, значительно превышающие значения, которые бы получались, если бы все разряды были отведены под значащую часть числа. Например, если бы тип данных Number использовал все свои 64 разряда для сохранения значащей части числа, в нем можно было бы сохранить значение $2^{65} - 1$. Используя 11 разрядов для хранения экспоненциальной части, тип данных Number может возводить свою значимую часть в степень 2^{1023}.

Максимальное и минимальное значения, которые может воспроизводить класс Number, хранятся в статических свойствах класса Number, называемых `Number.MAX_VALUE` и `Number.MIN_VALUE`.

```
Number.MAX_VALUE == 1.79769313486231e+308
Number.MIN_VALUE == 4.940656458412467e-324
```

Расплатой за возможность использовать столь огромный диапазон значений является точность. Тип данных Number использует 52 разряда для хранения значимой части числа, в результате числа, которым требуется больше 52 разрядов для точного воспроизведения, например 1/3, отображаются как приближенные значения. Если для приложения требуется абсолютная точность в работе с десятичными числами, необходимо использовать программное обеспечение, реализующее десятичные арифметические операции с плавающей точкой в качестве альтернативы двоичным операциям.

При сохранении целых значений с помощью типа данных Number для значимой части числа используется только 52 разряда. Тип данных Number использует эти 52 разряда и специальный скрытый разряд для представления целых чисел в диапазоне от -9007199254740992 (-2^{53}) до 9007199254740992 (2^{53}).

Значение `NaN` используется не только как значение по умолчанию для переменных с типом `Number`, но также в качестве результата любой операции, которая должна возвращать число, но не делает этого. Например, при попытке рассчитать квадратный корень отрицательного числа результатом будет `NaN`. Другими специальным значениями типа Number являются *плюс бесконечность* и *минус бесконечность* .

Примечание. *Результатом деления на `0` является только `NaN`, если делитель тоже `0`. В результате деления на `0` получается `infinity`, если делимое положительно, или `-infinity`, если делимое отрицательно.*

Тип данных String

Тип данных String передает последовательность 16-разрядных символов. Для внутреннего хранения значений с типом String используются символы Unicode и формат UTF-16. Значения с типом данных String являются постоянными, такими же, как и при программировании в языке Java. Операция со значением String возвращает новый экземпляр этой строки. По умолчанию значением для переменной, объявленной с типом данных String, является `null`. Значение `null` не равнозначно пустой строке (`""`). Значение `null` обозначает, что в переменной не сохранено значение, а пустая строка обозначает, что в переменной сохранено значение строкового типа String, не содержащее символов.

Тип данных uint

Для внутреннего хранения типа данных uint используются 32-разрядные целые числа без знака. Этот тип данных содержит диапазон целых значений от 0 до 4294967295 (2^{32} - 1) включительно. Тип данных uint применяется в особых случаях, когда необходимы положительные целые числа. Например, следует использовать тип данных uint для передачи значений цвета пиксела, поскольку тип данных int имеет внутренний разряд для знака, не удобный для обработки цветовых значений. Для целых значений, больших, чем максимальное значение uint, следует использовать тип данных Number, который позволяет обрабатывать 53-разрядные целые значения. По умолчанию значение переменной с типом данных uint равно 0.

Тип данных void

Тип данных void содержит только одно значение `undefined`. В предыдущих версиях ActionScript значение `undefined` было значением по умолчанию для экземпляров класса Object. В ActionScript 3.0 значением по умолчанию для экземпляров Object является `null`. При попытке назначить значение `undefined` экземпляру объекта Object значение преобразуется в `null`. Переменной, для которой не задан тип, можно назначить только значение `undefined`. Переменные без указания типа — это переменные, для которых либо отсутствует любая аннотация типа, либо указан символ звездочки (`*`) в качестве аннотации типа. Значение `void` можно использовать только для аннотации возвращаемого типа.

Тип данных Object

Тип данных Object определяется классом Object. Класс Object служит базовым классом для всех определений классов в ActionScript. В ActionScript 3.0 версия типа данных Object отличается от предыдущих версий тремя особенностями. Во-первых, тип данных Object больше не является типом данных по умолчанию, назначаемым переменным, для которых не задан тип. Во-вторых, тип данных Object больше не включает значение `undefined`, которое используется в качестве значения по умолчанию для экземпляров Object. В-третьих, в ActionScript 3.0 значением по умолчанию для экземпляров класса Object является `null`.

В предыдущих версиях ActionScript переменной без аннотации типа автоматически назначался тип Object. В ActionScript 3.0 это не так, здесь реализована концепция переменной, действительно не имеющей типа. Переменные без аннотации типа теперь расцениваются как переменные, не имеющие типа. Если требуется, чтобы читателям программного кода были ясны намерения автора оставить переменную без типа, можно использовать символ звездочки (`*`) для аннотации типа, что будет эквивалентно пропущенной аннотации типа. В следующем примере показаны две эквивалентные инструкции, обе они объявляют переменную `x`, не имеющую типа.

```
var x
var x:*
```

Только переменная, не имеющая типа, может содержать значение `undefined`. Если попытаться назначить значение `undefined` переменной, имеющей тип данных, среда выполнения преобразует значение `undefined` в значение по умолчанию этого типа данных. Для экземпляров типа данных Object значением по умолчанию является `null`, а это значит, что при попытке присвоить значение `undefined` экземпляру Object значение преобразуется в `null`.

Преобразования типа

Преобразование типа происходит, если значение превращается из одного типа данных в другой. Преобразование типа может быть либо *явным*, либо *подразумеваемым*. Иногда во время выполнения выполняется скрытое преобразование, которое также называется *приведением типа данных*. Например, если значение 2 назначено переменной с типом данных Boolean, значение 2 преобразуется в значение `true` типа данных Boolean перед его назначением переменной. Явное преобразование, которое также называется *явным приведением*, происходит в тех случаях, когда программный код запрашивает, чтобы компилятор переменную одного типа данных обработал как переменную другого типа. Если используются примитивные значения, явное приведение преобразует значения из одного типа данных в другой. Для явного приведения объекта к другому типу имя этого объекта заключают в круглые скобки и предваряют именем нового типа. Например, в следующем программном коде берется логическое значение и явным образом приводится к целому числу.

```
var myBoolean:Boolean = true;
var myINT:int = int(myBoolean);
trace(myINT); // 1
```

Неявные преобразования

Неявные преобразования происходят при выполнении программ в ряде следующих случаев.

- В инструкциях присваивания
- Если значения передаются как аргументы функции
- Если значения возвращаются из функции
- В выражениях, использующих определенные операторы, например оператор добавления (+)

Для определенных пользователем типов неявные преобразования выполняются успешно, если преобразуемое значение является экземпляром целевого класса или класса, производного от целевого. Если неявное преобразование завершается неудачно, происходит ошибка. Например, в следующем программном коде происходит успешное неявное преобразование и неудачное неявное преобразование.

```
class A {}
class B extends A {}

var objA:A = new A();
var objB:B = new B();
var arr:Array = new Array();

objA = objB; // Conversion succeeds.
objB = arr; // Conversion fails.
```

Для примитивных типов неявные преобразования обрабатываются путем вызова тех же внутренних алгоритмов преобразования, которые вызываются функциями явного преобразования.

Явные преобразования

Очень полезно использовать явные преобразования, или явные приведения, если компиляция происходит в строгом режиме, поскольку возможны случаи, когда требуется, чтобы при несовпадении типов не создавались ошибки во время компиляции. Такая ситуация может возникать, когда известно, что при неявном приведении значения будут правильно преобразованы во время выполнения. Например, при обработке данных, полученных из формы, можно возложить на явное приведение преобразования определенных строковых значений в числовые значения. В следующем программном коде создается ошибка во время компиляции даже несмотря на то, что этот код будет выполняться верно в стандартном режиме.

```
var quantityField:String = "3";
var quantity:int = quantityField; // compile time error in strict mode
```

Если требуется продолжить работу в строгом режиме, но необходимо преобразовывать строки в целые значения, можно использовать явное преобразование, делая это следующим образом.

```
var quantityField:String = "3";
var quantity:int = int(quantityField); // Explicit conversion succeeds.
```

Явное приведение в int, uint и Number

Можно выполнять явное приведение любого типа данных в один из трех числовых типов: int, uint и Number. Если преобразование числа невозможно по ряду причин, значение по умолчанию, равное 0, назначается типам данных int и uint, а также типу данных Number назначается значение по умолчанию `NaN`. Если происходит преобразование логического значения в числовое, то `true` становится значением 1, а `false` значением 0.

```
var myBoolean:Boolean = true;
var myUINT:uint = uint(myBoolean);
var myINT:int = int(myBoolean);
var myNum:Number = Number(myBoolean);
trace(myUINT, myINT, myNum); // 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // 0 0 0
```

Строковые значения, содержащие только числа, могут быть успешно преобразованы в один из цифровых типов. В числовые типы также можно преобразовывать строки, которые похожи на отрицательные значения, или строки, отображающие шестнадцатеричные значения (например, `0x1A`). В процессе преобразования отбрасываются предшествующий или завершающий символы пробелов в строковых значениях. Также можно выполнить явное приведение строк, которые напоминают числа с плавающей точкой, используя тип данных `Number()`. Включение десятичной точки заставляет значения с типом `uint()` и `int()` возвращать целое значение, отбрасывая десятичную часть и следующие за ней символы. Например, следующие строковые значения могут быть явным образом приведены в числа.

```
trace(uint("5")); // 5
trace(uint("-5")); // 4294967291. It wraps around from MAX_VALUE
trace(uint(" 27 ")); // 27
trace(uint("3.7")); // 3
trace(int("3.7")); // 3
trace(int("0x1A")); // 26
trace(Number("3.7")); // 3.7
```

При сохранении значений, содержащих нечисловые символы, возвращается 0, если выполнялось явное приведение в `int()` или `uint()`, а также `NaN`, если выполнялось явное приведение в `Number()`. В процессе преобразования предшествующие или завершающие пробелы отбрасываются, но возвращается 0 или `NaN`, если строка содержит пробел, разделяющий два числа.

```
trace(uint("5a")); // 0
trace(uint("ten")); // 0
trace(uint("17 63")); // 0
```

В ActionScript 3.0 функция `Number()` больше не поддерживает восьмеричные (имеющие основание 8) числа. Если передается строка, начинающаяся с 0 в функцию `Number()` ActionScript 2.0, это число интерпретируется как восьмеричное и преобразуется в свой десятеричный эквивалент. Это изменилось в функции `Number()` в ActionScript 3.0, которая просто отбрасывает первый ноль. Например, в следующем программном коде создаются разные выходные данные при компиляции в разных версиях ActionScript.

```
trace(Number("044"));
// ActionScript 3.0 44
// ActionScript 2.0 36
```

В явном приведении нет необходимости, если значение одного числового типа назначается переменной другого числового типа. Даже в строгом режиме числовые типы неявным образом преобразуются в другие числовые типы. Это означает, что в некоторых случаях, когда нарушаются диапазоны значений типа, могут выдаваться неожиданные значения. В следующих примерах все компиляции выполняются в строгом режиме, но в некоторых случаях все равно появляются непредвиденные значения.

```
var myUInt:uint = -3; // Assign int/Number value to uint variable
trace(myUInt); // 4294967293

var myNum:Number = sampleUINT; // Assign int/uint value to Number variable
trace(myNum) // 4294967293

var myInt:int = uint.MAX_VALUE + 1; // Assign Number value to uint variable
trace(myInt); // 0

myInt = int.MAX_VALUE + 1; // Assign uint/Number value to int variable
trace(myInt); // -2147483648
```

В следующей таблице резюмированы результаты явного приведения в типы Number, int или uint из других типов данных.

Тип данных или значение	Результат преобразования в Number, int или uint
Boolean	Если значение равно `true`, преобразуется в 1; иначе — в 0.
Date	Внутреннее представление объекта Date, показывающее число миллисекунд, которые прошли начиная с полуночи 1 января 1970 года, универсальное время.
`null`	0
Объект	Если значение экземпляра - `null`, и он преобразуется в Number, получается значение `NaN`; иначе получается значение 0.
String	Число, если строку можно преобразовать в число; в противном случае значение `NaN` при преобразовании в Number или 0 при преобразовании в int или uint.
`undefined`	Если преобразуется в Number, получается `NaN`; если преобразуется в int или uint — 0.

Явное приведение в Boolean

Явное приведение в Boolean из любого числового типа данных (uint, int и Number) дает значение `false`, если числовое значение равно 0, и `true` в остальных случаях. Для типа данных Number значение `NaN` также преобразуется в `false`. В следующем примере показаны результаты явного приведения чисел -1, 0 и 1.

```
var myNum:Number;
for (myNum = -1; myNum<2; myNum++)
{
    trace("Boolean(" + myNum +") is " + Boolean(myNum));
}
```

В результирующих данных примера показано, что из трех чисел только для 0 было возвращено значение `false`.

```
Boolean(-1) is true
Boolean(0) is false
Boolean(1) is true
```

Явное приведение в Boolean из значения String возвращает `false`, если значение строки `null` или это пустая строка (`""`). Во всех остальных случаях возвращается значение `true`.

```
var str1:String; // Uninitialized string is null.
trace(Boolean(str1)); // false

var str2:String = ""; // empty string
trace(Boolean(str2)); // false

var str3:String = " "; // white space only
trace(Boolean(str3)); // true
```

Явное приведение в Boolean экземпляра класса Object возвращает `false`, если значение экземпляра `null`; в остальных случаях возвращается значение `true`:

```
var myObj:Object; // Uninitialized object is null.
trace(Boolean(myObj)); // false

myObj = new Object(); // instantiate
trace(Boolean(myObj)); // true
```

Для переменных с типом Boolean выполняется специальная обработка в строгом режиме, при которой этим переменным можно назначить значения с любым типом данных, не проводя явного приведения. Неявное приведение из всех типов данных в тип данных Boolean происходит даже в строгом режиме. Говоря другими словами, в отличие практически от всех других типов данных, явное приведение в Boolean никогда не требуется, чтобы избежать ошибок в строгом режиме. В следующих примерах все компиляции выполняются в скрытом режиме и ведут себя так, как предполагается во время выполнения.

```
var myObj:Object = new Object(); // instantiate
var bool:Boolean = myObj;
trace(bool); // true
bool = "random string";
trace(bool); // true
bool = new Array();
trace(bool); // true
bool = NaN;
trace(bool); // false
```

В следующей таблице резюмированы результаты явного приведения в типы Boolean из других типов данных.

Тип данных или значение	Результат преобразования в Boolean
String	`false`, если значение `null` или пустая строка (`""`); `true` во всех остальных случаях.
`null`	`false`
Number, int или uint	`false`, если значение `NaN` или 0; `true` во всех остальных случаях.
Объект	`false`, если значение экземпляра `null`; `true` во всех остальных случаях.

Явное приведение в String

Явное приведение к типу данных String из любого числового типа данных возвращает строку, отображающую данное число. Явное приведение к типу данных String логического значения (Boolean) возвращает строку `"true"`, если значение `true`; возвращает строку `"false"`, если значение `false`.

Явное приведение типа данных String из значения экземпляра класса Object возвращает строку `"null"`, если значение этого экземпляра `null`. В остальных случаях явное приведение типа String из значения класса Object возвращает строку `"[object Object]"`.

Явное приведение в String из экземпляра класса Array возвращает строку, состоящую из разделенного запятыми списка всех элементов этого массива. Например, в следующем явном приведении к типу данных String возвращается одна строка, содержащая все три элемента массива.

```
var myArray:Array = ["primary", "secondary", "tertiary"];
trace(String(myArray)); // primary,secondary,tertiary
```

Явное приведение к типу String из экземпляра класса Date возвращает строчное представление даты, содержащейся в данном экземпляре. Например, в следующем примере возвращается строчное представление экземпляра класса Date (на выходе показан результат для тихоокеанского летнего времени).

```
var myDate:Date = new Date(2005,6,1);
trace(String(myDate)); // Fri Jul 1 00:00:00 GMT-0700 2005
```

В следующей таблице резюмированы результаты явного приведения в типы String из других типов данных.

Тип данных или значение	Результат преобразования в String
Массив	Строка состоит из всех элементов массива.
Boolean	`true` или `false`
Date	Строка, отображающая объект Date.
`null`	`"null"`
Number, int или uint	Строчное представление данного числа.
Объект	Если значение экземпляра null, возвращает `"null"`; во всех остальных случаях `"[object Object]"`.

Синтаксис

Синтаксис языка определяет набор правил, которые следует соблюдать при написании исполняемого программного кода.

Чувствительность к регистру

Язык ActionScript 3.0 чувствителен к регистру. Идентификаторы, различающиеся только значением регистра, считаются разными идентификаторами. Например, в следующем программном коде создаются две разные переменные.

```
var num1:int;
var Num1:int;
```

Синтаксис с точкой

Оператор точка (.) обеспечивает способ доступа к свойствам и методам объекта. Используя синтаксис с точкой, можно обращаться к свойствам или методам класса по имени экземпляра, указывая за оператором точка имя требуемого свойства или метода. Например, рассмотрим следующее определение класса.

```
class DotExample
{
    public var prop1:String;
    public function method1():void {}
}
```

Используя синтаксис с точкой, можно обратиться к свойству `prop1` и методу `method1()`, применяя имя экземпляра, созданного в следующем программном коде.

```
var myDotEx:DotExample = new DotExample();
myDotEx.prop1 = "hello";
myDotEx.method1();
```

Можно использовать синтаксис с точкой при определении пакетов. Оператор точка используется для обращения ко вложенным пакетам. Например, класс EventDispatcher находится в пакете с именем events, который вложен в пакет с именем flash. Можно обратиться к пакету events, используя следующее выражение.

```
flash.events
```

Также к классу EventDispatcher можно обратиться, используя следующее выражение.

```
flash.events.EventDispatcher
```

Синтаксис с косой чертой

Синтаксис с косой чертой не поддерживается в ActionScript 3.0. Синтаксис с косой чертой использовался в предыдущих версиях ActionScript для указания пути к фрагменту ролика или переменной.

Литералы

Литерал — это значение, которое появляется непосредственно в программном коде. Все следующие примеры являются литералами.

```
17
"hello"
-3
9.4
null
undefined
true
false
```

Литералы также могут группироваться, образуя составные литералы. Литералы массивов обрамляются квадратными скобками (`[]`) и используют запятые для разделения элементов массива.

Литерал массива можно использовать для инициализации массива. В следующем примере показаны два массива, инициализируемые с помощью литералов. Можно использовать оператор `new` и передать составной литерал в качестве параметра конструктору класса Array, но также можно назначить литеральные значения непосредственно при инициализации экземпляров следующих основных классов ActionScript: Object, Array, String, Number, int, uint, XML, XMLList и Boolean.

```
// Use new statement.
var myStrings:Array = new Array(["alpha", "beta", "gamma"]);
var myNums:Array = new Array([1,2,3,5,8]);

// Assign literal directly.
var myStrings:Array = ["alpha", "beta", "gamma"];
var myNums:Array = [1,2,3,5,8];
```

Литералы также можно использовать для инициализации родового объекта. Родовой объект — это экземпляр класса Object. Литералы объекта заключаются в фигурные скобки ({}), а для разделения свойств объекта используется запятая. Каждое свойство объявляется с помощью символа двоеточия (`:`), который отделяет имя свойства от его значения.

Можно создать родовой объект, используя инструкцию `new`, а затем передать литерал объекта в качестве параметра конструктору класса Object, также можно назначить литерал объекта напрямую объявляемому экземпляру. В следующем примере продемонстрированы два альтернативных способа создания нового родового объекта и инициализации этого объекта с тремя свойствами (`propA`, `propB` и `propC`), каждое с установленным значением 1, 2 и 3 соответственно.

```
// Use new statement and add properties.
var myObject:Object = new Object();
myObject.propA = 1;
myObject.propB = 2;
myObject.propC = 3;

// Assign literal directly.
var myObject:Object = {propA:1, propB:2, propC:3};
```

Дополнительные разделы справки

Работа со строками

Использование регулярных выражений

Инициализация переменных XML

Точки с запятой

Можно использовать символ точки с запятой (`;`) для завершения инструкции. В противном случае, если пропустить символ точки с запятой, компилятор предполагает, что каждая строка программного кода представляет один оператор. Поскольку многие программисты привыкли использовать точку с запятой для обозначения конца инструкции, программный код может быть удобнее для чтения, если согласованно использовать точки с запятой для завершения инструкций.

Использование точки с запятой для завершения инструкции позволяет поместить в одну строку больше одной инструкции, но при этом программный код становится менее удобочитаем.

Скобки

В ActionScript 3.0 можно использовать скобки (`()`) тремя способами. Во-первых, можно использовать скобки для изменения порядка операций в выражении. Операции, которые группируются в скобках, всегда выполняются первыми. Например, скобки используются для смены порядка операций в следующем программном коде.

```
trace(2 + 3 * 4); // 14
trace((2 + 3) * 4); // 20
```

Во-вторых, можно использовать скобки с оператором запятая (`,`) для вычисления ряда выражений и возврата результата окончательного выражения, как показано в следующем примере.

```
var a:int = 2;
var b:int = 3;
trace((a++, b++, a+b)); // 7
```

В-третьих, можно использовать скобки для передачи одного или нескольких параметров функциям или методам, как показано в следующем примере, в котором значение String передается функции `trace()`.

```
trace("hello"); // hello
```

Комментарии

Программный код ActionScript 3.0 поддерживает два типа комментариев: однострочные комментарии и многострочные комментарии. Механизмы этих комментариев сходны с аналогичными механизмами в языках C++ и Java. Компилятор игнорирует текст, отмеченный как комментарий.

Однострочный комментарий начинается с двух косых черт (`//`) и продолжается до конца строки. Например, в следующем программном коде содержится однострочный комментарий.

```
var someNumber:Number = 3; // a single line comment
```

Многострочные комментарии начинаются с косой черты и звездочки (`/*`) и заканчиваются звездочкой и косой чертой (`*/`).

```
/* This is multiline comment that can span
more than one line of code. */
```

Ключевые слова и зарезервированные слова

Зарезервированные слова — это слова, которые нельзя использовать в качестве идентификаторов в программном коде, потому что эти слова зарезервированы для применения в ActionScript. Зарезервированные слова включают *лексические ключевые слова*, которые удаляются компилятором из пространства имен программы. Компилятор сообщает об ошибке, если какое-либо лексическое ключевое слово используется в качестве идентификатора. В следующей таблице перечислен список лексических ключевых слов ActionScript 3.0.

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	for	function
if	implements	import	in
instanceof	interface	internal	is
native	new	null	package
private	protected	public	return
super	switch	this	throw
to	true	try	typeof
use	var	void	while
with			

Существует небольшое подмножество ключевых слов, называемых *синтаксическими ключевыми словами*, которые могут использоваться как идентификаторы, но при этом имеют специальное значение в определенных контекстах. В следующей таблице перечислен список синтаксических ключевых слов ActionScript 3.0.

each	get	set	namespace
include	dynamic	final	native
override	static		

Также существует несколько идентификаторов, которые иногда упоминаются как *зарезервированные для будущего использования слова*. Эти идентификаторы не зарезервированы в ActionScript 3.0, хотя некоторые из них могут расцениваться как ключевые слова программным обеспечением для работы с ActionScript 3.0. Многие из этих идентификаторов можно использовать в программном коде, но корпорация Adobe не рекомендует использовать их, поскольку они могут оказаться ключевыми словами в последующих версиях языка ActionScript.

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic

long	prototype	short	synchronized
throws	to	transient	type
virtual	volatile		

Константы

ActionScript 3.0 поддерживает инструкцию `const`, с помощью которой можно создавать константы. Константами называют свойства с фиксированным значением, которое нельзя изменить. Значение константе можно присвоить только один раз, и это назначение должно происходить в непосредственной близости от объявления этой константы. Например, если константа объявлена в качестве члена одного класса, можно назначить значение константе только в самом ее объявлении или внутри конструктора класса.

В следующем программном коде объявляются две константы. Первой константе, `MINIMUM`, значение назначается в инструкции объявления константы. Второй константе, `MAXIMUM`, значение назначается в конструкторе. Обратите внимание, что этот пример компилируется только в стандартном режиме, поскольку с строгом режиме значения константам разрешено присваивать только во время инициализации.

```
class A
{
    public const MINIMUM:int = 0;
    public const MAXIMUM:int;

    public function A()
    {
        MAXIMUM = 10;
    }
}

var a:A = new A();
trace(a.MINIMUM); // 0
trace(a.MAXIMUM); // 10
```

Если попытаться присвоить исходное значение константе любым другим способом, произойдет ошибка. Например, если предпринимается попытка установить исходное значение свойства `MAXIMUM` вне данного класса, возникает ошибка выполнения.

```
class A
{
public const MINIMUM:int = 0;
public const MAXIMUM:int;
}

var a:A = new A();
a["MAXIMUM"] = 10; // run-time error
```

ActionScript 3.0 определяет широкий диапазон готовых к использованию констант. В соответствии с соглашением, константы в ActionScript обозначаются заглавными буквами с разделением слов символом подчеркивания (_). Например, определение класса MouseEvent использует соглашение об именовании своих констант, каждая из которых отражает событие, связанное с вводом данных мышью.

```
package flash.events
{
    public class MouseEvent extends Event
    {
    public static const CLICK:String = "click";
    public static const DOUBLE_CLICK:String = "doubleClick";
    public static const MOUSE_DOWN:String = "mouseDown";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...
    }
}
```

Операторы

Операторы являются специальными функциями, принимающими один или несколько операндов и возвращающими значения. *Операнд* — это значение, обычно литерал, переменная или выражение, которые оператор использует в качестве входных данных. Например, в следующем программном коде операторы сложения (`+`) и умножения (`*`) используются с тремя литеральными операндами (2, 3, 4) для получения значения. Это значение затем используется оператором присваивания (`=`), чтобы назначить возвращаемое значение, 14, переменной `sumNumber`.

```
var sumNumber:uint = 2 + 3 * 4; // uint = 14
```

Операторы могут быть унарными, бинарными или тернарными. *Унарный* оператор принимает один операнд. Например, оператор инкрементального увеличения (`++`) является унарным оператором, поскольку он принимает только один операнд. *Бинарный* оператор принимает два операнда. Например, оператор деления (`/`) принимает два операнда. *Тернарный* оператор принимает три операнда. Например, оператор условия (`?:`) принимает три операнда.

Некоторые операторы являются *перегруженными*; это означает, что они ведут себя по-разному в зависимости от типа или количества переданных ими операндов. Оператор сложения (`+`) является примером перегруженного оператора, который ведет себя по-разному в зависимости от типа операндов. Если оба операнда являются числами, оператор сложения возвращает сумму этих значений. Если оба операнда являются строками, оператор сложения возвращает последовательное соединение двух операндов. В следующем программном коде показано, как оператор по-разному ведет себя в зависимости от операндов.

```
trace(5 + 5); // 10
trace("5" + "5"); // 55
```

Поведение операторов также может отличаться в зависимости от числа передаваемых операндов. Оператор вычитания (`-`) является одновременно унарным и бинарным оператором. Если оператор вычитания используется только с одним операндом, оператор меняет его знак и возвращает результат. Если оператор вычитания используется с двумя операндами, он возвращает разность между ними. В следующем примере показан оператор вычитания, используемый вначале как унарный оператор, а затем как бинарный.

```
trace(-3); // -3
trace(7 - 2); // 5
```

Старшинство и ассоциативность операторов

Старшинство и ассоциативность операторов определяют порядок, в котором обрабатываются операторы. Хотя может показаться абсолютно естественным для всех, кто знаком с арифметическими операциями, что компилятор должен обрабатывать оператор умножения (`*`) перед оператором сложения (`+`), в действительности компилятору необходимо явное указание того, какие операторы обрабатывать в первую очередь. Такие инструкции обычно упоминаются как *старшинство операторов*. ActionScript определяет старшинство операторов по умолчанию, которое можно изменить с помощью оператора скобки (`()`). Например, в следующем программном коде изменяется заданное по умолчанию старшинство операторов из предыдущего примера, чтобы заставить компилятор обработать оператор сложения перед оператором умножения.

```
var sumNumber:uint = (2 + 3) * 4; // uint == 20
```

Могут возникнуть ситуации, в которых в одном выражении встретятся два или больше операторов одного старшинства. В таких случаях компилятор использует правила *ассоциативности*, чтобы определить, какой оператор обрабатывать в первую очередь. Все бинарные операторы, за исключением оператора присваивания, являются *ассоциируемыми слева* , а это значит, что операторы слева обрабатываются раньше, чем операторы справа. Операторы присваивания и оператор условия (`?:`) являются *ассоциируемыми справа* операторами, и это означает, что операторы справа обрабатываются раньше, чем операторы слева.

Например, рассмотрим операторы меньше (`<`) и больше (`>`), имеющие одинаковое старшинство. Если оба оператора используются в одном выражении, оператор слева обрабатывается раньше, поскольку оба оператора являются ассоциируемыми слева. Это означает, что результат выполнения следующих двух инструкций будет одинаков.

```
trace(3 > 2 < 1); // false
trace((3 > 2) < 1); // false
```

Оператор «больше» обрабатывается в первую очередь, что приводит к итоговому значению `true`, поскольку операнд 3 больше, чем операнд 2. Значение `true` затем передается оператору «меньше» вместе с операндом 1. В следующем программном коде показано промежуточное состояние.

```
trace((true) < 1);
```

Оператор «меньше» преобразует значение `true` в цифровое значение 1 и сравнивает это числовое значение со вторым операндом 1, возвращая значение `false` (значение 1 не меньше значения 1).

```
trace(1 < 1); // false
```

Заданную по умолчанию ассоциативность слева можно изменить с помощью оператора скобки. Можно заставить компилятор обрабатывать оператор «меньше» первым, если заключить этот оператор и его операнды в скобки. В следующем примере используется оператор скобки, чтобы получить другой результат, взяв в качестве операндов числа из предыдущего примера.

```
trace(3 > (2 < 1)); // true
```

Оператор «меньше» обрабатывается в первую очередь, что приводит к итоговому значению `false`, поскольку операнд 2 не меньше, чем операнд 1. Значение `false` затем передается оператору «больше» вместе с операндом 3. В следующем программном коде показано промежуточное состояние.

```
trace(3 > (false));
```

Оператор «больше» преобразует значение `false` в числовое значение 0 и сравнивает это числовое значение с операндом 3, возвращая значение `true` (значение 3 больше, чем 0).

```
trace(3 > 0); // true
```

В следующей таблице приведен список операторов ActionScript 3.0 в порядке уменьшения их старшинства. Каждая строка таблицы содержит операторы одинакового старшинства. Каждая строка операторов имеет более высокое старшинство, чем строка, отображающаяся ниже в таблице.

Группа	**Операторы**
Основные	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Постфиксные	`x++ x--`
Унарные	`++x --x + - ~ ! delete typeof void`
Мультипликативные	`* / %`
Аддитивные	`+ -`
Побитовый сдвиг	`<< >> >>>`
Реляционные	`< > <= >= as in instanceof is`
Равенство	`== != === !==`
Побитовое AND	`&`
Побитовое XOR	`^`
Побитовое OR	`|`
Логическое AND	`&&`
Логическое OR	`||`
Условие	`?:`
Присваивание	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Запятая	`,`

Основные операторы

В число основных операторов входят те операторы, которые используются для создания литералов Array и Object, выражений группирования, функций вызова, инициации экземпляров классов, а также доступа к свойствам.

Все основные операторы, как показано в следующей таблице, имеют одинаковое старшинство. Операторы, являющиеся частью спецификации E4X, показаны с условным знаком (E4X).

Оператор	**Выполняемая операция**
`[]`	Инициализируется массив
`{x:y}`	Инициализируется объект
`()`	Группируются выражения
`f(x)`	Выполняется вызов функции
`new`	Выполняется вызов конструктора
`x.y x[y]`	Выполняется доступ к свойству
`<></>`	Инициализируется объект XMLList (E4X)

Оператор	Выполняемая операция
`@`	Выполняется доступ к атрибуту (E4X)
`::`	Уточняется имя (E4X)
`..`	Выполняется доступ к производному XML-элементу (E4X)

Постфиксные операторы

Постфиксные операторы берут какой-либо оператор и либо инкрементально увеличивают, либо декрементально уменьшают его значение. Хотя эти операторы и являются унарными, они классифицируются отдельно от остальных унарных операторов из-за своего высокого старшинства и особого поведения. Если постфиксный оператор используется как часть более крупного выражения, значение выражения возвращается до обработки этого оператора. Например, в следующем программном коде показано, что значение выражение `xNum++` возвращается до того, как значение инкрементально увеличивается.

```
var xNum:Number = 0;
trace(xNum++); // 0
trace(xNum); // 1
```

Все постфиксные операторы, как показано в следующей таблице, имеют одинаковое старшинство.

Оператор	Выполняемая операция
`++`	Инкрементально увеличивает (постфиксный)
`--`	Декрементально уменьшает (постфиксный)

Унарные операторы

Унарные операторы получают один операнд. Операторы инкрементального увеличения (`++`) и декрементального уменьшения (`--`) в этой группе являются *префикснымиоператорами*, и это означает, что они могут появляться перед операндами в выражении. Префиксные операторы отличаются от своих постфиксных эквивалентов тем, что операции инкрементального увеличения и декрементального уменьшения выполняются перед возвращением значения всего выражения. Например, в следующем программном коде показано, что значение выражение `++xNum` возвращается до того, как значение инкрементально увеличивается.

```
var xNum:Number = 0;
trace(++xNum); // 1
trace(xNum); // 1
```

Все унарные операторы, как показано в следующей таблице, имеют одинаковое старшинство.

Оператор	Выполняемая операция
`++`	Инкрементально увеличивает (префиксный)
`--`	Декрементально уменьшает (префиксный)
`+`	Унарный +
`-`	Унарный — (отрицание)
`!`	Логическое NOT
`~`	Побитовое NOT

Оператор	Выполняемая операция
`delete`	Свойство удаляется
`typeof`	Возвращается информация о типе
`void`	Возвращается значение undefined

Мультипликативные операторы

Мультипликативные операторы получают два операнда и выполняют умножение, деление или вычисление модуля.

Все мультипликативные операторы, как показано в следующей таблице, имеют одинаковое старшинство.

Оператор	Выполняемая операция
`*`	Умножение
`/`	Деление
`%`	Вычисление модуля

Аддитивные операторы

Аддитивные операторы получают два операнда и выполняют операции сложения или вычитания. Все аддитивные операторы, как показано в следующей таблице, имеют одинаковое старшинство.

Оператор	Выполняемая операция
`+`	Сложение
`-`	Вычитание

Операторы побитового сдвига

Операторы побитового сдвига получают два операнда и сдвигают биты первого операнда на величину, указанную во втором операнде. Все операторы побитового сдвига, как показано в следующей таблице, имеют одинаковое старшинство.

Оператор	Выполняемая операция
`<<`	Побитовый сдвиг влево
`>>`	Побитовый сдвиг вправо
`>>>`	Побитовый сдвиг вправо без знака

Реляционные операторы

Реляционные операторы получают два операнда, сравнивают их значения, а затем возвращают логическое значение (Boolean). Все реляционные операторы, как показано в следующей таблице, имеют одинаковое старшинство.

Оператор	Выполняемая операция
<	Меньше
>	Больше
<=	Меньше или равно
>=	Больше или равно
as	Проверяется тип данных
in	Проверяются свойства объекта
instanceof	Проверяется цепочка прототипа
is	Проверяется тип данных

Операторы равенства

Операторы равенства получают два операнда, сравнивают их значения, а затем возвращают логическое значение (Boolean). Все операторы равенства, как показано в следующей таблице, имеют одинаковое старшинство.

Оператор	Выполняемая операция
==	Равенство
!=	Неравенство
===	Строгое равенство
!==	Строгое неравенство

Побитовые логические операторы

Побитовые логические операторы получают два операнда и выполняют логические операции на битовом уровне. Побитовые логические операторы отличаются по старшинству и перечислены в следующей таблице в порядке убывания старшинства.

Оператор	Выполняемая операция
&	Побитовое AND
^	Побитовое XOR
|	Побитовое OR

Логические операторы

Логические операторы получают два операнда и возвращают логическое значение (Boolean). Логические операторы отличаются по старшинству и перечислены в следующей таблице в порядке убывания старшинства.

Оператор	Выполняемая операция
&&	Логическое AND
||	Логическое OR

Оператор условия

Оператор условия является тернарным оператором, и это значит, что ему передается три операнда. Оператор условия — это экспресс-метод применения инструкции условного выбора `if..else` .

Оператор	Выполняемая операция
`?:`	Условие

Операторы присваивания

Операторы присваивания получают два оператора и присваивают значение одному оператору на основе значения другого оператора. Все операторы присваивания, как показано в следующей таблице, имеют одинаковое старшинство.

Оператор	Выполняемая операция
`=`	Присваивание
`*=`	Присваивание умножения
`/=`	Присваивание удаление
`%=`	Присваивание модуля
`+=`	Присваивание сложения
`-=`	Присваивание вычитания
`<<=`	Присваивание побитового сдвига влево
`>>=`	Присваивание побитового сдвига вправо
`>>>=`	Присваивание побитового сдвига вправо без знака
`&=`	Присваивание побитового AND
`^=`	Присваивание побитового XOR
`|=`	Присваивание побитового OR

Условия

ActionScript 3.0 обеспечивает три основных инструкции условия, которые можно использовать для управления программным потоком.

инструкция if..else

Инструкция условия `if..else` позволяет проверить условие и выполнить блок программного кода, если это условие соблюдено, либо выполнить другой программный код, если условие нарушено. Например, в следующем программном коде проверяется, превышает ли `x` значение 20, создается функция `trace()`, если условие выполняется, или создается другая функция `trace()`, если это не так.

```
if (x > 20)
{
    trace("x is > 20");
}
else
{
    trace("x is <= 20");
}
```

Если не требуется выполнять альтернативный блок программного кода, можно использовать инструкцию `if` без инструкции `else`.

инструкция if..else if

Можно проверить больше одного условия, используя инструкцию условия `if..else if`. Например, в следующем программном коде не только проверяется, превышает ли `x` значение 20, но также проверяется, не является ли значение `x` отрицательным.

```
if (x > 20)
{
    trace("x is > 20");
}
else if (x < 0)
{
    trace("x is negative");
}
```

Если после оператора `if` или `else` следует только один оператор, этот оператор не нужно заключать в фигурные скобки. Например, в следующем программном коде фигурные скобки не используются.

```
if (x > 0)
    trace("x is positive");
else if (x < 0)
    trace("x is negative");
else
    trace("x is 0");
```

Однако Adobe рекомендует всегда использовать фигурные скобки, поскольку если в дальнейшем к оператору условия без скобок добавляется дополнительное условие, возможно непредвиденное поведение программного кода. Например, в следующем программном коде значение `positiveNums` увеличивается на 1 независимо от того, получается ли в результате проверки условия значение `true` или нет.

```
var x:int;
var positiveNums:int = 0;

if (x > 0)
    trace("x is positive");
    positiveNums++;

trace(positiveNums); // 1
```

switch

Инструкция `switch` полезна, если имеется несколько путей выполнения, которые зависят от одного выражения условия. При этом обеспечивается функциональность, сходная с несколькими последовательными инструкциями `if..else if`, но несколько более удобочитаемая. Вместо проверки условия для логического значения инструкция `switch` оценивает выражение и использует результат для определения того, какой блок программного кода следует выполнять. Блоки кода начинаются инструкциями `case` и заканчиваются инструкциями `break`. Например, в следующей инструкции `switch` печатается день недели на основе номера дня, возвращаемого методом `Date.getDay()`.

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
    case 0:
        trace("Sunday");
        break;
    case 1:
        trace("Monday");
        break;
    case 2:
        trace("Tuesday");
        break;
    case 3:
        trace("Wednesday");
        break;
    case 4:
        trace("Thursday");
        break;
    case 5:
        trace("Friday");
        break;
    case 6:
        trace("Saturday");
        break;
    default:
        trace("Out of range");
        break;
}
```

Повтор

Инструкции цикла позволяют повторно выполнить определенный блок программного кода, используя несколько значений переменных. Adobe рекомендует всегда заключать в фигурные скобки ({}) блок кода. Хотя можно пропустить фигурные скобки, если блок кода содержит всего один оператор, на практике так делать не рекомендуется по той же причине, по которой не рекомендуется и для условий: увеличивается вероятность непреднамеренного исключения оператора из блока кода после его добавления впоследствии. Если впоследствии добавится оператор, который нужно включить в блок кода, но при этом будут пропущены необходимые фигурные скобки, оператор не будет выполняться как часть цикла.

for

Инструкция цикла `for` позволяет итеративно увеличивать переменную в определенном диапазоне значений. Необходимо передать в инструкцию `for` три выражения: переменную, установленную в качестве исходного значения; инструкцию условия, определяющую окончание цикла; а также выражение, которое изменяет значение переменной для каждого цикла. Например, в следующем программном коде цикл выполняется пять раз. Значение переменной `i` начинается с 0 и заканчивается 4, выходным значением являются числа от 0 до 4, каждое в своей строке.

```
var i:int;
for (i = 0; i < 5; i++)
{
    trace(i);
}
```

for..in

Инструкция цикла `for..in` выполняет итерации для свойства объекта или элементов массива. Например, можно использовать инструкцию цикла `for..in` для выполнения итераций со значениями свойства родового объекта (свойства объекта не сохраняются в определенном порядке, поэтому свойства могут появляться в кажущемся произвольном порядке).

```
var myObj:Object = {x:20, y:30};
for (var i:String in myObj)
{
    trace(i + ": " + myObj[i]);
}
// output:
// x: 20
// y: 30
```

Также можно выполнять итерацию по элементам массива.

```
var myArray:Array = ["one", "two", "three"];
for (var i:String in myArray)
{
    trace(myArray[i]);
}
// output:
// one
// two
// three
```

Единственным действием, которое можно выполнить, является итерация для свойств объекта, если объект является экземпляром запечатанного класса (включая встроенные классы и заданные пользователем классы). Можно выполнить итерацию только для свойств динамического класса. Даже для экземпляров динамического класса можно выполнять только итерацию для свойств, добавляемых динамически.

for each..in

Инструкция цикла `for each..in` выполняет итерации по элементам коллекции, которые могут быть тегами в объекте XML или XMLList, значениями, содержащимися в свойствах объекта, или элементами массива. Например, как показывает следующая выборка, можно использовать цикл `for each..in` для итерации по свойствам родового объекта, но в отличие от цикла `for..in` переменная-итератор в цикле `for each..in` содержит значение свойства вместо имени свойства.

```
var myObj:Object = {x:20, y:30};
for each (var num in myObj)
{
    trace(num);
}
// output:
// 20
// 30
```

Можно выполнять итерации по объектам XML или XMLList, как показано в следующем примере.

```
var myXML:XML = <users>
    <fname>Jane</fname>
    <fname>Susan</fname>
    <fname>John</fname>
</users>;

for each (var item in myXML.fname)
{
    trace(item);
}
/* output
Jane
Susan
John
*/
```

Также можно выполнять итерации по элементам массива, как в следующем примере.

```
var myArray:Array = ["one", "two", "three"];
for each (var item in myArray)
{
    trace(item);
}
// output:
// one
// two
// three
```

Нельзя выполнять итерацию по свойствам объекта, если он является экземпляром запечатанного класса. Даже для экземпляров динамических классов нельзя выполнять итерации по каким-либо фиксированным свойствам, какими являются свойства, определенные как часть определения класса.

while

Цикл `while` работает аналогично инструкции `if`, выполняя повторения до тех пор, пока для условия выдается значение `true`. Например, в следующем программном коде получаются такие же результаты, что и в примере с циклом `for`.

```
var i:int = 0;
while (i < 5)
{
    trace(i);
    i++;
}
```

Одним из недостатков использования цикла `while` вместо цикла `for` является то, что неопределенные циклы проще писать с помощью циклов `while`. Пример программного кода с циклом `for` не компилируется, если пропущено выражение, которое инкрементально увеличивает переменную обратного счетчика, но цикл `while` не компилируется, если пропущен этот шаг. Без выражения, инкрементально увеличивающего переменную `i`, цикл становится неопределенным циклом.

do..while

Цикл `do..while` является циклом `while`, в котором гарантируется, что блок кода будет выполнен хотя бы один раз, поскольку условие проверяется после выполнения блока программного кода. В следующем программном коде показан простой пример цикла `do..while`, создающего выходные данные даже в случае несоблюдения условия.

```
var i:int = 5;
do
{
    trace(i);
    i++;
} while (i < 5);
// output: 5
```

Функции

Функциями называют блоки программного кода, выполняющие определенные задачи, которые можно повторно использовать в программах. В ActionScript 3.0 существует два типа функций: *методы* и *закрытия функций*. То, называется ли функция методом или закрытием функции, зависит от контекста, в котором определена эта функция. Функция называется методом, если она определяется как часть определения класса или прикрепляется к экземпляру объекта. Функция называется закрытием функции, если она определена каким-либо другим способом.

Функции всегда были очень важны в ActionScript. В ActionScript 1.0, например, ключевое слово `class` отсутствует, поэтому «классы» определялись функциями конструктора. Хотя ключевое слово `class` с тех пор и было добавлено к языку, четкое понимание функций по-прежнему важно, если требуется полностью использовать все преимущества, предлагаемые языком. Это, возможно, вызовет сложности у программистов, ожидающих, что функции ActionScript будут вести себя так же, как функции в языке C++ или Java. Хотя само определение основной функции и ее вызовов не должно представлять сложности для опытных программистов, некоторые более сложные возможности функций ActionScript требуют дополнительного объяснения.

Концепции основной функции

Вызов функций

При вызове функции с помощью ее идентификатора с указанием последующего оператора скобки (`()`). Оператор скобки используется для включения любых функциональных параметров, которые требуется передать функции. Например, функция `trace()` является функцией верхнего уровня в среде ActionScript 3.0:

```
trace("Use trace to help debug your script");
```

Если вызывается функция без параметров, необходимо использовать пустые парные скобки. Например, можно использовать метод `Math.random()`, который не получает параметров, чтобы создавать произвольные числа.

```
var randomNum:Number = Math.random();
```

Определение собственных функций

В ActionScript 3.0 существует два способа определить функцию: можно использовать инструкцию function или выражение function. Выбираемая технология зависит от того, какой стиль программирования предпочтительнее: статический или динамический. Определите собственные функции с помощью инструкции function, если предпочтителен статический или строгий режим программирования. Определяйте свои функции с помощью выражения function, если в этом есть особая необходимость. Выражение function более часто используется в динамическом программировании или при использовании стандартного режима.

Инструкции Function

Инструкции Function являются предпочтительной методикой для определения функций в строгом режиме. Инструкция function начинается с ключевого слова `function`, за которым идут следующие элементы.

- Имя функции
- Заключенный в скобки список параметров, разделенных запятыми
- Тело функции — это заключенный в фигурные скобки программный код ActionScript, который будет выполняться при вызове функции

Например, в следующем программном коде создается функция, определяющая параметр, а затем выполняется вызов этой функции с использованием строки `"hello"` в качестве значения параметра:

```
function traceParameter(aParam:String)
{
    trace(aParam);
}

traceParameter("hello"); // hello
```

Выражения Function

Можно другим способом объявить функцию, используя инструкцию присваивания с выражением function. Этот метод иногда называется литералом функции или анонимной функцией. Это более трудоемкий метод, который широко использовался в ранних версиях ActionScript.

Инструкция присваивания с выражением функции начинается с ключевого слова `var`, за которым идут следующие элементы.

- Имя функции
- Оператор двоеточие (`:`)
- Класс `Function` для указания типа данных
- Оператор присваивания (`=`)
- Ключевое слово `function`
- Заключенный в скобки список параметров, разделенных запятыми
- Тело функции — это заключенный в фигурные скобки программный код ActionScript, который будет выполняться при вызове функции

Например, в следующем программном коде объявляется функция `traceParameter`, использующая выражение function.

```
var traceParameter:Function = function (aParam:String)
{
    trace(aParam);
};
traceParameter("hello"); // hello
```

Обратите внимание, что в отличие от инструкции function имя функции не указывается. Другим важным отличием между выражениями function и инструкциями function является то, что выражение - это именно выражение, а не инструкция. Это означает, что выражение function не может использоваться отдельно подобно инструкции function. Выражение function может использоваться только в качестве части инструкции, обычно инструкции присваивания. В следующем примере показано выражение function, присвоенное элементу массива.

```
var traceArray:Array = new Array();
traceArray[0] = function (aParam:String)
{
    trace(aParam);
};
traceArray[0]("hello");
```

Выбор между инструкцией и выражением

Стандартное правило предписывает использовать инструкцию function до тех пор, пока в силу особых обстоятельств не потребуется использовать выражение. Инструкции function менее трудоемки и обеспечивают более согласованную работу в строгом и стандартном режиме, чем выражения function.

Инструкции function проще читаются чем инструкции присваивания, содержащие выражения function. Инструкции function делают код более кратким; они вносят меньше путаницы, чем выражения function, для которых требуется использовать оба ключевых слова: `var` и `function`.

Операторы function обеспечивают более согласованное поведение в обоих режимах компилятора, поскольку для вызова метода, объявленного с помощью оператора function, можно использовать синтаксис с точкой как в строгом, так и в стандартном режимах. Это не обязательно так для методов, объявленных с помощью выражения function. Например, в следующем программном коде определяется класс с именем Example, содержащий два метода: `methodExpression()`, объявляемый с помощью выражения function, а также метод `methodStatement()`, объявляемый с помощью инструкции function. В строгом режиме нельзя использовать синтаксис с точкой для вызова метода `methodExpression()`.

```
class Example
{
var methodExpression = function() {}
function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // error in strict mode; okay in standard mode
myEx.methodStatement(); // okay in strict and standard modes
```

Считается, что выражения function лучше подходят для программирования, в котором больше внимания уделяется поведению во время выполнения или динамическому поведению. Если предпочтительно использовать строгий режим, но также необходимо вызывать метод, объявленный с помощью выражения function, можно использовать одну из двух технологий. Во-первых, можно вызвать метод, используя квадратные скобки (`[]`) вместо оператора точки (`.`). Вызов следующего метода успешно выполняется как в строгом, так и в стандартном режимах.

```
myExample["methodLiteral"]();
```

Во-вторых, можно объявить весь класс динамическим. Хотя это позволит вызывать метод с помощью оператора точки, с другой стороны придется пожертвовать возможностью работы в строгом режиме со всеми экземплярами данного класса. Например, что компилятор не создает ошибку при попытке доступа к неопределенному свойству экземпляра динамического класса.

В некоторых обстоятельствах выражения могут быть очень полезны. Одним из наиболее распространенных случаев применения выражений function являются функции, которые используются однократно и потом сбрасываются. Другим менее частым вариантом применения является присоединение функции к свойству прототипа. Дополнительные сведения см. в разделе ««Объект прототипа» на странице 126».

Существует два тонких различия между инструкциями function и выражениями function, которые следует учитывать при выборе методики использования. Первое отличие состоит в том, что выражения function не существуют независимо как объекты в отношении управления памятью и освобождения ресурсов. Говоря другими словами, если выражение function присваивается другому объекту, такому как элемент массива или свойство объекта, создается только ссылка на это выражение function в программном коде. Если этот элемент массива или объект, к которому прикрепляется выражение function, выходит из области действия или, говоря другими словами, больше недоступен, доступ к этому выражению function прекращается. Если этот элемент массива или объект удаляются, память, использовавшаяся выражением function, становится доступной для освобождения ресурсов, и это значит, что данную память можно затребовать и повторно использовать для других целей.

В следующем примере показано, что в выражении function, как только свойство, к которому присвоено выражение, удаляется, функция перестает быть доступной. Класс Test является динамическим, что означает, что можно добавить свойство с именем `functionExp`, в котором содержится выражение функции. Функции `functionExp()` могут вызываться с помощью оператора точки, но как только свойство `functionExp` удаляется, функция перестает быть доступной.

```
dynamic class Test {}
var myTest:Test = new Test();

// function expression
myTest.functionExp = function () { trace("Function expression") };
myTest.functionExp(); // Function expression
delete myTest.functionExp;
myTest.functionExp(); // error
```

Но, с другой стороны, если функция вначале определяется с помощью инструкции function, то она существует как независимый объект и продолжает существовать даже после удаления свойства, к которому она прикреплена. Оператор `delete` работает только со свойствами объектов, поэтому даже запрос на удаление самой функции `stateFunc()` не срабатывает.

```
dynamic class Test {}
var myTest:Test = new Test();

// function statement
function stateFunc() { trace("Function statement") }
myTest.statement = stateFunc;
myTest.statement(); // Function statement
delete myTest.statement;
delete stateFunc; // no effect
stateFunc();// Function statement
myTest.statement(); // error
```

Второе различие между инструкциями function и выражениями function состоит в том, что инструкции function существуют во всей области действия, для которой они определены, в том числе и для инструкций, предшествовавших инструкции function. Выражения function, в отличие от этого, определяются только для последующих инструкций. Например, в следующем программном коде функция `scopeTest()` успешно вызывается до того, как она была определена.

```
statementTest(); // statementTest

function statementTest():void
{
    trace("statementTest");
}
```

Выражения function недоступны до тех пор, пока они не определены, поэтому возникает ошибка при выполнении следующего программного кода.

```
expressionTest(); // run-time error

var expressionTest:Function = function ()
{
    trace("expressionTest");
}
```

Возвращение значений из функций

Чтобы возвратить значение из функции, используйте инструкцию `return` с последующим указанием значения выражения или литерала, которое требуется вернуть. Например, в следующем программном коде возвращается выражение, отображающее этот параметр.

```
function doubleNum(baseNum:int):int
{
    return (baseNum * 2);
}
```

Обратите внимание, что оператор `return` прерывает выполнение функции, поэтому любые операторы, следующие за оператором `return`, выполняться не будут, как показано далее.

```
function doubleNum(baseNum:int):int {
    return (baseNum * 2);
    trace("after return"); // This trace statement will not be executed.
}
```

В строгом режиме необходимо, чтобы возвращалось значение соответствующего типа, если указан тип возвращаемого значения. Например, следующий программный код создает ошибку в строгом режиме, поскольку не возвращает действительное значение.

```
function doubleNum(baseNum:int):int
{
    trace("after return");
}
```

Вложенные функции

Можно выполнять вложение функций, это означает, что функции могут быть объявлены с другими функциями. Вложенная функция доступна только в рамках родительской функции до тех пор, пока ссылки на эту функцию передаются во внешний программный код. Например, в следующем программном коде объявляется две вложенных функции внутри функции `getNameAndVersion()`.

```
function getNameAndVersion():String
{
    function getVersion():String
    {
        return "10";
    }
    function getProductName():String
    {
        return "Flash Player";
    }
    return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Flash Player 10
```

Если вложенные функции передаются во внешний программный код, они передаются как закрытия функций, и это означает, что функция сохраняет все определения, бывшие в области действия во время ее определения. Дополнительные сведения см. в разделе ««Область действия функции» на странице 92».

Параметры функций

ActionScript 3.0 обеспечивает определенную функциональность для параметров функций, которые могут показаться нестандартными для программистов, мало знакомых с этим языком. Хотя идея передачи параметров значением или ссылкой должна быть знакома большинству программистов, объект `arguments` и параметр ... (rest) многим еще не знакомы.

Передача аргументов значением и ссылкой

Во многих языках программирования очень важно понимать различие между передачей аргументов значением или ссылкой; это различие может влиять на способ разработки программного кода.

Передача значением подразумевает, что значение аргумента копируется в локальную переменную для использования внутри функции. Передача ссылкой означает, что вместо фактического значения передается только ссылка на аргумент. Копирование фактического значения аргумента не выполняется. Вместо этого передается ссылка на переменную, поскольку аргумент создается и присваивается локальной переменной для использования внутри функции. Так как ссылка на переменную находится вне функции, локальная переменная дает возможность изменять значение исходной переменной.

В ActionScript 3.0 все аргументы передаются ссылками, поскольку все значения хранятся как объекты. Однако объекты, принадлежащие примитивным типам данных, к которым относятся Boolean, Number, int, uint, и String, имеют специальные операторы, которые позволяют воспроизвести поведение объектов при передаче аргументов значениями. Например, в следующем программном коде создается функция с именем `passPrimitives()`, определяющая два параметра с именем `xParam` и `yParam`, оба с типом int. Эти параметры сходны с локальными переменными, объявленными в теле функции `passPrimitives()`. Если функция вызывается с аргументами `xValue` и `yValue`, параметры `xParam` и `yParam` инициализируются со ссылками на объекты int, представленные значениями `xValue` и `yValue`. Поскольку эти аргументы имеют примитивные типы, они ведут себя так, как если бы передавались значениями. Хотя параметры `xParam` и `yParam` изначально содержат только ссылки на `xValue` и `yValue` объекты, любые изменения переменных в теле функции создают новые копии значений в памяти.

```
function passPrimitives(xParam:int, yParam:int):void
{
    xParam++;
    yParam++;
    trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);// 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);// 10 15
```

В функции `passPrimitives()` значения `xParam` и `yParam` являются инкрементальными, но это не влияет на значения `xValue` и `yValue`, как показано в последней инструкции `trace`. И это действительно будет так, даже если параметры будут названы так же, как переменные, `xValue` и `yValue`, поскольку имена `xValue` и `yValue` внутри функции будут указывать на новое местоположение в памяти, которое существует отдельно от переменных с таким же именем вне функции.

Все другие объекты, т.е объекты, которые не принадлежат примитивным типам данных, всегда передаются ссылками, что дает возможность изменять значение исходной переменной. Например, в следующем программном коде создается объект с именем `objVar` и двумя свойствами: `x` и `y`. Этот объект передается как аргумент функции `passByRef()`. Поскольку тип данных объекта не примитивный, он не только предается ссылкой, но также хранится как ссылка. Это означает, что изменения, вносимые в параметры в функции, влияют на свойства объекта вне функции.

```
function passByRef(objParam:Object):void
{
    objParam.x++;
    objParam.y++;
    trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar); // 11 16
trace(objVar.x, objVar.y); // 11 16
```

Параметр `objParam` ссылается на тот же объект, что и глобальная переменная `objVar`. Как можно видеть из инструкции `trace` в приведенном примере, изменения свойств `x` и `y` в объекте `objParam` отражаются в объекте `objVar`.

Значения параметров по умолчанию

В среде ActionScript 3.0 можно объявить *значения параметров по умолчанию* для функции. Если вызывается функция со значениями параметров по умолчанию, используются значения, указанные для параметров в определении функции. Все параметры со значениями по умолчанию должны быть помещены в конец списка параметров. Значения, которые присвоены в качестве значений по умолчанию, должны быть константами, определяемыми во время компиляции. Существование для параметра значения по умолчанию фактически делает этот параметр *необязательным параметром*. Параметр без значения по умолчанию считается *обязательным параметром*.

Например, в следующем программном коде создается функция с тремя параметрами, два из которых имеют значения по умолчанию. При вызове этой функции только с одним параметром используются значения для параметров по умолчанию.

```
function defaultValues(x:int, y:int = 3, z:int = 5):void
{
    trace(x, y, z);
}
defaultValues(1); // 1 3 5
```

Объект arguments

Если параметры передаются в функцию, можно использовать объект `arguments` для доступа к информации о параметрах, переданных функции. Наиболее важными для объекта `arguments` являются следующие аспекты.

- Объект `arguments` является массивом, содержащим все параметры, переданные функции.
- Свойство `arguments.length` сообщает о числе параметров, переданных функции.
- Свойство `arguments.callee` передает ссылку на саму функцию, которая очень полезна при рекурсивных вызовах выражений function.

  Примечание. *Объект `arguments` недоступен, если какой-либо параметр имеет имя `arguments`, а также в тех случаях, когда используется параметр ... (rest).*

  Если в теле функции есть ссылки на объект `arguments`, ActionScript 3.0 позволяет включать в вызов функции больше параметров, чем указано в определении функции. Однако в строгом режиме будет выдаваться ошибка компиляции, если число переданных параметров не соответствует числу обязательных параметров (и дополнительных параметров, если такие имеются). Можно использовать этот аспект массива в объекте `arguments` для доступа к любому параметру, переданному функции, без учета того, был ли этот параметр задан в определении функции или нет. В следующем примере, который компилируется только в стандартном режиме, массив `arguments` со свойством `arguments.length` используется для отслеживания всех параметров, переданных в функцию `traceArgArray()`.

  ```
  function traceArgArray(x:int):void
  {
      for (var i:uint = 0; i < arguments.length; i++)
      {
          trace(arguments[i]);
      }
  }

  traceArgArray(1, 2, 3);

  // output:
  // 1
  // 2
  // 3
  ```

  Свойство `arguments.callee` часто используется в анонимных функциях для выполнения рекурсии. С ее помощью можно добавить гибкости создаваемому программному коду. Если имя рекурсивной функции изменяется в ходе цикла разработки, не следует беспокоиться об изменении рекурсивных вызовов в теле функции, если вместо имени функции используется свойство `arguments.callee`. Свойство `arguments.callee` используется в следующем выражении function для разрешения рекурсии.

```
var factorial:Function = function (x:uint)
{
    if(x == 0)
    {
        return 1;
    }
    else
    {
        return (x * arguments.callee(x - 1));
    }
}

trace(factorial(5)); // 120
```

Если при объявлении функции используется параметр ... (rest), объект `arguments` недоступен. Вместо него необходимо будет обращаться к параметрам, используя объявленные для них имена.

Необходимо соблюдать осторожность, чтобы избежать использования строки `"arguments"` в качестве имени параметра, поскольку из-за этого объект `arguments` может стать недоступным. Например, если функция `traceArgArray()` перезаписывается так, что добавляется параметр `arguments`, ссылка `arguments` в теле функции будет вызывать обращение параметру, а не к объекту `arguments`. В следующем программном коде не создается значений на выходе.

```
function traceArgArray(x:int, arguments:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// no output
```

Объект `arguments` в предыдущих версиях ActionScript также содержал свойство с именем `caller`, которое является ссылкой на функцию, вызывающую текущую функцию. Свойство `caller` отсутствует в ActionScript 3.0, но если нужна ссылка на вызывающую функцию, можно изменить ее так, чтобы она передавала дополнительный параметр, являющейся ссылкой на нее саму.

Параметр ... (rest)

В ActionScript 3.0 появилось новая возможность объявления параметров, называемая параметр ... (rest). Этот параметр позволяет указать параметр массива, который принимает любое количество разделенных запятыми аргументов. Этот параметр может иметь любое имя при условии, что оно не является зарезервированным словом. Это объявление параметра должно быть последним указываемым параметром. Использование этого параметра делает объект `arguments` недоступным. Хотя параметр ... (rest) дает ту же функциональность, что массив `arguments` и свойство `arguments.length`, он не обеспечивает возможностей свойства `arguments.callee`. Необходимо убедиться, что использование возможностей свойства `arguments.callee` действительно не требуется, прежде чем использовать параметр ... (rest).

В следующем примере функция `traceArgArray()` перезаписывается с использованием параметра ... (rest) вместо объекта `arguments`.

```
function traceArgArray(... args):void
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

Параметр ... (rest) также может использоваться с другими параметрами с условием, что он будет последним параметром в списке. В следующем примере функция `traceArgArray()` изменяется таким образом, что ее первый параметр, `x`, становится типом int, а второй — использует параметр ... (rest). При выводе первое значение пропускается, поскольку первый параметр больше не является частью массива, созданного параметром ... (rest).

```
function traceArgArray(x: int, ... args)
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 2
// 3
```

Функции в качестве объектов

В ActionScript 3.0 функции являются объектами. При создании функции создается объект, который не только может передаваться как параметр другой функции, но также имеет свойства и методы, присоединенные к нему.

Функции передаются как аргументы другим функциям в виде ссылок, а не значений. Если функция передается как аргумент, используется только ее идентификатор, но не оператор скобки, применяемый для вызова метода. Например, в следующем программном коде функция с именем `clickListener()` передается как аргумент методу `addEventListener()`.

```
addEventListener(MouseEvent.CLICK, clickListener);
```

Хотя это может показаться странным для программистов, которые только начинают изучать ActionScript, как и любой другой объект, функции могут иметь свойства и методы. Действительно, каждая функция обладает доступным только для чтения свойством с именем `length`, в котором хранится число параметров, определенное для функции. Это свойство отличается от свойства `arguments.length`, в котором сообщалось о числе аргументов, переданных функции. Вспомним, что в ActionScript число аргументов, переданных функции, может превышать число параметров, заданных в определении этой функции. В следующем примере, который компилируется только в стандартном режиме, поскольку для строгого режима требуется точное соответствие числа переданных аргументов и параметров в определении, показано различие между этими двумя свойствами.

```
// Compiles only in standard mode
function traceLength(x:uint, y:uint):void
{
    trace("arguments received: " + arguments.length);
    trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* output:
arguments received: 4
arguments expected: 2 */
```

В стандартном режиме можно определить свои собственные свойства функции, поместив их определение вне тела создаваемой функции. Свойства функции могут выступать в роли квази-статических свойств, позволяя сохранять состояние переменной, связанной с данной функцией. Например, возможно, потребуется отследить, сколько раз вызывается определенная функция. Такая возможность может быть полезна, если пишется игра и требуется отследить, сколько раз пользователь применял определенную команду, хотя для этой цели вполне можно использовать и свойство статического класса. В следующем примере, компилируемом только в стандартном режиме, поскольку строгий режим не допускает добавления динамических свойств к функциям, создается свойство функции вне ее определения, а затем это свойство инкрементально увеличивается при каждом вызове функции.

```
// Compiles only in standard mode
var someFunction:Function = function ():void
{
    someFunction.counter++;
}

someFunction.counter = 0;

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

Область действия функции

Область действия функции определяет не только, где в программе может вызываться функция, но также определения, которые доступны этой функции. Те же правила области действия, которые применимы к идентификаторам переменных, применяются и к идентификаторам функций. Функция, объявленная в глобальной области действия, доступна в любом месте программного кода. Например, ActionScript 3.0 содержит глобальные функции, такие как `isNaN()` и `parseInt()`, которые доступны в любом месте программного кода. Вложенная — это функция, объявленная внутри другой функции, которая может использоваться в любом месте функции, внутри которой она объявлена.

Цепочка области действия

Всякий раз, когда начинается выполнение функции, создается ряд объектов и свойств. Во-первых, создается специальный объект с названием *объект активации*, который хранит параметры и все локальные переменные или функции, объявленные в теле данной функции. Напрямую обратиться к объекту активации нельзя, поскольку это внутренний механизм. Во-вторых, создается *цепочка области действия*, содержащая упорядоченный список объектов, в которых среда выполнения проверяет объявления идентификаторов. Каждая выполняемая функция имеет цепочку области действия, хранимую во внутреннем свойстве. Для вложенных функций цепочка области действия начинается с собственного объекта активации, за которым следует объект активации родительской функции. Цепочка продолжается таким же образом, пока не будет достигнут глобальный объект. Глобальный объект создается при запуске программы ActionScript и содержит все глобальные переменные и функции.

Закрытия функций

Закрытие функции — это объект, содержащий моментальный снимок функции и ее *лексическую среду*. Лексическая среда функции включает все ее переменные, свойства, методы и объекты в цепочке области действия функции, а также их значения. Закрытие функции создается при каждом выполнении функции отдельно от объекта или класса. Тот факт, что закрытия функций удерживают области действия, в которых они были определены, дает интересные результаты, если функция передается в качестве аргумента или возвращаемого значения в другую область действия.

Например, в следующем программном коде создаются две функции: `foo()`, которая возвращает вложенную функцию с именем `rectArea()`, рассчитывающую площадь прямоугольника, и функцию `bar()`, вызывающую `foo()` и сохраняющую возвращаемое закрытие функции в переменной с именем `myProduct`. Несмотря на то, что функция `bar()` определяет свою собственную локальную переменную `x` (со значением 2), если вызывается закрытие функции `myProduct()`, она сохраняет переменную `x` (со значением 40), определенную в функции `foo()`. Функция `bar()`, таким образом, возвращает значение `160` вместо `8`.

```
function foo():Function
{
    var x:int = 40;
    function rectArea(y:int):int // function closure defined
    {
        return x * y
    }
    return rectArea;
}
function bar():void
{
    var x:int = 2;
    var y:int = 4;
    var myProduct:Function = foo();
    trace(myProduct(4)); // function closure called
}
bar(); // 160
```

Методы ведут себя сходным образом, они также сохраняют информацию о лексической среде, в которой они были созданы. Эта особенность наиболее заметна, если метод извлекается из своего экземпляра, создавая связанный метод. Главное отличие между закрытием функции и связанным методом состоит в том, что значение ключевого слова `this` в связанном методе всегда ссылается на экземпляр, к которому он был первоначально прикреплен, в то время как в закрытии функции значение ключевого слова `this` можно изменять.

Глава 4. Объектно-ориентированное программирование на языке ActionScript

Введение в объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — это способ упорядочения кода в программе за счет его группировки в объектах. Термин *объект* в этом случае обозначает отдельный элемент, содержащий данные (значения данных) и функции. Если выбран объектно-ориентированный подход к организации программы, выполняется группировка определенных частей информации с общим набором функций или действий, связанных с этой информацией. Например, можно сгруппировать данные о музыке, такие как название альбома, название дорожки или имя исполнителя, с такими функциями, как добавление дорожки в список воспроизведения или воспроизведение всех песен какого-либо исполнителя. Эти части объединяются в один элемент, называемый объектом (например, Album или MusicTrack). Объединение значений и функций обеспечивает ряд преимуществ. Одним ключевым преимуществом является то, что вместо нескольких переменных необходимо использовать одну переменную. Кроме того, связанные функции хранятся вместе. Наконец, объединение информации и функций позволяет создать структуру программы, которая более точно соответствует реальным условиям.

Классы

Класс — это абстрактное представление объекта. В классе хранится информация о типах данных, которые может содержать объект, и о его возможных видах поведения. Полезность такой абстракции может быть не такой заметной при создании небольших скриптов, содержащих всего несколько взаимодействующих между собой объектов. Однако по мере увеличения размера программы увеличивается число объектов, которыми необходимо управлять. В этом случае классы позволяют обеспечить оптимальное управление созданием и взаимодействием объектов.

Когда был создан язык ActionScript 1.0, использующие его программисты могли использовать объекты Function для создания конструктов, напоминающих классы. В версии ActionScript 2.0 была добавлена формальная поддержка для классов и введены ключевые слова, такие как `class` и `extends`. ActionScript 3.0 не только обеспечивает прежнюю поддержку ключевых слов, представленную в ActionScript 2.0, он также включает новые возможности. Например, ActionScript 3.0 включает усовершенствованное управление доступом с помощью атрибутов `protected` и `internal`. Кроме того, обеспечивается более усовершенствованное управление наследованием с помощью ключевых слов `final` и `override`.

Для разработчиков, имеющих опыт создания классов на таких языках программирования, как Java, C++ или C#, в ActionScript доступны знакомые функции. В ActionScript используется много аналогичных ключевых слов и имен атрибутов, таких как `class`, `extends` и `public`.

Примечание. *В документации Adobe ActionScript термин «свойство» обозначает любой член объекта или класса, включая переменные, константы и методы. Кроме того, несмотря на то, что термины «класс» и «статический член» нередко взаимозаменяются, в этом документе они обозначают разные понятия. Например, в этой главе словосочетание «свойства класса» относится ко всем членам класса, а не только к его статическим членам.*

Определения классов

В ActionScript 3.0 в определениях классов используется синтаксис, подобный тому, который использовался для определения классов в ActionScript 2.0. Правильный синтаксис для определения класса вызывает ключевое слово `class`, за которым следует имя класса. Тело класса, заключенное в фигурные скобки ({}), следует за именем класса. Например, следующий код создает класс с именем Shape, содержащий одну переменную `visible`.

```
public class Shape
{
    var visible:Boolean = true;
}
```

Одно значительное изменение в синтаксисе затрагивает определения классов, находящиеся внутри пакета. В ActionScript 2.0, если класс находится в пакете, имя пакета должно быть включено в объявление класса. В языке ActionScript 3.0, в котором вводится инструкция `package`, имя пакета должно быть включено в объявление пакета, а не в объявление класса. Например, следующие объявления классов показывают, как класс BitmapData, включенный в пакет flash.display, определяется в ActionScript 2.0 и ActionScript 3.0.

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
    public class BitmapData {}
}
```

Атрибуты классов

ActionScript 3.0 позволяет изменять определения классов с помощью одного из четырех атрибутов.

Атрибут	Определение
`dynamic`	Позволяет добавлять свойства для экземпляров во время выполнения.
`final`	Не должен расширяться другим классом.
`internal` (по умолчанию)	Видимый для ссылок внутри текущего пакета.
`public`	Видимый для всех ссылок.

Каждый из этих атрибутов, кроме `internal`, необходимо явно добавлять для получения соответствующего поведения. Например, если не включить атрибут `dynamic` в определение класса, то для его экземпляров нельзя будет добавлять свойства во время выполнения. Чтобы присвоить атрибут, необходимо поместить его в начало определения класса, как показано в следующем коде.

```
dynamic class Shape {}
```

Обратите внимание, что в список поддерживаемых атрибутов не входит `abstract`. Абстрактные классы не поддерживаются в ActionScript 3.0. Также обратите внимание на то, что в список не включены атрибуты `private` и `protected`. Эти атрибуты имеют значение только внутри определения класса и не могут применяться к самим классам. Если класс не должен открыто отображаться за пределами пакета, поместите его внутрь пакета и пометьте его атрибутом `internal`. В противном случае можно пропустить атрибуты `internal` и `public`, и компилятор автоматически добавит атрибут `internal`. Можно также определить класс, чтобы он был виден только в исходном файле, в котором он определен. Поместите класс в конец исходного файла ниже закрывающейся фигурной скобки определения пакета.

Тело класса

Тело класса заключается в фигурные скобки. В нем определяются переменные, константы и методы класса. В следующем примере показано объявление класса Accessibility в среде ActionScript 3.0:

```
public final class Accessibility
{
    public static function get active():Boolean;
    public static function updateProperties():void;
}
```

Также внутри тела класса можно определить пространство имен. В следующем примере показано, как можно определить в теле класса пространство имен и использовать его в качестве атрибута метода этого класса.

```
public class SampleClass
{
    public namespace sampleNamespace;
    sampleNamespace function doSomething():void;
}
```

ActionScript 3.0 позволяет включать в тело класса не только определения, но и инструкции. Операторы, которые находятся внутри тела класса, но не включены в определение метода, выполняются только один раз. Они выполняются, когда определение класса встречается в первый раз, и создается связанный класс. В следующем примере демонстрируется вызов внешней функции `hello()` и инструкция `trace`, которая выводит на экран подтверждение при определении класса.

```
function hello():String
{
    trace("hola");
}
class SampleClass
{
    hello();
    trace("class created");
}
// output when class is created
hola
class created
```

В ActionScript 3.0 можно определить одноименные статическое свойство и свойство экземпляра в одном теле класса. Например, следующий код объявляет статическую переменную с именем `message` и переменную экземпляра с тем же именем.

```
class StaticTest
{
    static var message:String = "static variable";
    var message:String = "instance variable";
}
// In your script
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // output: static variable
trace(myST.message); // output: instance variable
```

Атрибуты свойств классов

Когда речь идет об объектной модели ActionScript, термин *свойство* обозначает все, что может быть членом класса, включая переменные, константы и методы. Однако в справочной документации Adobe ActionScript 3.0 для платформы Adobe Flash Platform данный термин используется в более узком смысле. В этом контексте термин «свойство» включает только члены класса, которые являются переменными или определены в методе установки или получения. В ActionScript 3.0 имеется набор атрибутов, которые могут использоваться с любым свойством класса. Эти атрибуты перечислены в следующей таблице.

Атрибут	**Определение**
`internal` (по умолчанию)	Видимый для ссылок внутри того же пакета.
`private`	Видимый для ссылок в том же классе.
`protected`	Видимый для ссылок в том же классе и в производных классах.
`public`	Видимый для всех ссылок.
`static`	Показывает, что свойство принадлежит классу, а не экземплярам класса.
`UserDefinedNamespace`	Имя пространства имен, определенное пользователем.

Атрибуты пространства имен для управления доступом

ActionScript 3.0 обеспечивает четыре специальных атрибута для управления доступом к свойствам, определенным внутри класса: `public`, `private`, `protected` и `internal`.

Атрибут `public` делает свойство видимым во всем сценарии. Например, чтобы сделать метод доступным для кода за пределами пакета, необходимо объявить его с атрибутом `public`. Это относится ко всем свойствам независимо от того, с использованием каких ключевых слов они объявлены: `var`, `const` или `function`.

Атрибут `private` делает свойство видимым только для вызывающих методов внутри определяющего класса данного свойства. Это поведение отличает его от атрибута `private` в ActionScript 2.0, который разрешал подклассу доступ к закрытому свойству суперкласса. Другое значительное изменение поведение имеет отношение к доступу во время выполнения. В ActionScript 2.0 ключевое слово `private` запрещало доступ только во время компиляции, а во время выполнения этот запрет можно было легко обойти. В ActionScript 3.0 это уже не удастся. Свойства, помеченные атрибутом `private`, остаются недоступными как во время компиляции, так и во время выполнения.

Например, следующий код создает простой класс с именем PrivateExample с одной закрытой переменной, а затем к этой переменной пытается обратиться внешний метод.

```
class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // ActionScript 2.0 allows access, but in ActionScript 3.0, this
is a run-time error.
```

В ActionScript 3.0 попытка обращения к свойству с помощью оператора «точка» (`myExample.privVar`) приведет к ошибке компиляции, если используется строгий режим. В противном случае ошибка появляется во время выполнения, как и в случае использования оператора доступа к свойству (`myExample["privVar"]`).

В следующей таблице подведены результаты попыток обращения к закрытому свойству, которое принадлежит статическому (не динамическому) классу.

	Строгий режим	**Стандартный режим**
оператор «точка» (`.`)	ошибка компиляции	ошибка выполнения
оператор «квадратные скобки» (`[]`)	ошибка выполнения	ошибка выполнения

В классах, объявленных с использованием атрибута `dynamic`, попытки обращения к закрытым свойствам не приведут к ошибкам выполнения. Переменная не является видимой, поэтому возвращается значение `undefined`. Однако ошибка компиляции возникает, если в строгом режиме используется оператор «точка». Следующий пример похож на предыдущий и отличается от него только тем, что класс PrivateExample объявляется с атрибутом dynamic.

```
dynamic class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // output: undefined
```

Динамические классы обычно возвращают значение `undefined` вместо ошибки, когда внешний код пытается обратиться к закрытому свойству класса. В следующей таблице показано, что ошибка выдается, только когда оператор «точка» используется для обращения к закрытому свойству в строгом режиме.

	Строгий режим	**Стандартный режим**
оператор «точка» (`.`)	ошибка компиляции	`undefined`
оператор «квадратные скобки» (`[]`)	`undefined`	`undefined`

Атрибут `protected`, впервые введенный в версии ActionScript 3.0, делает свойство видимым для вызывающих методов собственного класса или подкласса. Другими словами, защищенное свойство доступно для собственного класса и подклассов, находящихся под ним в иерархии наследования. Оно доступно даже для подклассов, находящихся в других пакетах.

Для пользователей, знакомых со средой ActionScript 2.0, эта возможность аналогична атрибуту `private` в среде ActionScript 2.0. Атрибут `protected` среды ActionScript 3.0 также равнозначен атрибуту `protected` в Java. Он отличается от атрибута в версии Java тем, что также разрешает доступ к вызывающим объектам в том же пакете. Атрибут `protected` полезен, когда имеется свойство (переменная или метод), необходимое для подкласса, которое требуется скрыть для кода, находящегося за пределами цепи наследования.

Атрибут `internal`, впервые введенный в версии ActionScript 3.0, делает свойство видимым для вызывающих методов внутри своего пакета. Это атрибут по умолчанию для кода внутри пакета, он применяется ко всем свойствам, у которых нет ни одного из перечисленных ниже атрибутов:

- `public`;
- `private`;
- `protected`;
- пространство имен, определенное пользователем.

Атрибут `internal` похож на управление доступом по умолчанию в Java. Однако в Java нет выраженного имени для этого уровня доступа, и его можно реализовать только путем опущения других модификаторов доступа. Атрибут `internal` введен в ActionScript 3.0, чтобы можно было выразить намерение сделать свойство доступным только для вызывающих методов в его собственном пакете.

Атрибут static

Атрибут `static`, который может использоваться со свойствами, объявленными с использованием ключевых слов `var`, `const` или `function`, позволяет присоединить свойство к классу, а не к его экземплярам. Внешний код должен вызывать статические свойства с использованием имени класса, а не экземпляра.

Статические свойства не наследуются подклассами, но входят в цепочку области действия подкласса. Это означает, что в теле подкласса статическое свойство (переменная или метод) может использоваться без ссылки на класс, в котором оно было определено.

Дополнительные разделы справки

«Статические свойства не наследуются» на странице 119

Определенные пользователем атрибуты пространства имен

В качестве альтернативы предварительно определенным атрибутам управления доступом можно создать пользовательское пространство имен и использовать его как атрибут. Для одного определения можно использовать только один атрибут пространства имен. Кроме того, нельзя использовать атрибут пространства имен в сочетании с другими атрибутами управления доступом (`public`, `private`, `protected`, `internal`).

Дополнительные разделы справки

«Пространства имен» на странице 42

Переменные

Переменные могут объявляться с использованием ключевых слов `var` и `const`. Для переменных, объявленных с ключевым словом `var`, значения могут меняться несколько раз во время выполнения сценария. Переменные, объявленные с ключевым словом `const`, называются *константами*. Значение присваивается им только один раз. При попытке присвоить новое значение инициализированной константе выдается ошибка.

Статические переменные

Статические переменные объявляются с использованием комбинации ключевого слова `static` с инструкцией `var` или `const`. Статические переменные, присоединяемые к классу, а не к его экземпляру, полезны для хранения и совместного использования информации, которая применяется ко всему классу объектов. Например, статическую переменную уместно использовать, если требуется вести учет количества созданных экземпляров объекта или сохранить максимально допустимое количество экземпляров класса.

В следующем примере создается переменная `totalCount` для отслеживания количества экземпляров класса и константа`MAX_NUM` для хранения максимального числа экземпляров. Переменные `totalCount` и `MAX_NUM` являются статическими, так как содержат значения, применяемые ко всему классу, а не к отдельному экземпляру.

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

Код, внешний для класса StaticVars и его подклассов, может ссылаться на свойства `totalCount` и `MAX_NUM` только через сам класс. Например, выполняется следующий код:

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

К статическим переменным нельзя обратиться через экземпляр класса, поэтому следующий код возвращает ошибки.

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Переменные, объявленные с использованием ключевых слов `static` и `const`, должны инициализироваться одновременно с объявлением константы, что и делает класс StaticVars для `MAX_NUM`. Для константы `MAX_NUM` нельзя присвоить значение внутри конструктора или метода экземпляра. Следующий код создает ошибку, так как его недопустимо использовать для инициализации статической константы.

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Переменные экземпляра

Переменные экземпляра объявляются с использованием ключевых слов `var` и `const`, но без ключевого слова `static`. Переменные экземпляра, присоединяемые к отдельным экземплярам, а не ко всему классу, полезны для хранения значений, присущих экземпляру. Например, класс Array имеет свойство экземпляра с именем `length`, которое хранит количество элементов массива, содержащееся в определенном экземпляре класса Array.

Переменные экземпляра, объявленные с ключевыми словами `var` и `const`, нельзя переопределять в подклассе. Однако для получения функции, подобной переопределению переменных, можно переопределить методы доступа get и set.

Дополнительные разделы справки

«Константы» на странице 70

«Методы доступа get и set» на странице 104

Методы

Методы — это функции, которые входят в состав определения класса. Когда создается экземпляр класса, с ним связывается метод. В отличие от функции, объявленной за пределами класса, метод нельзя использовать отдельно от экземпляра, с которым он связан.

Методы определяются с использованием ключевого слова `function`. Как и любое свойство класса, можно применить любой атрибут свойства класса к методам, включая методы private, protected, public, internal, static или custom namespace. Можно использовать оператор function, аналогичный следующему:

```
public function sampleFunction():String {}
```

Также можно использовать переменную, которой присваивается выражение функции, как показано ниже.

```
public var sampleFunction:Function = function () {}
```

В большинстве случаев используйте оператор функции, а не выражение, потому что:

- Инструкции функции более точные и простые для чтения.
- Инструкции функции позволяют использовать ключевые слова `override` и `final`.
- Операторы функции создают более крепкую связь между идентификатором (именем функции) и кодом в теле метода. Так как значение переменной может быть изменено с помощью инструкции присваивания, связь между переменной и ее выражением функции может в любой момент быть разорвана. Хотя эту проблему можно обойти, объявив переменную с ключевым словом `const` вместо `var`, такой прием лучше не использовать, так как он делает код трудным для чтения и не позволяет использовать ключевые слова `override` и `final`.

Единственным случаем, когда необходимо использовать именно выражение функции, является присоединение функции к объекту прототипа.

Методы-конструкторы

Методы-конструкторы (иногда их называют просто *конструкторами*) представляют собой функции с тем же именем, которым назван класс, в котором они определены. Любой код, включенный в метод-конструктор, выполняется при каждом создании экземпляра класса с использованием ключевого слова `new`. Например, следующий код определяет простой класс Example, содержащий одно свойство с именем `status`. Начальное значение переменной `status` задается в функции конструктора.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Методы-конструкторы могут быть только открытыми, но использование атрибута `public` является необязательным. Для конструктора нельзя использовать никаких других спецификаторов управления доступом, включая `private`, `protected` или `internal`. Кроме того, с методом-конструктором нельзя использовать пользовательское пространство имен.

Конструктор может явно вызвать конструктор находящегося непосредственно над ним суперкласса с помощью инструкции `super()`. Если конструктор суперкласса не вызывается явно, компилятор автоматически вставляет вызов перед первой инструкцией в теле конструктора. Также вызывать методы суперкласса можно, используя префикс `super` в качестве ссылки на суперкласс. Если в теле одного конструктора требуется использовать и конструктор `super()`, и префикс `super`, первым должен идти конструктор `super()`. В противном случае ссылка `super` может повести себя не так, как предполагалось. Конструктор `super()` также необходимо вызывать перед инструкциями `throw` или `return`.

В следующем примере показано, что происходит при попытке использования ссылки `super` перед вызовом конструктора `super()`. Новый класс, ExampleEx, расширяет класс Example. Конструктор ExampleEx пытается получить переменную status, определенную в его суперклассе, но делает это перед вызовом `super()`. Инструкция `trace()` в конструкторе ExampleEx возвращает значение `null`, так как переменная `status` недоступна, пока не будет вызван конструктор `super()`.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Хотя в конструкторе можно использовать инструкцию `return`, он не может вернуть значение. Другими словами, инструкции `return` не должны иметь связанных выражений или значений. Соответственно, методы-конструкторы не могут возвращать значения, вследствие чего невозможно указать тип возврата.

Если для класса не определить метод-конструктор, компилятор автоматически создает пустой конструктор. Если данный класс расширяет другой, компилятор включает вызов `super()` в созданный им конструктор.

Статические методы

Статические методы, также называемые *методами класса* — это методы, которые объявляются с использованием ключевого слова `static`. Статические методы, присоединяемые к классу, а не к его экземпляру, полезны для инкапсуляции, которая затрагивает не только состояние отдельного экземпляра. Так как статические методы присоединяются ко всему классу, их можно вызвать только через класс, а через его экземпляр — нет.

Статические методы полезны для инкапсуляции, действие которой не ограничивается состоянием экземпляров класса. Другими словами, метод должен быть статическим, если он обеспечивает функцию, которая не оказывает прямого влияния на значение экземпляра класса. Например, класс Date имеет статический метод `parse()`, который преобразует строку в число. Это статический метод, так как он не оказывает влияния на отдельные экземпляры класса. Вместо этого метод `parse()` берет строку, которая представляет собой значение даты, анализирует ее и возвращает число в формате, совместимом с внутренним представлением объекта Date. Этот метод не является методом экземпляра, так как его бессмысленно применять к одному экземпляру класса Date.

Сравните статический метод `parse()` с одним из методов экземпляра класса Date, таким как `getMonth()`. Метод `getMonth()` является методом экземпляра, так как он воздействует непосредственно на значение экземпляра, получая определенный компонент (месяц) экземпляра Date.

Так как статические методы не связаны с отдельными экземплярами, в их теле нельзя использовать ключевые слова `this` и `super`. Ссылки `this` и `super` имеют значение, только в контексте метода экземпляра.

В отличие от других языков программирования на базе классов, в ActionScript 3.0 статические методы не наследуются.

Методы экземпляра

Методами экземпляра являются методы, объявленные без ключевого слова `static`. Методы экземпляра, присоединяемые к экземплярам, а не ко всему классу, полезны для реализации функций, затрагивающих отдельные экземпляры класса. Например, класс Array содержит метод экземпляра `sort()`, который воздействует непосредственно на экземпляры Array.

В теле метода экземпляра могут содержаться и статические переменные, и переменные экземпляра. Это означает, что на переменные, определенные в одном классе, можно ссылаться с использованием простого идентификатора. Например, следующий класс, CustomArray, расширяет класс Array. Класс CustomArray определяет статическую переменную `arrayCountTotal` для отслеживания общего числа экземпляров класса, переменную экземпляра `arrayNumber`, отслеживающую порядок создания экземпляров, и метод экземпляра `getPosition()`, возвращающий значения этих переменных.

```
public class CustomArray extends Array
{
    public static var arrayCountTotal:int = 0;
    public var arrayNumber:int;

    public function CustomArray()
    {
        arrayNumber = ++arrayCountTotal;
    }

    public function getArrayPosition():String
    {
         return ("Array " + arrayNumber + " of " + arrayCountTotal);
    }
}
```

Хотя внешний код класса должен обращаться к статической переменной `arrayCountTotal` через объект класса с помощью `CustomArray.arrayCountTotal`, код, находящийся внутри тела метода `getPosition()`, может ссылаться непосредственно на статическую переменную `arrayCountTotal`. Это относится даже к статическим переменным суперклассов. Хотя статические свойства не наследуются в ActionScript 3.0, статические свойства суперклассов входят в область действия. Например, класс Array имеет несколько статических переменных, одна из которых является константой с именем `DESCENDING`. Код, находящийся в подклассе Array, может обращаться к статической константе `DESCENDING` с помощью простого идентификатора:

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

Значением ссылки `this` в теле метода экземпляра является ссылка на экземпляр, к которому присоединен метод. Следующий код демонстрирует, как ссылка `this` указывает на экземпляр, содержащий метод.

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

Наследованием методов экземпляров можно управлять с помощью ключевых слов `override` и `final`. Атрибут `override` можно использовать для переопределения унаследованного метода, а атрибут `final` — для запрета переопределения метода подклассами.

Методы доступа get и set

Методы доступа get и set, также называемые *getter* и *setter*, позволяют соблюдать принципы сокрытия информации и инкапсуляции, обеспечивая простой в использовании интерфейс программирования для создаваемых классов. Функции get и set позволяют делать свойства класса закрытыми, но при этом пользователи класса могут обращаться к этим свойствам так, будто они обращаются к переменной класса, а не вызывают метод класса.

Преимущество такого подхода заключается в том, что он позволяет отказаться от традиционных функций методов доступа с громоздкими именами, такими как `getPropertyName()` и `setPropertyName()`. Другое преимущество методов get и set состоит в том, что они позволяют избежать создания двух открытых функций для каждого свойства, разрешающего доступ для чтения и записи.

Следующий пример класса GetSet включает методы доступа get и set с именем `publicAccess()`, которые обеспечивают доступ к закрытой переменной `privateProperty`.

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

Если обратиться к свойству `privateProperty` напрямую, возникает ошибка, как показано ниже.

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

Вместо этого пользователь класса GetSet использует свойство, похожее на свойство `publicAccess`, но на самом деле являющееся парой функций методов доступа get и set, которые работают с закрытым свойством `privateProperty`. В следующем примере создается экземпляр класса GetSet, а затем задается значение свойства `privateProperty` с помощью открытого метода доступа `publicAccess`.

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Функции get и set также позволяют переопределять свойства, унаследованные от суперкласса, что невозможно выполнить при использовании обычных переменных-членов класса. Переменные-члены класса, объявленные с использованием ключевого слова `var`, не могут переопределяться в подклассе. Однако это ограничение не распространяется на свойства, созданные с помощью функций get и set. Для функций get и set, унаследованных от суперкласса, можно использовать атрибут `override`.

Связанные методы

Связанный метод, иногда называемый *замыканием метода* — это метод, извлеченный из своего экземпляра. К числу связанных методов относятся методы, переданные функции в качестве аргументов или возвращенные функцией в качестве значений. Впервые в ActionScript 3.0 связанный метод уподобляется замыканию функции в том, что он сохраняет всю свою лексическую среду даже при извлечении из экземпляра. Однако между связанным методом и замыканием функции есть важное отличие: ключевое слово `this` для связанного метода продолжает ссылаться на экземпляр, в котором реализуется метод, то есть остается связанным с ним. Другими словами, ссылка `this` в связанном методе всегда указывает на исходный объект, в котором определен метод. Для замыканий функций ссылка `this` является общей, то есть она указывает на любой объект, с которым связана функция в момент вызова.

Важно понимать суть связанных методов, если используется ключевое слово `this`. Как вы помните, ключевое слово `this` обеспечивает ссылку на родительский объект метода. Большинство программистов, использующих ActionScript, считают, что ключевое слово `this` всегда представляет объект или класс, содержащий определение метода. Однако это не всегда так, если не применяется связывание метода. Например, в предыдущих версиях ActionScript ключевое слово `this` не всегда ссылается на экземпляр, в котором реализован метод. Когда методы извлекаются из экземпляра в ActionScript 2.0, ссылка `this` не связывается с исходным экземпляром, а переменные и методы экземпляра становятся недоступными. В ActionScript 3.0 этого не происходит, так как при передаче методов в качестве параметра автоматически создаются связанные методы. Связанные методы обеспечивают, чтобы ключевое слово `this` всегда ссылалось на объект или класс, в котором определяется метод.

Следующий код определяет класс ThisTest, который содержит метод `foo()`, определяющий связанный метод, и метод `bar()`, возвращающий связанный метод. Внешний код создает экземпляр класса ThisTest, создает метод `bar()` и сохраняет возвращенное значение в переменной `myFunc`.

```
class ThisTest
{
    private var num:Number = 3;
    function foo():void // bound method defined
    {
        trace("foo's this: " + this);
        trace("num: " + num);
    }
    function bar():Function
    {
        return foo; // bound method returned
    }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
output: num: 3 */
```

Две последние строки кода показывают, что ссылка `this` в связанном методе `foo()` по-прежнему указывает на экземпляр класса ThisTest, несмотря на то, что ссылка `this` в предшествующей строке указывает на глобальный объект. Более того, связанный метод, сохраненный в переменной `myFunc`, сохранил доступ к переменным-членам класса ThisTest. Если выполнить этот же код в ActionScript 2.0, ссылки `this` будут одинаковыми, а переменная `num` получит значение `undefined`.

Добавление связанных методов имеет самое большое значение, когда речь идет об обработчиках событий, так как метод `addEventListener()` требует, чтобы в качестве аргумента передавалась функция или метод.

Перечисления с классами

Перечисления — это пользовательские типы данных, которые создаются для инкапсуляции небольшого набора значений. Язык ActionScript 3.0 не поддерживает специфическую функцию перечисления, в отличие от языка C++, в котором есть ключевое слово`enum`, или Java, где предусмотрен интерфейс Enumeration. Однако перечисления можно создавать с помощью классов и статических констант. Например, класс PrintJob в ActionScript 3.0 использует перечисление PrintJobOrientation для хранения значений `"landscape"` (альбомная) и `"portrait"` (книжная), как показано в следующем коде:

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

Класс перечисления условно объявляется с использованием атрибута `final`, так как нет необходимости расширять класс. Класс включает только статические члены, а это означает, что экземпляры класса не создаются. Вместо этого, объект класса обеспечивает прямой доступ к значениям перечисления, как показано в следующем фрагменте кода.

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Все классы перечисления в ActionScript 3.0 содержат только переменные типа String, int или uint. Преимущество использования перечислений вместо буквенной строки или числовых значений заключается в том, что в перечислениях проще выявить опечатки. Если имя перечисления введено с ошибкой, компилятор ActionScript выдает ошибку. При использовании строк компилятор не сообщает о неправильном написании слова или числа. В предыдущем примере компилятор выдает ошибку, если введено неправильное имя для константы перечисления, как показано в следующем фрагменте.

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

Однако компилятор не выдает ошибку, если неправильно введено строковое значение, как показано ниже.

```
    if (pj.orientation == "portrai") // no compiler error
```

Второй метод создания перечислений также подразумевает создание отдельного класса со статическими свойствами для перечисления. Однако этот метод отличается тем, что каждое из статических свойств, содержит экземпляр класса вместо строки или целого числа. Например, следующий код создает класс перечисления для дней недели.

```
public final class Day
{
    public static const MONDAY:Day = new Day();
    public static const TUESDAY:Day = new Day();
    public static const WEDNESDAY:Day = new Day();
    public static const THURSDAY:Day = new Day();
    public static const FRIDAY:Day = new Day();
    public static const SATURDAY:Day = new Day();
    public static const SUNDAY:Day = new Day();
}
```

Этот метод не используется в ActionScript 3.0, но используется многими разработчиками, которые предпочитают его улучшенную проверку написания. Например, метод, возвращающий значение перечисления, может ограничить возвращаемое значение определенным типом данных перечисления. Следующий код демонстрирует функцию, возвращающую день недели, а также вызов функции, использующий тип перечисления в качестве аннотации типа.

```
function getDay():Day
{
    var date:Date = new Date();
    var retDay:Day;
    switch (date.day)
    {
        case 0:
            retDay = Day.MONDAY;
            break;
        case 1:
            retDay = Day.TUESDAY;
            break;
        case 2:
            retDay = Day.WEDNESDAY;
            break;
        case 3:
            retDay = Day.THURSDAY;
            break;
        case 4:
            retDay = Day.FRIDAY;
            break;
        case 5:
            retDay = Day.SATURDAY;
            break;
        case 6:
            retDay = Day.SUNDAY;
            break;
    }
    return retDay;
}

var dayOfWeek:Day = getDay();
```

Кроме того, класс Day можно расширить, чтобы он связывал целое число с каждым днем недели и определял метод `toString()`, возвращающий строковое представление дня.

Классы встроенных ресурсов

В языке ActionScript 3.0 используются особые классы для представления встроенных ресурсов, называемые *классами встроенных ресурсов*. *Встроенный ресурс* — это ресурс, такой как звук, изображение или шрифт, включенный в SWF-файл во время компиляции. Встраивание ресурса вместо его динамической загрузки гарантирует его доступность во время выполнения, однако за счет увеличения размера SWF-файла.

Использование классов встроенных ресурсов в инструменте Flash Professional

Чтобы встроить ресурс, сначала поместите его в библиотеку FLA-файла. Затем используйте свойство связывания ресурса, чтобы передать имя для его класса встроенных ресурсов. Если класса с таким именем нет в пути к классам, он создается автоматически. После этого можно создать экземпляр класса встроенных ресурсов и использовать любые определенные или унаследованные им свойства и методы. Например, следующий код можно использовать для воспроизведения встроенного звука, связанного с классом встроенных ресурсов PianoMusic.

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

В противном случае можно использовать тег метаданных `[Embed]` для встраивания ресурсов в проект Flash Professional, как описано далее. Если в коде используется тег метаданных `[Embed]`, в инструменте Flash Professional для компиляции проекта используется компилятор Flex, а не компилятор Flash Professional.

Использование классов встроенных ресурсов с помощью компилятора Flex

При компиляции кода с помощью компилятора Flex для встраивания ресурса в код ActionScript используйте тег метаданных `[Embed]`. Поместите ресурс в основной исходной папке или в другой папке, включенной в путь сборки проекта. Когда компилятор Flex обнаружит тег метаданных Embed, он создаст класс встроенных ресурсов. Обратиться к классу можно через переменную типа данных Class, объявленную сразу после тега метаданных `[Embed]`. Например, следующий код встраивает звук с именем sound1.mp3 и использует переменную `soundCls` для хранения ссылки на класс встроенных ресурсов, связанный с этим звуком. После этого создается экземпляр класса встроенных ресурсов, для которого вызывается метод`play()`.

```
package
{
    import flash.display.Sprite;
    import flash.media.SoundChannel;
    import mx.core.SoundAsset;

    public class SoundAssetExample extends Sprite
    {
        [Embed(source="sound1.mp3")]
        public var soundCls:Class;

        public function SoundAssetExample()
        {
            var mySound:SoundAsset = new soundCls() as SoundAsset;
            var sndChannel:SoundChannel = mySound.play();
        }
    }
}
```

Adobe Flash Builder

Чтобы использовать тег метаданных `[Embed]` в проекте Flash Builder ActionScript, импортируйте все необходимые классы из среды Flex. Например, чтобы встроить звуки, импортируйте класс mx.core.SoundAsset. Чтобы использовать систему Flex, включите файл framework.swc в путь сборки ActionScript. Это приводит к увеличению размера SWF-файла.

Adobe Flex

В противном случае в среде Flex ресурс можно встроить с помощью директивы `@Embed()` в определении тега MXML.

Интерфейсы

Интерфейс — это набор объявлений методов, который позволяет несвязанным объектам взаимодействовать друг с другом. Например, ActionScript 3.0 определяет интерфейс IEventDispatcher, который содержит объявления методов, которые класс может использовать для обработки событий. Интерфейс IEventDispatcher устанавливает стандартную процедуру передачи событий между объектами. Следующий код демонстрирует определение интерфейса IEventDispatcher.

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
            useCapture:Boolean=false, priority:int=0,
            useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
            useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

Интерфейсы основаны на различии между интерфейсом метода и его реализацией. Интерфейс метода включает всю информацию, необходимую для вызова этого метода, включая имя метода, все его параметры и тип возвращаемого значения. Реализация метода включает не только сведения об интерфейсе, но и исполняемые инструкции, определяющие поведение метода. Определение интерфейса содержит только интерфейсы метода, и класс, внедряющий интерфейс, несет ответственность за определение реализаций метода.

В ActionScript 3.0 класс EventDispatcher реализует интерфейс IEventDispatcher, определяя все методы интерфейса IEventDispatcher и добавляя соответствующие тела методов. Следующий код представляет собой фрагмент определения класса EventDispatcher.

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

Интерфейс IEventDispatcher служит в качестве протокола, который используют экземпляры EventDispatcher для обработки событий и их отправки другим объектам, также внедрившим этот интерфейс.

Интерфейс также можно описать как элемент, определяющий тип данных так же, как и класс. Соответствующим образом, интерфейс можно использовать в качестве аннотации типа, как и класс. В качестве типа данных, интерфейс можно также использовать вместе с операторами, требующими тип данных, такими как `is` и `as`. Однако, в отличие от класса, интерфейс не поддерживает создание экземпляров. Из-за этого отличия многие программисты называют интерфейсы абстрактными типами данных, а классы — конкретными типами данных.

Определение интерфейса

Структура определения интерфейса похожа на структуру определения класса за тем исключением, что интерфейс может содержать только методы без тел. Интерфейсы не могут содержать переменных или констант, но могут включать функции get и set. Для определения интерфейса используется ключевое слово `interface`. Например, следующий интерфейс, IExternalizable, входит в пакет flash.utils в ActionScript 3.0. Интерфейс IExternalizable определяет протокол для сериализации объекта, то есть для преобразования объекта в формат, пригодный для хранения на устройстве или для транспортировки по сети.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

Интерфейс IExternalizable объявляется с использованием модификатора управления доступом `public`. Определения интерфейса можно изменить только с использованием спецификаторов управления доступом `public` и `internal`. Объявления методов внутри определения интерфейса не могут иметь модификаторов управления доступом.

В ActionScript 3.0 используется правило именования интерфейсов с заглавной буквы `I`, однако в качестве имени интерфейса можно использовать любой допустимый идентификатор. Определения интерфейсов часто помещаются в верхнем уровне пакета. Определения интерфейсов нельзя помещать внутри определения класса или определения другого интерфейса.

Интерфейс может расширять один или несколько других интерфейсов. Например, следующий интерфейс IExample расширяет интерфейс IExternalizable.

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Любой класс, внедряющий интерфейс IExample, должен включать реализации не только для метода `extra()`, но также методы `writeExternal()` и `readExternal()`, унаследованные от интерфейса IExternalizable.

Реализация интерфейса в классе

Класс является единственным элементом языка ActionScript 3.0, который может реализовать интерфейс. Используйте ключевое слово `implements` в объявлении класса, чтобы реализовать один или несколько интерфейсов. В следующем примере определяется два интерфейса (IAlpha и IBeta), а также реализующий их класс Alpha.

```
interface IAlpha
{
    function foo(str:String):String;
}

interface IBeta
{
    function bar():void;
}

class Alpha implements IAlpha, IBeta
{
    public function foo(param:String):String {}
    public function bar():void {}
}
```

В классе, внедряющем интерфейс, реализованные методы должны выполнять следующие действия:

- использовать идентификатор управления доступом `public`;
- использовать то же имя, что и метод интерфейса;
- иметь столько же параметров, сколько и методы интерфейса, а их типы данных должны соответствовать типам данных параметров методов интерфейса;
- использовать тот же тип возвращаемых значений.

```
public function foo(param:String):String {}
```

Однако вам предоставляется некоторая свобода в наименовании параметров для реализуемых методов. Хотя реализуемый метод должен иметь столько же параметров и с теми же типами данных, что и метод интерфейса, имена параметров могут быть разными. Например, в предыдущем примере параметру метода `Alpha.foo()` было присвоено имя `param`.

А в методе интерфейса `IAlpha.foo()` он называется `str`.

```
function foo(str:String):String;
```

Также предоставляется некоторая гибкость в отношении значений параметров по умолчанию. Определение интерфейса можно включать объявления функций со значениями параметров по умолчанию. Для метода, реализующего такое объявление функции, параметра должен иметь значение по умолчанию, являющееся членом того же типа данных, что и значение, заданное в определении интерфейса, но само значение может быть другим. Например, следующий код определяет интерфейс, содержащий метод со значением параметра по умолчанию 3.

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

Следующее определение класса реализует интерфейс IGamma, но использует другое значение параметра по умолчанию:

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

Такая гибкость обусловлена тем, что правила реализации интерфейса разработаны таким образом, чтобы обеспечить совместимость типа данных, а для достижения этой цели использование идентичных имен и значения по умолчанию для параметров не является обязательным.

Наследование

Наследование — это форма повторного использования кода, позволяющая программистам разрабатывать новые классы на основе существующих. Существующие классы часто называются *базовыми классами* или *суперклассами*, а новые классы — *подклассами*. Основным преимуществом наследования является то, что оно позволяет повторно использовать код из базового класса, не изменяя первоисточник. Более того, при наследовании не требуется изменять метод взаимодействия других классов с базовым. В отличие от изменения существующего класса, который уже тщательно протестирован и, возможно, используется, наследование позволяет работать с классом как с интегрированным модулем, который можно расширить с помощью дополнительных свойств или методов. Таким образом, для обозначения класса, наследующего свойства и методы другого класса используется ключевое слово `extends` (расширяет).

Наследование также позволяет использовать в коде преимущества *полиморфизма*. Полиморфизм — это возможность использовать одно имя для метода, который ведет себя по-разному при применении к разным типам данных. Самый простой пример — это базовый класс Shape, который имеет два подкласса с именами Circle и Square. Класс Shape определяет метод `area()`, который возвращает площадь фигуры. При применении полиморфизма метод `area()` можно вызывать для объектов типа Circle и Square, получая при этом правильные расчеты. Наследование делает возможным полиморфизм, так как позволяет подклассам наследовать и переопределять (*override*) методы из базового класса. В следующем примере метод `area()` переопределяется классами Circle и Square.

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

Так как каждый класс определяет тип данных, наследование создает определенные отношения между базовым и расширяющим классами. Подкласс гарантированно получает все свойства базового класса, то есть экземпляр подкласса можно в любой момент заменить экземпляр базового класса. Например, если метод определяет параметр типа Shape, разрешается передать аргумент типа Circle, так как класс Circle расширяет класс Shape, как показано в следующем примере.

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Свойства экземпляра и наследование

Свойство экземпляра, определенное с использованием ключевых слов `function`, `var` или `const`, наследуется всеми подклассами, если оно не объявлено с атрибутом `private` в базовом классе. Например, класс Event в ActionScript 3.0 имеет ряд подклассов, наследующих свойства, общие для всех объектов событий.

Для некоторый типов событий класс Event содержит все свойства, необходимые для определения события. Эти типы событий не требуют других свойств экземпляра, кроме определенных в классе Event. К таким событиям относятся событие `complete`, которое происходит после успешной загрузки данных, и событие `connect`, которое происходит после успешной установки соединения.

В следующем примере приводится фрагмент из кода класса Event, в котором представлены некоторые свойства и методы, наследуемые подклассами. Так как свойства наследуются, к ним может обращаться экземпляр любого подкласса.

```
public class Event
{
    public function get type():String;
    public function get bubbles():Boolean;
    ...

    public function stopPropagation():void {}
    public function stopImmediatePropagation():void {}
    public function preventDefault():void {}
    public function isDefaultPrevented():Boolean {}
    ...
}
```

Для других типов событий требуются уникальные свойства, недоступные в классе Event. Эти события определяются с использованием подклассов класса Event, чтобы к числу свойств, определенных в классе Event, можно было добавить новые свойства. В качестве примера можно привести подкласс MouseEvent, который добавляет свойства, уникальные для событий, связанных с движением или щелчками мыши, таких как `mouseMove` и `click`. В следующем примере приводится фрагмент кода класса MouseEvent, в котором определяются свойства, существующие только в подклассе, а в базовом классе — нет.

```
public class MouseEvent extends Event
{
    public static const CLICK:String= "click";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...

    public function get stageX():Number {}
    public function get stageY():Number {}
    ...
}
```

Идентификаторы управления доступом и наследование

Если свойство объявлено с использованием ключевого слова `public`, оно будет доступно для всего кода. Это означает, что ключевое слово `public`, в отличие от слов `private`, `protected` и `internal`, не накладывает ограничений на наследование свойств.

Если свойство объявлено с использованием ключевого слова `private`, оно доступно только в определяющем его классе, то есть не наследуется никакими подклассами. Такое поведение отличается от предыдущих версий ActionScript, где поведение ключевого слова `private` больше напоминало поведение `protected` в ActionScript 3.0.

Ключевое слово `protected` указывает, что свойство доступно не только для определяющего его класса, но и для всех подклассов. В отличие от ключевого слова `protected` в языке программирования Java, в ActionScript 3.0 `protected` не делает свойство доступным для всех других классов в одном пакете. В ActionScript 3.0 свойство, объявленное с использованием ключевого слова `protected` доступно только для подклассов. Более того, защищенное свойство доступно для подкласса независимо от того, находится он в том же пакете, что и базовый класс, или в другом.

Чтобы ограничить доступность свойства только тем пакетом, в котором оно определено, следует использовать только ключевое слово `internal` без других идентификаторов управления доступом. Идентификатор управления доступом `internal` используется по умолчанию, когда нет других атрибутов. Свойство, отмеченное как `internal`, наследуется только подклассом, который находится в том же пакете.

В следующем примере демонстрируется влияние каждого из идентификаторов управления доступом на наследование в пределах пакета. Следующий код определяет основной класс приложения AccessControl и два других класса с именами Base и Extender. Класс Base находится в пакете foo, а его подкласс Extender находится в пакете bar. Класс AccessControl импортирует только класс Extender и создает экземпляр Extender, который пытается обратиться к переменной `str`, определенной в классе Base. Переменная `str` объявляется с идентификатором `public`, поэтому код компилируется и выполняется, как показано в следующем фрагменте.

```
// Base.as in a folder named foo
package foo
{
    public class Base
    {
        public var str:String = "hello"; // change public on this line
    }
}

// Extender.as in a folder named bar
package bar
{
    import foo.Base;
    public class Extender extends Base
    {
        public function getString():String {
            return str;
        }
    }
}

// main application class in file named AccessControl.as
package
{
    import flash.display.MovieClip;
    import bar.Extender;
    public class AccessControl extends MovieClip
    {
        public function AccessControl()
        {
            var myExt:Extender = new Extender();
            trace(myExt.str);// error if str is not public
            trace(myExt.getString()); // error if str is private or internal
        }
    }
}
```

Чтобы оценить влияние других идентификаторов управления доступом на компиляцию и выполнения предыдущего примера, измените атрибут переменной `str` на `private`, `protected` или `internal`, удалив или скрыв с помощью комментария следующую строку из класса `AccessControl`:

```
trace(myExt.str);// error if str is not public
```

Запрет переопределения переменных

Свойства, объявленные с ключевыми словами `var` и `const` наследуются, но не могут быть переопределены. Переопределение свойства выполняется в подклассе. Единственными типами свойств, допускающими переопределение, являются средства доступа get и set (свойства, объявленные с ключевым словом `function`). Хотя нельзя переопределить переменную экземпляра, можно получить подобный результат путем создания методов get и set для переменной экземпляра и переопределения этих методов.

Переопределение методов

Переопределение метода означает переопределение поведения наследуемого метода. Статические методы не наследуются и не могут переопределяться. Однако методы экземпляра наследуются подклассами и могут переопределяться, если выполняется два следующих условия:

- Метод экземпляра объявлен в базовом классе без использования ключевого слова `final`. Атрибут `final` рядом с методом экземпляра указывает на то, что программист явно запретил переопределение метода подклассами.
- Метод экземпляра объявлен в базовом классе без использования идентификатора управления доступом `private`. Если метод отмечен как `private` в базовом классе, то при определении одноименного метода в подклассе не требуется использовать ключевое слово `override`, так как метод базового класса не будет доступен для подкласса.

  Чтобы переопределить метод экземпляра, соответствующий этим критериям, в определении метода подкласса должно использоваться ключевое слово `override` и соблюдаться соответствие версии суперкласса по следующим аспектам:

- Переопределенный метод должен иметь тот же уровень управления доступом, что и метод базового класса. Методы с атрибутом internal должны иметь тот же уровень управления доступом, что и методы без идентификатора.
- Переопределенный метод должен иметь столько же параметров, что и метод базового класса.
- Параметры переопределенного метода должны иметь те же аннотации типа данных, что и параметры в метода базового класса.
- Переопределенный метод должен иметь от же тип возвращаемого значения, что и метод базового класса.

Однако параметры в переопределенном методе и параметры в базовом классе могут иметь разные имена, при условии что число параметров и тот же тип данных совпадают.

Инструкция super

Переопределяя метод, программисты нередко хотят расширить поведение метода суперкласса, а не заменить его полностью. Для этого требуется механизм, позволяющий методу подкласса вызывать версию этого же метода в суперклассе. Инструкция `super` обеспечивает такой механизм тем, что содержит ссылку на непосредственный суперкласс. В следующем примере определяется класс Base, содержащий метод `thanks()`, и его подкласс Extender, переопределяющий метод `thanks()`. Метод `Extender.thanks()` использует инструкцию `super` для вызова `Base.thanks()`.

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

Переопределение методов get и set

Несмотря на то, что нельзя переопределять переменные суперкласса, можно переопределять методы get и set. Например, следующий код переопределяет метод get `currentLabel`, определенный в классе MovieClip в ActionScript 3.0.

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

Инструкция `trace()` в конструкторе класса OverrideExample выдает значение `Override: null`, которое указывает на то, что код смог переопределить унаследованное свойство `currentLabel`.

Статические свойства не наследуются

Статические свойства не наследуются подклассами Это означает, что к статическим свойствам невозможно обратиться через экземпляр подкласса. Статическое свойство можно получить только через объект определяющего его класса. Например, следующий код определяет базовый класс Base и его подкласс Extender. Статическая переменная `test` определена в классе Base. Код, представленный в следующем фрагменте, не компилируется в строгом режиме и выдает ошибку выполнения в стандартном режиме.

```
package {
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// error
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base { }
```

Обратиться к статической переменной `test` можно только через объект класса, как показано в следующем коде.

```
Base.test;
```

Однако можно определить свойство экземпляра с тем же именем, что и статическое свойство. Такое свойство экземпляра может быть определено в том же классе, что и статическое, или в подклассе. Например, в классе Base в предыдущем примере можно было бы определить свойство экземпляра `test`. Следующий код компилируется и выполняется без ошибок, так как класс Extender наследует свойство экземпляра. Этот код также будет компилироваться и выполняться, если определение переменной экземпляра test будет перемещено, а не скопировано, в класс Extender.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

Статические свойства и цепочка области действия

Несмотря на то, что статические свойства не наследуются, они находятся в цепочке области действия определяющего их класса, а также его подклассов. Поэтому статические свойства, так сказать, находятся *в области действия* и класса, в котором они определены, и его подклассов. Это означает, что статическое свойство напрямую доступно в теле определяющего его класса и всех его подклассов.

В следующем примере модифицируются классы, определенные в предыдущем примере, чтобы продемонстрировать то, что статическая переменная `test`, определенная в классе Base, находится в области действия класса Extender. Другими словами, класс Extender может обращаться к статической переменной `test`, не добавляя к ней в качестве префикса имя класса, определяющего `test`.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base
{
    public function Extender()
    {
        trace(test); // output: static
    }

}
```

Если в одном классе или суперклассе определено свойство экземпляра с тем же именем, что и статическое свойство, то свойство экземпляра имеет более высокий приоритет в цепочке области действия. Свойство экземпляра, так сказать, *затеняет* статическое свойство, то есть значение свойства экземпляра используется вместо значения статического свойства. Например, следующий код демонстрирует, что если класс Extender определяет переменную экземпляра с именем `test`, инструкция `trace()` использует ее значение вместо значения статической переменной.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base
{
    public static var test:String = "static";
}

class Extender extends Base
{
    public var test:String = "instance";
    public function Extender()
    {
        trace(test); // output: instance
    }

}
```

Дополнительные темы

История объектно-ориентированного программирования на языке ActionScript

Так как язык ActionScript 3.0 создан на основе предыдущих версий ActionScript, будет полезно проследить развитие объектной модели ActionScript. Язык ActionScript сначала представлял собой простой механизм создания скриптов для ранних версий Flash Professional. Впоследствии программисты стали создавать с помощью ActionScript более сложные приложения. В ответ на потребности таких программистов в каждом новом выпуске расширялись возможности языка, упрощающие создание сложных приложений.

ActionScript 1.0

ActionScript 1.0 — это версия языка, используемого во Flash Player 6 и более ранних версиях. С самого начала объектная модель языка ActionScript строилась на базе концепции, согласно которой объект является основополагающим типом данных. Объект ActionScript представляет собой составной тип данных с группой *свойств*. Применительно к объектной модели термин *свойства* включает все, что присоединяется к объекту, то есть переменные, функции или методы.

Хотя первое поколение языка ActionScript не поддерживает определение классов с помощью ключевого слова `class`, класс можно определить с помощью объекта особого типа, так называемого прототипа. Вместо того чтобы определять с помощью ключевого слова `class` абстрактного класса, из которого затем можно создать конкретные экземпляры, как в языках на базе классов, таких как Java и C++, языки на базе прототипов, такие как ActionScript 1.0, используют в качестве модели (или прототипа) для новых объектов существующий объект. В языках на базе классов объект может указывать на класс, используемый в качестве его шаблона, а в языках на базе прототипа объект указывает на другой объект, свой прототип, который является его шаблоном.

Чтобы создать класс в ActionScript 1.0, необходимо определить функцию-конструктор для этого класса. В ActionScript функции — это не абстрактные определения, а конкретные объекты. Созданная функция-конструктор служит в качестве прототипичного объекта для экземпляров этого класса. Следующий код создает класс Shape и определяет одно свойство `visible`, которому задается значение по умолчанию `true`.

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Эта функция-конструктор определяет класс Shape и создает его экземпляр с помощью оператора `new`, как показано ниже.

```
myShape = new Shape();
```

Объект функции-конструктора `Shape()` может служить прототипом для экземпляров не только класса Shape, но и его подклассов, то есть классов, расширяющих класс Shape.

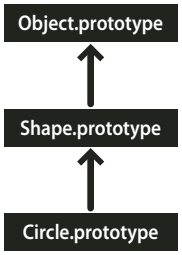
Создание подкласса для класса Shape происходит в два этапа. На первом этапе создается класс посредством определения его функции-конструктора, как показано ниже.

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

На втором этапе с помощью оператора `new` класс Shape объявляется в качестве прототипа класса Circle. По умолчанию любой создаваемый класс использует в качестве прототипа класс Object, то есть `Circle.prototype` пока содержит родовой объект (экземпляр класса Object). Чтобы указать, что прототипом класса Circle является класс Shape, а не Object, используйте следующий код, который изменяет значение `Circle.prototype` так, что оно вместо родового объекта будет содержать объект Shape.

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

Теперь классы Shape и Circle связаны друг с другом отношением наследования, часто называемым *цепочкой прототипов*. На следующей схеме представлены отношения в цепочке прототипов.



Базовым классом, который находится в конце любой цепочки прототипов, является класс Object. Класс Object содержит статическое свойство `Object.prototype`, которое указывает на базовый прототипичный объект для всех объектов, созданных в ActionScript 1.0. Следующим объектом в примере цепочки прототипов является объект Shape. Свойство `Shape.prototype` не задавалось явно, поэтому оно по-прежнему содержит родовой объект (экземпляр класса Object). Последняя ссылка в этой цепочке — класс Circle, связанный со своим прототипом, классом Shape (свойство `Circle.prototype` содержит объект Shape).

Если создается экземпляр класса Circle, как в следующем примере, он наследует цепочку прототипов класса Circle.

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Вспомним пример, в котором свойство `visible` использовалось в качестве члена класса Shape. В этом примере свойство `visible` не существует в объекте `myCircle`, являясь членом только для объекта Shape. Несмотря на это, следующая строка кода возвращает значение `true`:

```
trace(myCircle.visible); // output: true
```

Среда выполнения может установить, что объект `myCircle` наследует свойство `visible`, поднявшись по цепочке прототипов. Выполняя код, среда выполнения сначала ищет свойство `visible` среди свойств объекта `myCircle`, но не находит его. После этого она выполняет дальнейший поиск в объекте `Circle.prototype`, но в нем также не оказывается свойства `visible`. Поднимаясь выше по цепочке прототипов, она наконец находит свойство `visible`, определенное в объекте `Shape.prototype`, и выводит его значение.

В целях упрощения детали и сложности работы с цепочкой прототипов не описываются. Целью является предоставление информации, достаточной для понимания объектной модели ActionScript 3.0.

ActionScript 2.0

В язык ActionScript 2.0 были введены новые ключевые слова, такие как `class`, `extends`, `public` и `private`, которые позволяют определять классы так, как это привыкли делать все, кто работает с языками Java и C++. Важно понимать, что в основе языков ActionScript 1.0 и ActionScript 2.0 лежит один и тот же механизм наследования. В версии ActionScript 2.0 просто добавлены новые элементы синтаксиса для определения классов. Цепочка прототипов работает одинаково в обеих версиях языка.

Новый синтаксис, введенный в ActionScript 2.0, позволяет определять классы более простым для многих программистов способом, как показано ниже.

```
// base class
class Shape
{
var visible:Boolean = true;
}
```

Обратите внимание, что в языке ActionScript 2.0 были введены аннотации типов для использования при проверке типов во время компиляции. Это позволяет объявить, что свойство `visible` в предыдущем примере должно содержать только логическое значение. Новое ключевое слово `extends` также упрощает процесс создания подклассов. В следующем примере, двухэтапный процесс в ActionScript 1.0 выполняется одним действие с помощью ключевого слова `extends`.

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

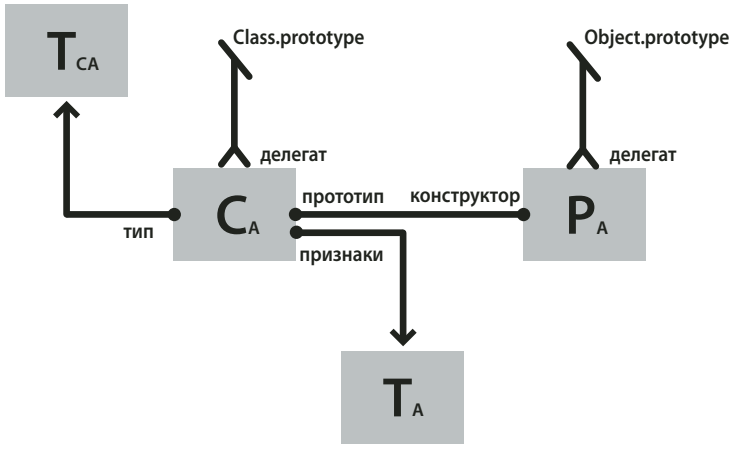
Теперь конструктор объявляется в составе определения класса, и свойства класса `id` и `radius` также должны быть явно объявлены.

В языке ActionScript 2.0 также добавлена поддержка для определения интерфейсов, позволяющая еще больше усовершенствовать объектно-ориентированные программы с помощью формально определенных протоколов для взаимодействия объектов.

Объект класса в ActionScript 3.0

Общая парадигма объектно-ориентированного программирования, которая чаще всего ассоциируется с языками Java и C++, использует классы для определения типов объектов. Языки программирования, применяющие эту парадигму, как правило, используют классы также для создания экземпляров типа данных, определяемого классом. Язык ActionScript использует классы в обеих целях, но, будучи разработанным на базе прототипов, он также имеет еще одну интересную отличительную черту. ActionScript создает для каждого определения класса особый объект класса, который обеспечивает совместное использование поведения и состояния. Однако для многих программистов, работающих с языком ActionScript, это отличие не применяется на практике. Язык ActionScript 3.0 разработан таким образом, что создавать сложные объектно-ориентированные приложения ActionScript можно, не используя этих специальных объектов классов и даже не имея о них никакого понятия.

На следующей схеме показана структура объекта класса, представляющего простой класс A, определенный с помощью инструкции `class A {}`.



Каждый прямоугольник в схеме представляет объект. Каждый объект в схеме имеет подстрочный символ «A», который указывает на то, что он принадлежит классу A. Объект класса (CA) содержит ссылки на ряд других важных объектов. Объект признаков экземпляра (TA) хранит свойства экземпляра, объявленные в определении класса. Объект признаков класса (TCA) представляет внутренний тип класса и хранит статические свойства, определенные классом (подстрочный символ «C» обозначает «класс»). Объектом прототипа (PA) всегда называется объект класса, к которому он изначально был присоединен с помощью свойства `constructor`.

Объект признаков

Объект признаков, впервые введенный в версии ActionScript 3.0, служит для повышения производительности. В предыдущих версиях ActionScript, поиск имени мог занимать много времени, так как проигрывателю Flash Player требовалось пройти всю цепочку прототипов. В ActionScript 3.0 поиск имени выполняется намного эффективней и быстрей, так как наследуемые свойства копируются из суперклассов в объект признаков подклассов.

Программный код не может обратиться к объекту признаков напрямую, но его присутствие повышает производительность и оптимизирует использование памяти. Объект признаков предоставляет для AVM2 подробные сведения о макете и содержимом класса. Располагая такими данными, AVM2 может значительно сократить время выполнения, так как виртуальная машина во многих случаях создает прямые инструкции для обращения к свойствам или непосредственно вызывает методы, не затрачивая времени на поиск имен.

Благодаря объекту признаков, объем памяти, занимаемый объектом, может быть намного меньше, чем у подобных объектов в предыдущих версиях ActionScript. Например, если класс не объявлен как динамический (то есть является фиксированным), его экземпляру не требуется хэш-таблица для динамически добавляемых свойств, и он может содержать лишь указатель на объекты признаков и несколько слотов для фиксированных свойств, определенных в классе. В результате, объект, который требовал 100 байтов памяти в ActionScript 2.0, в версии ActionScript 3.0 будет требовать лишь 20 байтов.

Примечание. *Объект признаков — это внутренняя деталь реализации. Нельзя гарантировать, что он не изменится или совсем не исчезнет в будущих версиях ActionScript.*

Объект прототипа

Каждый объект класса ActionScript имеет свойство `prototype`, которое представляет собой на объект прототипа этого класса. Объект прототипа является наследием ActionScript как языка на базе прототипов. Дополнительные сведения см. в разделе ««История объектно-ориентированного программирования на языке ActionScript» на странице 122».

Свойство `prototype` предназначено только для чтения: его нельзя изменить так, чтобы оно указывало на другие объекты. А в предыдущих версиях ActionScript свойство `prototype` можно было переназначить, чтобы оно указывало на другой класс. Хотя свойство `prototype` доступно только для чтения, это ограничение не распространяется на указываемый им объект прототипа. Другими словами, к объекту прототипа можно добавлять новые свойства. Свойства, добавляемые к объекту прототипа, совместно используются всеми экземплярами класса.

Цепочка прототипов, которая являлась единственным механизмом наследования в предыдущих версиях ActionScript, выполняет лишь второстепенную роли в ActionScript 3.0. Основной механизм наследования, то есть наследование фиксированных свойств, осуществляется на внутреннем уровне объектом признаков. Фиксированное свойство — это переменная или метод, объявленный как часть определения класса. Наследование фиксированных свойств также называется наследованием класса, так как этот механизм наследования связан с такими ключевыми словами, как `class`, `extends` и `override`.

Цепочка прототипов обеспечивает альтернативный механизм наследования, отличающийся от наследования фиксированный свойств большей динамичностью. Объект прототипа класса можно добавлять свойства не только в составе определения класса, но и во время выполнения через свойство `prototype` объекта класса. Однако обратите внимание, что если компилятор запущен в строгом режиме, то свойства, добавленные в объект прототипа, могут быть недоступны, если класс не объявлен с использование ключевого слова `dynamic`.

Рассмотрим присоединение нескольких свойств к объекту прототипа на примере класса Object. Методы `toString()` и `valueOf()` класса Object на самом деле являются функциями, которые назначены свойствам объекта прототипа класса Object. Ниже приводится пример того, как может выглядеть объявление этих методов теоретически (фактическая реализация выглядит по-другому из-за деталей реализации).

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Как говорилось выше, свойство можно присоединить к объекту прототипа класса за рамками определения класса. Например, метод `toString()` можно также определить за пределами определения класса Object, как показано ниже.

```
Object.prototype.toString = function()
{
    // statements
};
```

Однако, в отличие от наследования фиксированных свойств, наследование прототипа не требует ключевого слова `override`, если требуется переопределить метод в подклассе. Например: если требуется переопределить метод `valueOf()` в подклассе класса Object, это можно сделать одним из трех методов. Во-первых, можно определить метод `valueOf()` для объекта прототипа подкласса в определении класса. Следующий код создает подкласс Foo класса Object и переопределяет метод `valueOf()` объекта прототипа Foo в составе определения класса. Так как каждый класс является наследником класса Object, использовать ключевое слово `extends` не требуется.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

Во-вторых, можно определить метод `valueOf()` для объекта прототипа класса Foo в составе определения класса, как в следующем коде.

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

В-третьих, можно определить фиксированное свойство `valueOf()` в составе класса Foo. Этот метод отличается от остальных тем, что в нем объединено наследование фиксированных свойств с наследованием прототипов. Если в каком-либо подклассе класса Foo потребуется переопределить метод `valueOf()`, необходимо использовать ключевое слово `override`. Следующий код демонстрирует определение метода `valueOf()` в качестве фиксированного свойства класса Foo.

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

Пространство имен AS3

В результате существования двух механизмов наследования, наследования фиксированных свойств и наследования прототипов, появляется интересная проблема совместимости в отношении свойств и методов базовых классов. Для совместимости со спецификацией языка ECMAScript, на которой основан ActionScript, требуется наследование прототипов, то есть методы и свойства базового класса должны определяться в объекте прототипа класса. С другой стороны, для совместимости с ActionScript 3.0 требуется наследование фиксированных свойств, то есть свойства и методы базового класса присоединяются через определение класса с помощью ключевых слов `const`, `var` и `function`. Кроме того, использование фиксированных свойств вместо свойств прототипов может значительно повысить производительность во время выполнения.

ActionScript 3.0 решает эту проблему с помощью использования для основных классов как наследования прототипов, так и наследования фиксированных свойств. Каждый базовый класс содержит два набора свойств и методов. Один набор определяется для объекта прототипа с целью обеспечения совместимости со спецификацией ECMAScript, а другой определяется с использование фиксированных свойств и пространством имен AS3 для обеспечения совместимости с ActionScript 3.0.

Пространство имен AS3 обеспечивает удобный механизм выбора одного из двух наборов свойств и методов. Если не используется пространство имен AS3, экземпляр базового класса наследует свойства и методы, определенные в объекте прототипа базового класса. Если используется пространство имен AS3, экземпляр базового класса наследует версии AS3, так как фиксированные свойства всегда имеют более высокий приоритет, чем свойства прототипа. Другими словами, если доступно фиксированное свойство, оно всегда используется вместо одноименного свойства прототипа.

Можно избирательно использовать AS3-версию свойства или метода, указав его с помощью пространства имен AS3. Например, следующий код использует AS3-версию метода `Array.pop()`.

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Также можно использовать директиву `use namespace`, чтобы открыть пространство имен AS3 для всех определений в блоке кода. Например, следующий код использует директиву `use namespace`, чтобы открыть пространство имен AS3 для методов `pop()` и `push()`.

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0 также предоставляет параметры компилятора для каждого набора свойств, чтобы пространство AS3 можно было применить ко всей программе. Параметр компилятора `-as3` обозначает пространство имен AS3, а параметр компилятора `-es` обозначает наследование прототипов (`es` — это сокращение от ECMAScript). Чтобы открыть пространство имен AS3 для всей программы, задайте параметру компилятора `-as3` значение `true`, а параметру компилятора `-es` — значение `false`. Чтобы использовать версии прототипа, задайте параметрам компилятора противоположные значения. По умолчанию в компиляторах Flash Builder и Flash Professional используются следующие настройки: `-as3 = true` и `-es = false`.

Если вы планируете расширять базовые классы и переопределять методы, то необходимо разобраться в том, как пространство имен AS3 может повлиять на способ объявления переопределенного метода. Если используется пространство имен AS3, в любом переопределении метода базового класса также должно использоваться пространство имен AS3 вместе с атрибутом `override`. Если пространство имен AS3 не используется, то при переопределении метода базового класса в подклассе не следует использовать пространство имен AS3 и ключевое слово `override`.

Пример: GeometricShapes

Пример приложения GeometricShapes демонстрирует, как ряд понятий и возможностей объектно-ориентированного программирования можно применить с использованием ActionScript 3.0, включая:

- определение классов,
- расширение классов,
- полиморфизм и ключевое слово `override`,
- определение, расширение и реализацию интерфейсов.

Кроме того, в примере используется «фабричный метод», который создает экземпляры классов, объявляя в качестве возвращаемого значения экземпляр интерфейса, и использует возвращенный объект в общем виде.

Получить файлы приложения для этого примера можно на странице www.adobe.com/go/learn_programmingAS3samples_flash_ru. Файлы приложения GeometricShapes находятся в папке Samples/GeometricShapes. Приложение состоит из следующих файлов.

Файл	**Описание**
GeometricShapes.mxml или GeometricShapes.fla	Основной файл приложения Flash (FLA) или Flex (MXML).
com/example/programmingas3/geometricshapes/IGeometricShape.as	Базовый интерфейс, определяющий методы, которые должны быть реализованы всеми классами приложения GeometricShapes.
com/example/programmingas3/geometricshapes/IPolygon.as	Интерфейс, определяющий методы, которые должны быть реализованы всеми классами приложения GeometricShapes с несколькими сторонами.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Тип геометрической фигуры, равные стороны которого симметрично противоположны относительно центра фигуры.

Файл	Описание
com/example/programmingas3/geometricshapes/Circle.as	Тип геометрической фигуры, определяющей круг.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Подкласс класса RegularPolygon, определяющий равносторонний треугольник.
com/example/programmingas3/geometricshapes/Square.as	Подкласс класса RegularPolygon, определяющий равносторонний прямоугольник.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Класс, содержащий фабричный метод для создания фигур заданного типа и размера.

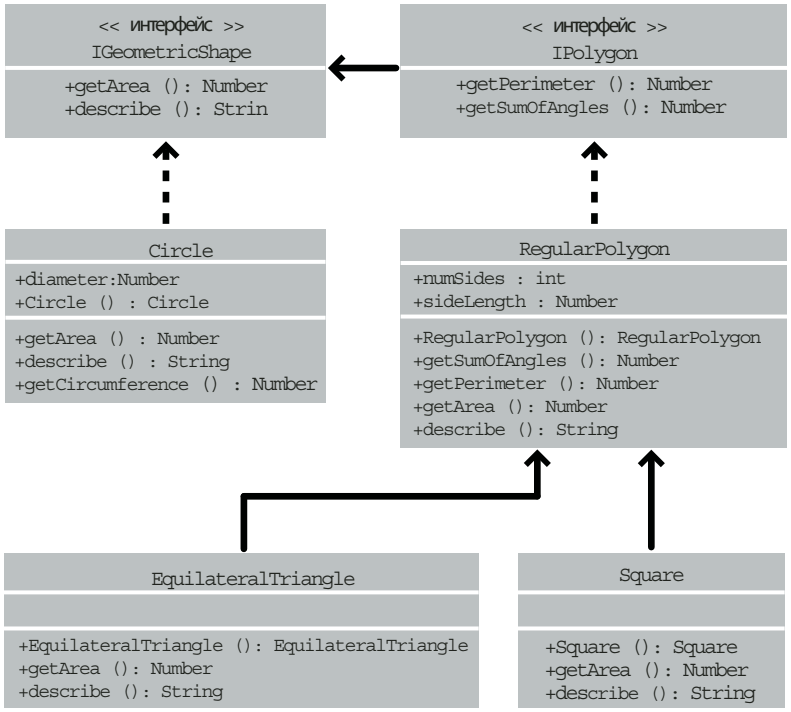
Определение классов GeometricShapes

Приложение GeometricShapes дает пользователю возможность указать тип и размер геометрической фигуры. После этого выдается описание фигуры, ее площадь и ее периметр.

В приложении использован обычный пользовательский интерфейс, включающий несколько элементов управления для выбора типа фигуры, указания ее размера и отображения описания. Самая интересная часть приложения кроется под поверхностью, в структуре самих классов и интерфейсов.

Это приложение работает с геометрическими фигурами, но не отображает их графически.

Классы и интерфейсы, определяющие геометрические фигуры в этом примере, представлены на следующей схеме с использованием нотации UML (Унифицированный язык моделирования).



Классы в примере GeometricShapes

Определение общего поведения с помощью интерфейсов

Это приложение GeometricShapes работает с тремя типами фигур: кругами, квадратами и равносторонними треугольниками. Структура классов GeometricShapes начинается с очень простого интерфейса, IGeometricShape, в котором перечислены методы, общие для фигур всех трех типов.

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

Интерфейс определяет два метода: метод `getArea()`, который вычисляет и возвращает площадь фигуры, и метод `describe()`, который генерирует текстовое описание свойств фигуры.

Также желательно знать периметр каждой фигуры. Однако периметр круга называется окружностью, он вычисляется по особой формуле, поэтому его поведение отличается от поведения треугольника и квадрата. Несмотря на это, между треугольниками, квадратами и другими многоугольниками существует достаточно большое сходство, чтобы для них можно было определить новый класс интерфейса: IPolygon. Интерфейс IPolygon также довольно простой, как показано ниже.

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Этот интерфейс определяет два метода, общие для всех многоугольников: метод `getPerimeter()`, который изменяет совокупный размер всех сторон, и метод `getSumOfAngles()`, который вычисляет сумму всех внутренних углов.

Интерфейс IPolygon расширяет интерфейс IGeometricShape, то есть любой класс, реализующий интерфейс IPolygon, должен объявить все четыре метода: два из IGeometricShape и два из IPolygon.

Определение классов фигур

Разобравшись с методами, общими для каждого типа фигур, можно определить сами классы фигур. С точки зрения количества реализуемых методов, самой простой фигурой является круг (класс Circle), как показано ниже.

```
package com.example.programmingas3.geometricshapes
{
    public class Circle implements IGeometricShape
    {
        public var diameter:Number;

        public function Circle(diam:Number = 100):void
        {
            this.diameter = diam;
        }

        public function getArea():Number
        {
            // The formula is Pi * radius * radius.
            var radius:Number = diameter / 2;
            return Math.PI * radius * radius;
        }

        public function getCircumference():Number
        {
            // The formula is Pi * diameter.
            return Math.PI * diameter;
        }

        public function describe():String
        {
            var desc:String = "This shape is a Circle.\n";
            desc += "Its diameter is " + diameter + " pixels.\n";
            desc += "Its area is " + getArea() + ".\n";
            desc += "Its circumference is " + getCircumference() + ".\n";
            return desc;
        }
    }
}
```

Класс Circle реализует интерфейс IGeometricShape, поэтому он должен включать код для методов `getArea()` и `describe()`. Помимо этого, определяется метод `getCircumference()`, уникальный для класса Circle. Класс Circle также объявляет свойство `diameter`, которого нет у классов многоугольников, так как оно содержит значение диаметра.

Остальные два типа фигур (квадраты и равносторонние треугольники) имеют больше общих свойств: все их стороны равны, а для расчета их периметра и суммы внутренних углов используются общие формулы. В действительности эти распространенные формулы применяются и к другим правильным многоугольникам, которые вам также придется определять в будущем.

Класс RegularPolygon является суперклассом для классов Square и EquilateralTriangle. В суперклассе можно определить все общие методы, чтобы их не нужно было определять по отдельности в каждом подклассе. Ниже приводится код класса RegularPolygon:

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + "
degrees.\n";
            return desc;
        }
    }
}
```

В первую очередь, класс RegularPolygon объявляет два свойства, общие для всех правильных многоугольников: длину каждой стороны (свойство `sideLength`) и количество сторон (свойство `numSides`).

Класс RegularPolygon реализует интерфейс IPolygon и объявляет все четыре метода интерфейса IPolygon. Два из них, `getPerimeter()` и `getSumOfAngles()`, реализуются с помощью общих формул.

Так как для каждой фигуры используется своя формула метода `getArea()`, версия базового класса метода не может содержать общую логику, которую могли бы наследовать методы подклассов. Вместо этого возвращается значение по умолчанию 0, указывающее на то, что площадь не вычислена. Чтобы правильно вычислить площадь каждой фигуры, подклассы класса RegularPolygon должны самостоятельно переопределять метод `getArea()`.

Следующий код для класса EquilateralTriangle демонстрирует переопределение метода `getArea()`.

```
package com.example.programmingas3.geometricshapes
{
    public class EquilateralTriangle extends RegularPolygon
    {
        public function EquilateralTriangle(len:Number = 100):void
        {
            super(len, 3);
        }

        public override function getArea():Number
        {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
        }

        public override function describe():String
        {
                /* starts with the name of the shape, then delegates the rest
                of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
        desc += super.describe();
        return desc;
        }
    }
}
```

Ключевое слово `override` указывает на то, что метод `EquilateralTriangle.getArea()` намеренно переопределяет метод `getArea()` суперкласса RegularPolygon. Когда вызывается метод `EquilateralTriangle.getArea()`, он вычисляет площадь по формуле из предыдущего кода, а код `RegularPolygon.getArea()` никогда не выполняется.

Класс EquilateralTriangle не определяет собственную версию метода `getPerimeter()`. Когда вызывается метод `EquilateralTriangle.getPerimeter()`, этот вызов передается вверх по цепочке наследования и выполняет под в методе`getPerimeter()` суперкласса RegularPolygon.

Конструктор `EquilateralTriangle()` использует инструкцию `super()`, чтобы сделать явный вызов конструктора `RegularPolygon()` своего суперкласса. Если бы оба конструктора имели одинаковый набор параметров, можно было бы совсем опустить конструктор `EquilateralTriangle()`, а вместо этого вызывался бы конструктор `RegularPolygon()`. Однако конструктору `RegularPolygon()` требуется дополнительный параметр, `numSides`. Поэтому конструктор `EquilateralTriangle()` вызывает метод `super(len, 3)`, который передает параметр ввода `len` и значение 3, указывающее на то, что у фигуры будет три стороны.

В методе `describe()` также используется оператор `super()`, но другим способом. Он используется для вызова версии суперкласса RegularPolygon метода `describe()`. Метод `EquilateralTriangle.describe()` сначала задает в качестве значения строковой переменной `desc` предложение, описывающее тип фигуры. Затем он получает результаты метода `RegularPolygon.describe()`, вызвав `super.describe()`, и добавляет результат в строку `desc`.

Здесь не приводится подробное описание класса Square, но он похож на класс EquilateralTriangle, в котором есть конструктор и собственные версии методов `getArea()` и `describe()`.

Полиморфизм и фабричный метод

Набор классов, в которых используются интерфейсы и наследования, может применяться для достижения разных интересных целей. Например, все классы фигур, описанные выше, либо реализуют интерфейс IGeometricShape, либо расширяют реализующий его суперкласс. Поэтому если переменная определена в качестве экземпляра IGeometricShape, совершенно не нужно знать, является ли она на самом деле экземпляром класса Circle или Square, чтобы вызвать ее метод `describe()`.

Следующий код демонстрирует, как это происходит.

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

Когда вызывается метод `myShape.describe()`, он выполняет метод `Circle.describe()`, так как несмотря на то, что переменная определена как экземпляр IGeometricShape, ее базовым классом является Circle.

Этот пример демонстрирует принцип полиморфизма в действии: один и тот же метод может вызывать результаты другого выполняемого в данный момент кода в зависимости от класса того объекта, для которого вызывался метод.

Приложение GeometricShapes применяет этот тип полиморфизма на базе интерфейса, используя упрощенную версию шаблона, известного под названием «фабричный метод». Термин *фабричный метод* обозначает функцию, которая возвращает объект, базовый тип данных или содержимое которого могут быть разными в зависимости от контекста.

Описанный здесь класс GeometricShapeFactory определяет фабричный метод с именем `createShape()`.

```
package com.example.programmingas3.geometricshapes
{
    public class GeometricShapeFactory
    {
        public static var currentShape:IGeometricShape;

        public static function createShape(shapeName:String,
                                           len:Number):IGeometricShape
        {
            switch (shapeName)
            {
                case "Triangle":
                    return new EquilateralTriangle(len);

                case "Square":
                    return new Square(len);

                case "Circle":
                    return new Circle(len);
            }
            return null;
        }

        public static function describeShape(shapeType:String, shapeSize:Number):String
        {
            GeometricShapeFactory.currentShape =
                GeometricShapeFactory.createShape(shapeType, shapeSize);
            return GeometricShapeFactory.currentShape.describe();
        }
    }
}
```

Фабричный метод `createShape()` позволяет конструкторам подклассов фигур определять детали создаваемых ими экземпляров и возвращать новые объекты в качестве экземпляров IGeometricShape, чтобы они проходили более общую обработку в приложении.

Метод `describeShape()` в предыдущем примере демонстрирует, как приложение может использовать фабричный метод для получения общей ссылки на более конкретный объект. Приложение может получить описание для только что созданного объекта Circle, как показано ниже.

```
GeometricShapeFactory.describeShape("Circle", 100);
```

Затем метод `describeShape()` вызывает фабричный метод `createShape()` с теми же параметрами, сохраняя новый объект Circle в статической переменной `currentShape`, которая была задана в качестве объекта IGeometricShape. После этого вызывается метод `describe()` объекта `currentShape`, и в результате разрешения этот метод автоматически выполняет метод `Circle.describe()`, возвращая подробное описание круга.

Расширение примера приложения

Настоящая сила интерфейсов и наследования становится очевидной, когда требуется расширить или изменить приложение.

Допустим, требуется добавить новую фигуру, пятиугольник, в этот пример приложения. Для этого нужно создать класс Pentagon, который расширяет класс RegularPolygon и определяет собственные версии методов `getArea()` и `describe()`. После этого следует добавить новый элемент «Pentagon» (Пятиугольник) в комбинированное поле пользовательского интерфейса приложения. Вот и все! Класс Pentagon автоматически получает методы `getPerimeter()` и `getSumOfAngles()`, наследуя их от класса RegularPolygon. Так как класс Pentagon является потомком класса, который реализует интерфейс IGeometricShape, его экземпляр также можно создать в качестве экземпляра IGeometricShape. Это означает, что для добавления нового типа фигуры не требуется изменять сигнатуру методов класса GeometricShapeFactory (и, следовательно, не требуется изменять коды, использующие класс GeometricShapeFactory).

В качестве упражнения, попробуйте добавить класс Pentagon в пример GeometricShapes, чтобы оценить, насколько интерфейсы и наследование упрощают добавление новых возможностей в приложение.